Table of Contents

1. Introduction

Welcome to the concepts book of the *assignment* **Coding**. The chapters of this assignment will teach you the skills required to follow the different lectures and to solve the exercises. Even more, though, they will enable you to automate many of the tasks you do on a daily basis - be it at work or privately.

1.1 Overview

You will learn to read (and partially write) code in a few different languages:

- JavaScript
- Shell scripts
- Python
- C
- Assembly
- Java
- Visual Basic for Applications (VBA)

In order to have a common foundation to build upon and in order to learn about each of these languages more quickly, I will first introduce you to the building blocks of formal languages. Please don't be scared by the theory, we'll get over it relatively quickly.

You will also be introduced to some Unix tools which will make your life easier and more enjoyable. And you will learn to love regular expressions - they're probably my #1 time saver!

1.2 Additional Resources

Where appropriate, I will add links to some external references at the beginning of a chapter. You can use these to e.g. get alternative explanations or further information. You are not required to make use of these at all, but they may be helpful to you.

If you're looking for an easy way of solving more exercises, I recommend you head over to exercism[1] . The platform offers exercises with automated grading for most of the languages we are looking at.

[1] https://exercism.org

1.3 Feedback

Please let me know of any issues you find with the chapters that make up this course. It's very well possible that

- I got something wrong or missed information I should have been giving you
- The explanations could be better - either in general or for your personal way of learning

Even if it turns out that you misunderstood something, we get two positive results:

- You get to learn something
- Future students get improved scripts

Also, please let me know if you later discover something you would have liked me to have covered or external resources you believe should be linked. I will be happy to give you access to updated versions of this script if you're interested.

1.4 Time Planning

Each chapter has a timetable which should help you with how much time you're roughly expected to spend on each part. It's very well possible that you'll take more time on some chapters - particularly if you haven't had any prior training in formal languages.

Each chapter will culminate in a *journeyman's piece*, which is meant to let you evaluate how much of an understanding you've gained of the concepts and methods conveyed in the chapter.

- $1/4$ to $1/3$ will be theory - teaching you the relevant concepts
- $2/3$ to $3/4$ will be practice - teaching you the relevant methods

1.5 Structure

What you're holding in your hands is the concept book. You will get each chapter's content in the form of either an on-screen version or a printable booklet which you can add to the cover to create a complete book.

There are two more books:

- one containing the exercises that accompany the concepts and which you're invited to complete.
- one containing possible solutions to the exercises - you will receive these over time.

2. Formal Languages

2.1 Introduction

Programming languages are *formal languages*: languages following a strict and well-defined grammar, with each word describing a defined concept. This particularly means that unlike in *natural languages*, there is no ambiguity as to what a given expression represents.

In order to understand programs, this chapter will first introduce you to the basic concepts which make up programming languages, which will allow you to more quickly understand specific programming languages. If you're not into "theoretical" concepts very much, but would rather be able to execute code right away, feel free to skip ahead to other languages and come back later. For some, this chapter will be a good introduction, for others it will serve as a reference, and for others still - it will seem rather useless.

2.2 Timetable

Topic	Theory [minutes]	Practice [minutes]	Total [minutes]
Boolean Logic	20	60	***80***
Pseudocode	55	135	***190***
Total		**195**	**270**

2.3 Boolean Logic

2.3.1 Background

In programming, a lot revolves around evaluating whether or not something is right or wrong / correct or incorrect / true or false.

Conditions are expressed using Boolean expressions [1] , and conditions are one of the basic building blocks of programming languages. We will thus get started by getting to know the basics of Boolean logic.

2.3.2 Basics

In the most simple terms, there are two different values we concern ourselves with in logic: `true` / `false`.

We then work with these by combining them using logical *operators* such as `NOT`, `AND` and/or `OR`.

2.3.3 Symbols

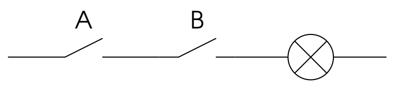
`AND` is formally expressed as $\wedge$, `OR` as $\vee$, and `NOT` as $\neg$.

2.3.4 Analogy: Electricity

If you've worked with electricity before, there's an easy analogy.

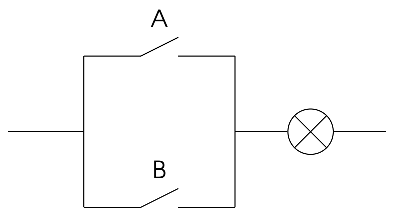
Electricity flowing through a closed switch: `true`. An open switch: `false`.

`AND` then is two switches connected serially, so the bulb will light up if both are closed.



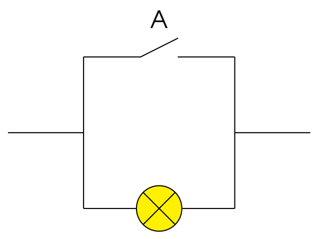
$$A \wedge B$$

`OR` is those same switches, but parallel to each other. The bulb will light up if either one is closed.



$$A \vee B$$

`NOT` can be thought of as a light bulb switched in parallel to the negated expression. Because its resistance is greater than that of the wires and switches combined, it will only light up when current cannot flow through the circuit it is parallel to.



$$\neg A$$

[1] https://en.wikipedia.org/wiki/Boolean_expression

2.3.5 Truth Tables

When “operating” on Boolean values, a convenient way of representing outcomes is a truth table.

2.3.5.1 Truth Table for `AND`

To show the output an `AND` operator produces given two inputs A and B, we can construct the following table:

A	B	$A \land B$	Example
`true`	`true`	`true`	A: lava is hot, B: snow is cold
`true`	`false`	`false`	A: lava is hot, B: snow is hot
`false`	`true`	`false`	A: lava is cold, B: snow is cold
`false`	`false`	`false`	A: lava is cold, B: snow is hot

2.3.5.2 Truth Table for `OR`

Doing the same for `OR`, we can construct the according table as well. Going forward, we will use `T` for `true` and `F` for `false`.

A	B	$A \lor B$
`T`	`T`	`T`
`T`	`F`	`T`
`F`	`T`	`T`
`F`	`F`	`F`

2.3.5.3 Truth Table for `NOT`

A very simple, yet very powerful operator, $\neg$ turns one Boolean value into the other.

A	$\neg A$
`T`	`F`
`F`	`T`

2.3.6 Operator Precedence

Remember how in maths multiplication and division are always “stronger” than addition and subtraction? Like when in `1 + 3 * 2` the `3 * 2` part is evaluated first, resulting in the expression to be simplified to `1 + 6`, which then results in `7`?

Similarly, $\wedge$ is “stronger” than $\vee$, so if you want to say “either A or B need to be true and C must be true”, you would have to write $(A \vee B) \wedge C$. If you don’t add the parenthesis, you’re saying “either A must be true or then both B and C must be true”.

Last but not least, $\neg$ is strongest of all - similar to a negative sign in front of a number.

Time to exercise!

2.3.7 Intermezzo: What’s All This For?

So why are we looking at Boolean values, operators and truth tables in the first place? We said they were the basis of programming, but what do we use them for? Let’s start with an example, say, programming a traffic light. What is the logic for turning to a specific signal (we’ll use the US example since it’s a bit simpler)?

- Turn to green if the signal was red and 30 seconds have passed
- Turn to orange if the signal was green and 30 seconds have passed
- Turn to red if the signal was orange and 5 seconds have passed

That’s rather hard to read, isn’t it? How about this:

$$next = \begin{cases} GREEN \text{ if } & RED \wedge T \geq 30 \\ ORANGE \text{ if } & GREEN \wedge T \geq 30 \\ RED \text{ if } & ORANGE \wedge T \geq 5 \end{cases}$$

I would argue that’s a bit easier to read and get a quick overview of, wouldn’t you?

2.3.8 Morgan’s Law

So far, we’ve looked at very simple expressions, but of course there’s much more complicated logic out in the wild. That’s where Morgan’s law comes in.

What it law says is essentially this: if you negate a (complex) Boolean expression, you can transform it by changing each operator to its counterpart ($\wedge \leftrightarrow \vee$) and negating each individual value. Please be careful not to remove multiple parentheses at once. One’s often best off by starting with the innermost negated parenthesis.

> When an `AND` becomes `OR`, you may have to add brackets to retain the meaning: $\wedge$ binds more strongly than $\vee$!

2.3.8.1 Examples

Let’s have a look at some examples before you head over and exercise yourself!

$$\begin{aligned}
\neg(A \wedge B) &\rightarrow \neg A \vee \neg B\\
\neg(A \vee B) &\rightarrow \neg A \wedge \neg B\\
\neg(A \vee \neg B) &\rightarrow \neg A \wedge B\\
\neg(A \vee \neg B \wedge C) &\rightarrow \neg A \wedge B \vee \neg C
\end{aligned}$$

And a bit more complicated

$$\neg(A \wedge B \wedge \neg(C \vee D)) \rightarrow \neg A \vee \neg B \vee C \vee D$$

Boolean expressions are also distributive [2] :

- $A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C)$

Time to exercise!

[2] https://en.wikipedia.org/wiki/Distributive_property

2.3.9 Simplifying Expressions

Sometimes we don't realize we could say things more simply than we do. Let's look at simplifying some expressions!

$$\begin{aligned} A \wedge \neg A &\rightarrow false \\ A \vee \neg A &\rightarrow true \\ A \vee B \wedge A &\rightarrow A \\ (A \vee \neg A) \wedge B &\rightarrow B \end{aligned}$$

This may be the right place to introduce yet another operator: `XOR`. It says that exactly one argument must be true. It's denoted $\veebar$, so we would write $A \veebar B$. Just one more: to denote that two variables must be equal, we can write $A \Leftrightarrow B$

Time to exercise!

2.4 Pseudocode

Now that we have covered the basics, we can make use of these to start with some actual programming - well, almost. Because we are going to look at quite a few different programming languages, we will introduce a generalized language and a few tools that help you deal with complex control flow (that's the "path" a program takes: the step by step instructions executed).

Every programming language has a (at least slightly) different syntax. Because of this, we will use a generalized syntax to always compare to. You can look at it as *pseudocode*: it is not meant to be interpreted by a machine, but is only aimed at conveying the intention of a program to a human reader. The nice thing about pseudocode is that you can leave away things that aren't relevant to the point you're making. You also don't have to adhere to any rules in general - so long as you get across the intention.

In order to skip certain aspects which are not relevant for a given example, we will simply describe a number of steps in a comment. In our syntax, comments will start with two forward slashes (`//`) followed by text. Comments can generally be used to add information that is not executed by the computer - it's directed at human readers.

If you don't feel comfortable with pseudocode (or if you already know a programming language) and would rather be able to execute code to "feel" what it does, you can choose one of the languages we'll look at and you should find an example for the concepts in this chapter which compiles and can be run.

2.4.1 Input and Output

2.4.1.1 Printing

To output information to the user, we'll use the `print` instruction (a *method* as we will see later). We will look at it as something built into the language, an instruction which simply exists for us to use.

Pseudocode

```
print("Hi, there!")
```

2.4.1.2 Reading

On the other hand, to get information from the user, we may need to read their input. We'll use the built-in `read` instruction for that.

Pseudocode

```
constant x = read()
```

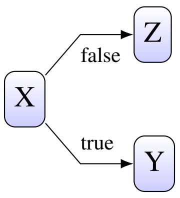
2.4.2 Control Structures

Sometimes it helps to visualize the control flow of a program. We will use simple graphs - control flow diagrams - to give you a visual of control flows.

2.4.2.1 Conditions

Probably the most important control structure - the one that is most likely present in every programming language - is the condition. With conditions, you can tell the computer “If X is the case, do Y, otherwise do Z.”

As you may have noticed, the `X` above is a Boolean expression, which visualized looks like this:



The same thing in pseudocode:

Pseudocode

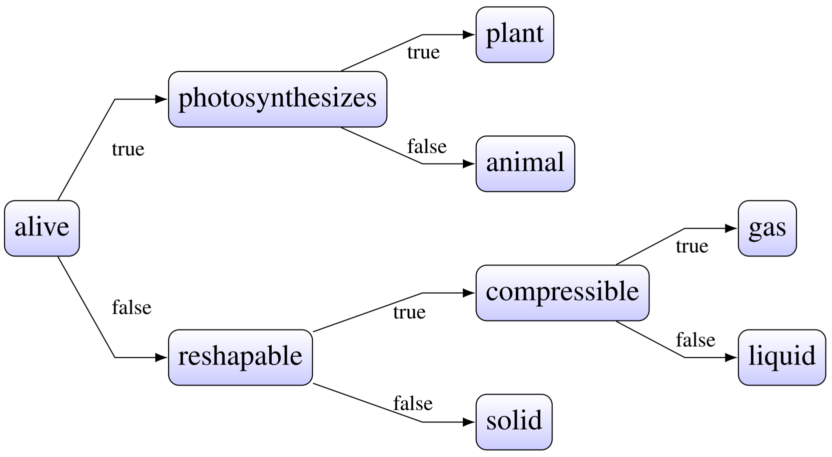
```
if x < 5 then

  print("smaller")
else
  print("greater or equal")
end if
```

Let's look at a more complex example: classifying objects.

Pseudocode

```
if alive then
  if photosynthesizes then
    // plant
  else
    // animal
  end if
else
  if reshapable then
    if compressible then
      // gas
    else
      // liquid
    end if
  else
    // solid
  end if
end if
```

As a diagram, this might look as follows:



If this looks a bit like a fallen tree to you, you're quite observant! This is why this kind of structure is also called a *decision tree*.

Time to exercise!

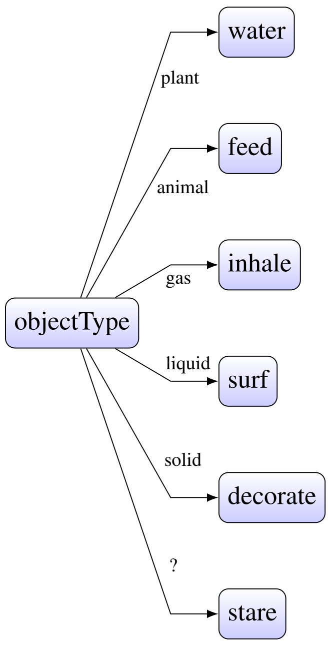
2.4.2.2 Case Matching

It often happens that we don't have binary decisions (`true` or `false`) we're concerned with but rather have different cases to look at. For example, once we have decided whether our object is a plant, animal, gas, liquid or solid, we may want do different things based on this. We could of course do this with a battery of `if` and `else`, but that would not be very readable. For this we use *case matching* (also often called "switch statement"):

Pseudocode

```
if objectType
  matches plant  then
    print("water it")

  matches animal then
    print("feed it")

  matches gas    then
    print("inhale it")

  matches liquid then
    print("surf it")

  matches solid  then
    print("decorate it")

else
    print("stare at it")
endif
```

As a diagram, this might look as follows:



Time to exercise!

2.4.2.3 Loops

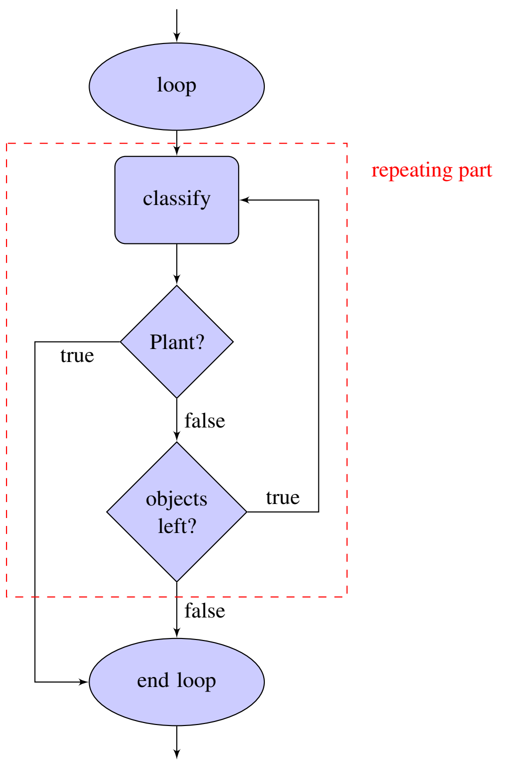
Loops allow repeatedly executing a block of code. Let's have a look at an example:

Pseudocode

```
loop // over objects
  // classify current object
  exit if plant
  // advance to the next object
end loop
```

The above example will classify a given list of objects and will exit the loop once an object has been classified as a plant. The code executed repeatedly (the part between `loop` and `end loop`) is omitted and has been replaced by a comment.

Drawing this in a diagram, we get the following:



Time to exercise!

2.4.3 Data Structures

One thing we graciously overlooked so far is that we require more than just logic and control flow, we typically also need to access data. When we decided whether or not to exit a loop above, we just wrote `exit if plant`. But what is this `plant`?

2.4.3.1 Constants and Variables

When a program executes, information is typically stored in *constants* and *variables*. As the name suggests, variables can be modified, whereas constants cannot be changed.

Let's look at a loop calculating a factorial [3] to get a feel for variables and constants:

Pseudocode

```
 1 constant input = 3
 2
 3 variable result = 1
 4 variable counter = 1
 5
 6 loop
 7   result = result * counter
 8   exit if counter == input
 9   counter = counter + 1
10 end loop
```

At the end of the program, our variable `result` will contain the factorial of our constant `input`.

On lines 3 and 4, we *declare* variables `result` and `counter` and immediately assign them a value. Very basically, this means that we create an information store, give it a name and store a value in it.

We've already seen that we can draw a diagram in order to easily follow the control flow of a program. Another great tool to check correctness of a program is to track variables during the program execution. Here's a table which tracks `result` and `counter` for the above and also shows the evaluation of the `exit` condition of the loop (values before line executed).

Following the control flow of our little program, we can observe how these variables are re-assigned (given new values) based on their previous values:

Step	Line	counter	result	In Loop	counter == input
1	3	n/a	n/a	F	F
2	4	n/a	1	F	F
3	6	1	1	F	F
4	7	1	1	T	F
5	8	1	1	T	F
6	9	1	1	T	F
7	7	2	1	T	F
8	8	2	2	T	F
9	9	2	2	T	F
10	7	3	2	T	T
11	8	3	6	T	T
12	10	3	6	F	T

Looking at the table, do you notice anything?

Time to exercise!

[3] https://en.wikipedia.org/wiki/Factorial

2.4.3.2 Arrays

To take this one step further, imagine you want a program to multiply a list of numbers. With only *scalar* values, you wouldn't get too far. This is where arrays come into play. They are declared as follows: `constant input = [2, 4, 9, 1, 7]`. One of the most important properties of arrays is that they preserve the order of their elements, which is why individual values can be accessed by pointing to their index: `input[0]` (which in the example is `2`).

> Arrays typically are *0-indexed*, which means that the above array has content at indexes 0 to 4 (we will stick to this convention because it is the most commonly observed one).

Arrays also normally have a *property* `length`, which contains the number of elements in them. We can access this property via `input.length` (which is `5` in this example).

To add elements to an array, we will simply write to the index we want a value at (`input[5] = 18`). Any elements between the previously last one and the new one will simply be empty.

Time to exercise!

2.4.3.3 Maps

The second core data structure we will look at is a *map*. Maps provide mappings (or translations, which is why they are sometimes called *dictionaries*) between a key and a value. The following example declares a mapping between names of geometrical shapes and the number of their corners.

To access an element, one uses their key the same way as using an array index: `shapeCorners["Triangle]` equals 3.

Pseudocode

```
constant corners = {
  "Triangle":  3,
  "Rectangle": 4,
  "Pentagon":  5,
}
print(corners["Triangle"]) // 3
```

Because maps are defined as a mapping between key and value, duplicate keys don't make sense in a map. If they exist, we will consider that an error, but many programming languages just have the last entry added with a given key prevail.

Time to exercise!

2.4.4 Program Structures

So far, we've looked at how we can express what might be called "local" control flow. It's time to take this a step further and to actually make these programs useful.

2.4.4.1 Methods

Most programs (all languages we'll look at) are structured into building blocks called *methods*. Methods can (but needn't) have *input parameters* (or simply *parameters*), which allow passing information to them. Thus, instead of simply declaring a constant `input` as we did so far (which of course makes very little sense: if we know what the input is, we can simply hard-code the value `result` will have at the end), we can pass this information to a method via a parameter (we say we pass an *argument* to the method).

In our pseudo-code, we will declare methods very similarly to constants and variables, such as:

Pseudocode
```
method calculateFactorial = (input) => {
  variable result = 1
  variable counter = 1

  loop
    result = result * counter
    exit if counter == input
    counter = counter + 1
  end loop
}
```

The missing bit of course is about getting information **out** of a method. What use is it to calculate a factorial if we're not going to do anything with it? Methods need to be able to *return* values as well.

You may wonder why we need procedures in the first place if they don't return any values. One straight forward example is output of information, e.g. to the hard disk or to the user's screen.

Methods which return values are called *functions*. Those that don't, are called *procedures*.

Here's the above method again, now clearly a function:

Pseudocode
```
method calculateFactorial = (input) => {
  variable result = 1
  variable counter = 1

  loop
    result = result * counter
    exit if counter == input
    counter = counter + 1
  end loop

  return result
}
```

To execute this function, we can simply write `calculateFactorial(5)`.

Time to exercise!

2.4.5 Data Types

We’re almost at the end! There’s one more concept we should discuss: data types. Some of them, we introduced explicitly, like arrays (e.g. [2, 4, 9, 1, 7]) and maps (e.g. { “Triangle”: 3, “Rectangle”: 4, “Pentagon”: 5 }). Others, we just took for granted, like strings (e.g. “Heptagon”) and numbers (e.g. 1).

Why do we even need to talk about this? Because operations that are available on variables depend on the variables’ types. For example, what would “Heptagon”++ do? Wouldn’t it be cool if that gave us “Octagon” ;-)? Below is a table of the operators which are supported in our pseudocode.

Operator	Operand 1	Operand 2	Operation	Example
`=`	any		Assignment	`a = 3`
`==`	any	any	Equals	`a == 4`
`+`	String	String	Concatenation	`"Hello" + "you"` → `"Hello you"`
`+`	String	Number	Concatenation	`"It's" + 42 + "!"` → `"It's 42!"`
`+`	Number	Number	Addition	`2 + 4` → `6`
`-`	Number	Number	Subtraction	`5 - 3` → `2`
`++`	Number		Increment	`5++` → `6`
`--`	Number		Decrement	`5--` → `4`
`>`	Number	Number	Greater than	`a > b`
`>=`	Number	Number	Greater or equal than	`a >= b`
`<`	Number	Number	Smaller than	`a < -1`
`<=`	Number	Number	Smaller or equal than	`a <= c`
`&&`	Boolean	Boolean	`AND` / $\wedge$	`if A && B then`
`||`	Boolean	Boolean	`OR` / $\vee$	`if A || B then`
`.length`	Array		# elements	`if arr.length > input then`
`[n]`	Array		element at index `n`	`result = result * input[counter]`
`[key]`	Map		element with key `key`	`result = result * input[key]`

2.4.5.1 Method Signatures

Now that we know about data types, we have the means to discuss what method *signatures* are. They are the contract between the method and its users: they state what parameters the method accepts and - if it is a function - what it returns.

As an example, the signature of our above method `calculateFactorial` is `calculateFactorial(input: number) => number`.

3. Regular Expressions

3.1 Introduction

Regular expressions (or *regexes*) define a pattern that can be used to search for strings within longer text: a search pattern. You can think of it as text search on steroids. If you haven't used regular expressions so far, your life will have changed for the (significantly) better in a bit less than two hours.

Why are we teaching regular expressions in a coding module? First off, they can be very powerful as part of your code (e.g. to find interesting sections in log files), but also for you to search and modify your own code or any other textual information.

3.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Matching	40	130	***170***
Replacing	10	30	***40***
Total		**160**	**210**

3.3 Resources

- Self-proclaimed "Premier website about Regular Expressions" [1]

[1] https://www.regular-expressions.info/

3.4 Matching

Let's look at examples of what we might want to search for and introduce how to construct the regular expressions used to find it.

To try by yourself (which you definitely should!), there are multiple options:

- Start up any (somewhat capable) text editor locally, e.g. `vi` or `Visual Studio Code` (which you'll most likely be using later anyway) and use regular expressions to search.
- Go to an online regular expression site such as regex 101 [2] . The advantage is that such sites typically also do a good job explaining what is happening, so this is the recommended option if you're online.

> Whenever you don't understand how an expression is being parsed [a] , it helps to enter it into regex 101.

3.4.1 Literal matching

The most simple thing to match is probably a single character or a literal string. If, for example, we are looking for the symbol `@` somewhere in a text, our regular expression to match it will be `/@/`. The two forward slashes (`/`) around the `@` are delimiters used to indicate where the regular expression starts and ends.

To match a longer sequence of characters, we can simply put that between delimiters, e.g. `/loop/` to match an occurrance of the word `loop`.

Time to exercise!

3.4.2 Alternatives

What do we do if we want to match one of a selection of different literal strings? We can use alternatives. To look for either a loop or a condition in our pseudocode syntax, we can use `/if|loop/`. The pipe (`|`) character will serve as a separator between options.

Time to exercise!

[2] https://regex101.com/
[a] https://en.wikipedia.org/wiki/Parsing

3.4.3 Word Boundaries

You may have noticed that e.g. `if` was matched as part of longer words. This is expected, we never said we wanted to match entire words only! If we do, we need to add word boundaries to our expression: `/\bif\b/`.

> *Words* are defined to consist of only letters, digits and the underscore (`_`) character.

What is this `\b`, though? It's a so-called *meta-character*. We will encounter quite a few more of these - they're used to match common patterns that would otherwise have to be constructed manually (word boundary can indeed be expressed in another way as well).

> Meta-characters are not interpreted the same in every environment. As an example, `\b` describes a boundary that separates ASCII-letters, digits and underscore from anything else. In *Python 3* and *.NET*, for example, instead of ASCII-letters, any Unicode letter can be used. We will use the *PCRE* [a] *2* standard - any differences will be mentioned in the language specific sections later.

Time to exercise!

[a] https://www.pcre.org/

3.4.4 Whitespace

You may now think “well, why don’t we simply search for `/ if /`? You’d be right to assume that a space will be seen as any other literal character and we would would indeed not match e.g. ”g**if**t” any longer. The issue is, though, that while some programmers do use spaces to indent, others use tabs, so we should probably make sure to match tabs, too.

There is another meta-character that does exactly that: `\s` (or `[[:space:]]`) matches any whitespace. So what kinds of white-space are there?

3.4.4.1 Horizonatal Whitespace

Whitespace Pattern	Meaning
` `	Simple space
`\t`	Tab

There is an additional meta-character which can be used to match any horizonatl whitespace: `\h` or `[[:blank:]]`.

3.4.4.2 Vertical Whitespace

Whitespace Pattern	Meaning
`\r`	Carriage return
`\n`	Line feed
`\f`	Form feed
n/a	Vertical tab

To match any vertical whitespace, one can use the additional meta-character `\v`.

> Vertical tab does not have a meta-character in PCRE2 - probably because `\v` is used for *any vertical whitespace*.

Time to exercise!

3.4.5 Character Classes

3.4.5.1 Single Character

If instead of matching a literal string, we want to find slightly differing patterns - say, e.g. “cat”, “rat”, “hat”, and “pat”, we can use *sets*: `/[chpr]at/` will match these. Any character within the brackets (`[]`) will match.

3.4.5.2 Ranges

If we need all characters within a given range, we can simplify even further and write e.g. `[a-z]` to match any lower case character. We can do the same for digits and upper case letters: `[0-9]` / `[A-Z]`. And last but not least, we can also combine all of them: `[0-9a-zA-Z]`.

3.4.5.3 Letter & Digit Classes

There are also shorthands for the complete ranges and for words.

Character Class	Description	Corresponding meta-character
[[:lower:]]	Lowercase letters	
[[:upper:]]	Uppercase letters	
[[:alpha:]]	Letters	
[[:digit:]]	Digits	`\d`
[[:alnum:]]	Letters and digits	
[[:word:]]	Word characters (letters, digits, underscore)	`\w`

3.4.5.4 Word Boundaries

To match the start or end of a word explicitly, we can use the following character classes:

Character Class	Description
[[:<:]]	Start of word
[[:>:]]	End of word

Time to exercise!

3.4.6 Quantifiers

3.4.6.1 Optional

Let's move on to something a bit more advanced. Up to now, if we want to match either `duck` or `ducks`, we would have to write e.g. `/duck|ducks/`. That seems a bit redundant again, doesn't it? To simplify, we can make the `s` optional: `/ducks?/`. The questionmark (`?`) marks the preceding element as optional, in this case the `s`.

3.4.6.2 Zero or More

Sometimes one wants to match a pattern that contains an arbitrary number of occurrances of some sub-pattern. We could e.g. want to match lines up to and including an `@`, we can search for `/^[^@]*@/m` (note that the first caret means "start of line", while the one inside the brackets denotes a negation). Having a star (`*`) follow a character, range or group means it may occur between zero and infinite times.

3.4.6.3 One or More

Building on the previous example, if we want to find e-mail handles, we will expect at least one character to preced the `@`. We will thus rather use `/^[^@]+@/m`, with the plus (`+`) meaning between one and infinite times.

3.4.6.4 Between M and N

If we need a specific number of occurrances, we can use yet another notation, namely comma separated integers within braces denoting minimum and maximum cardinality (`{m,n}`). The maximum (`n`) can be left blank to allow infinite matches (`{m,}`). To match any exact number, only that number will be between braces: `{m}`.

As an example, we can match `cool, cool, cool` or `cool, cool, cool, cool` (but not e.g. `cool` or `cool, cool` or `cool, cool, cool, cool, cool` - not completely, that is) using `/cool(, cool){2,3}/`

> Note that there mus be **no space** within the braces!

3.4.6.5 Greedyness

It's very important to note that quantifiers will always try to match the most possible characters that will still produce a match. Thus, `/<.*>/` will match the entirety of `<a href="...">link</a>` instead of just `<a href="...">` as you might expect - it matches *greedily*: as much as possible. We saw above that you can use `/<[^>]*>/` to solve this problem.

There is another solution as well, though: use *lazy* quantifiers. You do that by adding a question mark directly after them, so e.g. `*?` instead of `*`. In our example, we would thus use `/<.*?>/` to only match the first tag.

Time to exercise!

3.4.7 Wildcard

The dot (`.`) can be used to match any character (except newline). In our example with rats, cats, . . . we could alternatively use `/\b.at\b/` if we just want to match any three letter word ending in “at”.

Time to exercise!

3.4.8 Negation

To match everything that does **not** correspond to a given character class, a *cartet* (`^`) is included first within the brackets. Thus, `/^[a-z]/` or `/[^[:lower:]]/` will match anything that is not a lowercase character.

`/[^^]/` matches anything that is not a caret.

There are also a few meta-characters that are the negation of others:

Positive	Negation	Meaning (of negation)
`\s`	`\S`	Any non-whitespace character
`\d`	`\D`	Any non-digit
`\w`	`\W`	Any non-word character

Time to exercise!

3.4.9 Escaping Characters

If we need to match a character which has meaning in regular expressions, such as e.g. `(`, we can prefix it with a backslash to *escape* it, so we’d use `\(` to match a literal opening parenthesis.

To match the backslash (`\`) as part of a character class pattern (in brackets), it has to be escaped itself (`\\`). If it isn’t, it will escape the following character.

Time to exercise!

3.4.10 Groups

Now what if we want to match either `end if` or `end loop`? We can of course Write `/end if|end loop/`, but that redundanct `end` hurts one's eyes, doesn't it? On the other hand, we can't just write `/end if|loop/` (why not?).

In order to get what we want, we need an additional concept: *groups*. Groups are delimited by opening and closing parenthesis (`()`). For our example, we can write `/end\s(if|loop)/`.

3.4.10.1 Capturing and Backreferences

By default, every group will *capture* its content and allow one to reuse what it matches by referring to the nth capturing group using `\n` (so the second group would be referenced as `\2`). As an example, if we want to match an entire HTML tag (from opening to closing), we could use something like `/<([^>]*).*>.*<\/\1>/`.

> `\0` references the entire match, so the individual groups are 1-indexed.

3.4.10.2 Named Groups

If we have very complex expressions, we may want to refer to groups' content by name instead of by position. We can name a group by adding `?'name'` directly after the opening bracket. We can then reference it using `\g{name}`. If we use "tag" to refer to our fist group, our above expression could thus be `/<(?'tag'[^>]*).*>.*<\/\g{tag}>/`).

> Even if one uses a named group, it can still (also) be referenced positionally (by index).

3.4.10.3 Non-Capturing Groups

In some cases we're not interested in re-using a groups' content, but simply require a group to apply quantifiers to it. To avoid these groups' content being captured, we can add `?:` directly after the opening bracket. We could thus write the above `/end\s(if|loop)/` as `/end\s(?:if|loop)/`

Time to exercise!

3.4.11 Anchors and Multi-Line Matching

The characters `^` and `$` are used to *anchor* a pattern to the start (`^`) or the end (`$`) of the input. Thus, `$myself^` will only match if there's nothing else in the text it is applied to.

This very often isn't a very useful behavior in file based applications, so there is the possibility to add a flag `m` at the end of a regular expression (`/.../m`). If this flag is added, the behavior of the two anchors is changed to match the start and end of a line. Thus, `/^if$/m` will only match a single word "if" that stands alone on a line of text (vertical whitespace will be considered a line separator)

Time to exercise!

3.4.12 Global Matching

Besides `m`, another flag is implicitly added in most (or at least many) text editors when searching: `g`. Adding `g` at the end of a regular expression will find *every* match instead of just the first one. `/mypattern/g` will match "mypattern" globally, whereas `/mypattern/` will only find its first occurrance.

3.4.13 Lookahead / Lookbehind

To only match the e-mail handle, but to ingore the "@", we need to be able to specify so-called *lookahead* pattern. We can write `/.*(?=@)/` to achieve this.

The below table shows the expressions that match (first row) or don't match (second row) a pattern (in this specific case) an `@` that occurs after (left column, placed after what is to be matched) or before (right column, placed before what is to be matched) the pattern we are looking for.

	Lookahead	Lookbehind
Positive	`(?=@)`	`(?<=@)`
Negative	`(?!=@)`	`(?<!=@)`

Time to exercise!

3.4.14 Regex Delimiters

What if we need to match the literal character `/`? We can of course escape it, but if we need it multiple times within an expression (e.g. when matching a URL), that would be quite cumbersome.

To account for that, it is also possible to use different delimiters around the regular expression, namely `#`, `%`, `+`, `~`.

3.4.15 Further Flags

There are a number of other flags that can be used in regular expressions (other than “g” and “m” we saw above):

Flag	Meaning
`i`	Case insensitive
`s`	Single line - `.` also matches newline
`u`	Unicode instead of ASCII
`U`	Make all quantifiers lazy (not greedy)

3.5 Replacing

So far we've only looked at how to find patterns - and that will be very useful by itself. Even more useful, though will be replacing regular expressions by other text.

To denote a replacement, there is a further expression added after the match end delimiter, followed again by a delimiter. Any flags (such as for global or multiline matching) will be moved to after this final delimiter. A replacement could thus be `/foo/bar/g`, which would globally replace `foo` by `bar`.

3.5.1 Referencing Groups

The syntax used to reference groups in replacement text is similar to backreferences, but not quite the same. Instead of prefixing the position number by a backslash (e.g `\1`), the dollar sign (`$`) is prepended, making the reference to the first capturing group `$1` (`${1}` is also valid). This can e.g. be used to replace any occurrance of an abbreviation: `/(\d*)(?: )?kilo(\w*)/$1'000 $2/g`.

> If there are too little capturing groups for a double-digit backreference to be valid, then the content of the single-digit backreference will be used, followed by the second digit. In `/(\d*)da/$10/`, where there are only two groups, the first group will be referenced and postfixed by a "0". To force the use of a single-digit capturing group, surround the reference by curly braces.

Similarly, named groups can be referneced by `${name}`. In the above example, we could thus also write `/(?'quantity'\d*)(?: )?kilo(?'baseunit'\w*)/${quantity}'000 ${baseunit}/g`

Time to exercise!

3.5.2 Conditional Replacement

> This section is here for completeness. Most implementations do not support this. The good news: no exercise! ;-)

It is possible to output different replacements based on whether or not a capturing group was matched or not (groups with quantifiers that allow zero occurrances). This is written as `${1:+outputIfMatched:outputIfNotMatched}` (or `${name:+outputIfMatched:outputIfNotMatched}` for a capturing group with name "name"). As an example, we could use `/(minus )?(\d*)/${1+-:+}$2/` to prefix numbers with the "+" or "-" symbol.

3.5.3 Case Conversion

To convert text to a different case, we can use so-called *case conversion escape sequences*.

Escape	Conversion	Scope
`\U`	to uppercase	everything following
`\u`	to uppercase	next letter
`\L`	to lowercase	everything following
`\l`	to lowercase	next letter
`\E`	none	everything following

We can thus convert any text to upper case by using `/(.*)/\U$1/`

Any such escape sequence will cancel the effect of any previous one - that's what `\E` is used for.

Time to exercise!

4. JavaScript

4.1 Introduction

JavaScript (JS) is one of today's most commonly used programming languages. This is certainly due to many newer applications being delivered as web apps, but also because backend programs move to lean, JS based servers such as NodeJs [1] .

JS is an interpreted language. That means (very simply put) that while a program executes, every line of code is "read" by the *interpreter*, and then executed. For JS, the most common interpreters are browsers.

There are different versions of the JS language, resulting in different features being supported. To allow developers to write code using the newest language features, so-called *transpilers* translate code written in (typically) newer versions to the ones that will run in a target environment. Thus, if one wishes to support certain browsers (such as *IE*), one has to transpile to rather old versions of the language.

We will write code making use of *ES2020* [2] , but will introduce differences to older versions where that helps understanding legacy code.

4.1.1 Code Repository

The code snippets for this chapter can be found here [3] . Please ask to have your user added (if you haven't already)!

[1] `nodejs.org`
[2] `https://www.ecma-international.org/ecma-262/11.0/index.html`
[3] `https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/JavaScript`

4.2 Structure

All our language chapters will be structured roughly the same:

1. In the first section, we will introduce some tooling you can use for the exercises (or in real life). If you're used to another toolchain, please feel free to use that.
2. The main section is mapping the pseudocode syntax we looked at to the language taught in the chapter. This will give you the basic understanding of the language and allow you to easily dig deeper.
3. As a next step, we will look at some useful language features. These are things that don't exist in all (or even most) programming languages, but are at the core of the language.
4. In the fourth section, we will look at some important algorithms and their implementation. These are the "recipes" used to get specific routine tasks done.

4.3 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Tooling	20	60	***60***
Syntax Mapping	40	165	***120***
Language Features	20	50	***60***
Algorithms	40	185	***120***
Total	**120**	**360**	**480**

4.4 Tooling

4.4.1 Integrated Development Environment (IDE)

We will use a free IDE for most of our languages: Visual Studio Code [4] (VSCode). If you haven't already, please download and install the application in your working environment.

As a next step, please install a first extension: Code Runner [5] . This will enable you to run code snippets without creating the full infrastructure normally necessary to run a program. This will allow us to hit the ground running. Code runner requires node.js to be installed to execute JS code. Please follow the installation instructions on the download page [6] to set it up - we need at least version 14.0.0.

In order to execute code snippets, create a new file with extension `.js` and add your instructions in there. If you select the code you've written, you can right-click on the selection and choose "Run code". You will see any output produced directly in VSCode.

Time to exercise!

4.4.2 Browser

The nice thing about JS is that it runs in a browser (assuming you haven't disabled it). If you have e.g. Chrome installed, you can open *Developer Tools*, go to "Console" and enter and run code line by line. It's not as comfortable as using an IDE, but for just short snippets, it can serve its purpose as well.

Time to exercise!

[4] https://code.visualstudio.com/
[5] https://marketplace.visualstudio.com/items?itemName=formulahendry.code-runner
[6] https://nodejs.org/en/download/current/

4.4.3 Debugging

Feel free to skip this section - you can still come back to it later if you'd like. It's not strictly necessary for the course - but it can really help you understand the code you're executing later.

If your code doesn't do what you'd like, you might write down it's execution in a table tracking control flow step by step (as we did in the "Formal Languages" chapter). But there are two issues with that:

- If the code doesn't do what you think it should, you might not write down the correct steps
- It's a nightmare doing this for longer programs

Wouldn't it be nice if we could get our computer to run our code step by step and to show us the values of our variables at every step? Fortunately, this is exactly what a *debugger* does.

4.4.3.1 Setup

To set up debugging for JS in VSCode, we need to do two things:

1. Modify the "executorMap" for the *Code Runner* extension
2. Set up a debug configuration

If it doesn't already exist (it might if you're reusing an existing workspace), create the file `.vscode/settings.json` in the workspace root and paste the following snippet:

JSON

```json
{
  "code-runner.executorMap": {
    "javascript": "node --inspect-brk=localhost:9229"
  }
}
```

This will tell *Code Runner* to launch *node.js* in debug mode and suspend execution immediately. If you don't want to debug, you can simply remove `--inspect-brk=localhost:9229` and code will execute immediately.

Next, create (or extend) the file `.vscode/launch.json` with the following content:

JSON

```json
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "JavaScript: Attach to node",
      "address": "localhost",
      "port": 9229,
      "request": "attach",
      "skipFiles": [],
      "type": "pwa-node"
    }
  ]
}
```

4.4.3.2 Stepping Through Code

Now you're ready to step through some code. To do so, select the lines of code to execute, right click and select "Run Code" (if you do this often, you may want to remember - or customize [7] - the keyboard shortcut). VSCode's *output* panel will open and it will tell you that the debugger is listening on a specific URL. To start stepping, you have to go to the *Run* view (e.g. "View > Run") and press the ▶(play) button.

In the *Variables* panel, you will now see the values of all variables that are part of your program's context. Many of them you would not expect - just ignore these. You can now start to step through the code using "Run > Step Over" (or "Run > Step Into" once you are calling methods) to execute your code line by line. Once you're at the end of your code, use "Run > Continue" or "Run > Stop Debugging" to finish up.

While you're at it, have a quick look at the "Call Stack": this shows the methods that were called "on the way" to the one that is currently suspended. Surprising, how much there is going on, isn't it?

Time to exercise!

[7] https://code.visualstudio.com/docs/getstarted/tips-and-tricks#_customize-your-keyboard-shortcuts

4.5 Syntax Mapping

In this section, we will map our pseudocode syntax to the syntax used in JS in order to give a quick start into reading the language. We will of course "get our hands dirty" with some small exercises.

4.5.1 Comments

In JavaScript, there are two different types of comments:

- `//` will make the rest of a line a comment
- `/*` denotes the start of a comment, whereas `*/` marks its end. Such comments can span multiple lines.

```
console.log("No /*comments*/ in strings" /*+ " but outside"*/); // an example
```

4.5.2 Input and Output

4.5.2.1 Printing

To output information to the user, we'll use `console.log(string)` method. The mapping to our pseudocode is as follows:

JavaScript

```
console.log("Hello, JavaScript!");
```

Pseudocode

```
print("Hello, Pseudocode")
```

4.5.2.2 Reading

In theory, there's `readline()` to read from the console, but it's not supported in many environments. In reality, when in the browser, user input will be received from a graphical UI; running on a backend, input will typically be received via API requests. We will thus ignore direct user input here and focus on arguments passed to function parameters.

4.5.3 Data Types

JavaScript knows 8 basic data types:

Type	Example	Description
`undefined`	`undefined`	Value for variables that have no (other) assigned value (yet)
`boolean`	`true`	Logical value, either `true` or `false`
`string`	`"foo"`	Ordered sequence of characters
`number`	`42`	Floating point numbers with three special elements: `NaN`, `Infinity`, `-Infinity`
`bigint`	`12345678901234567890n`	Arbitrary magnitude integer (whole number) value
`object`	`{ a: true }`	A collection of properties with key and value. Details in a later sectio
`null`	`null`	Value for object type variables with no other value
`symbol`	`Symbol("foo")`	A guaranteed unique and immutable non-string value, used as object property key

> If you're *really* interested, head over to ECMAScript Data Types and Values [a] to have a look at the official specification of ES2020 for (much) more information.

We can evaluate the type of any expression (or constant / variable) by using the `typeof` built-in command: `typeof Symbol("foo")` will return `symbol`.

If you're looking for more information about JS, the official specification is probably way too complex. Let me introduce a far more easily accessible resource: The MDN JavaScript Guide [8] .

[a] https://www.ecma-international.org/ecma-262/#sec-ecmascript-data-types-and-values
[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

4.5.3.1 Operators

The operators we looked at for our pseudocode are all applicable to JS. There is one important caveat and one according addition, though. `==` doesn't care about its' operators types very much. The operator looks for equivalence, not equality. If one operand be converted to the other, `==` will be `true`:

- `1 == "1"`
- `0 == false` (both are falsy)

If we want to know whether two values are really equal, we have to use `===`.

The fact that operators interpret `truthy` and `falsy` values can also be used to only evaluate an expression when another one is either `truthy` or `falsy`

JavaScript

```javascript
1 const foo = undefined;
2 const x = foo || 1;
3 const y = x && 2;
```

- On line 2, the value assigned to x will be `1` because `foo` is `undefined`.
- On line 3, `y` will be assigned `2` because `x` is now `1` (which is `truthy`).

This behaviour is extremely useful, but may lead to new problems: if we're looking to assign a default for an undefined number, we have the problem that `0` is `falsy` and will thus be replaced by the default. To cater for this, a new operator was introduced in *ES2020*: the "nullish coalescing operator", `??`. It basically behaves like `||`, but only evaluates to the second argument if the first operator is either `undefined` or `null`.

JavaScript

```javascript
const foo = undefined;
const x = foo ?? 0; // x is assigned 0
const y = x   ?? 1; // y is assigned 0
const z = x   || 1; // z is assigned 1
```

Time to exercise!

4.5.4 Data Structures

4.5.4.1 Constants and Variables

Originally, JS only knew one keyword used to declare data structures: `var`. Since *ES6*, immutable data is declared using `const`, whereas mutable data is declared using `let`.

JavaScript

```
let x = 9;
let y = 5;
const z = 2;
console.log(x); // 9
```

Pseudocode

```
variable x = 9
variable y = 5
constant z = 2
print(x) // 9
```

4.5.4.2 Arrays

Arrays work exactly the same as in our pseudocode:

JavaScript

```
const input = [2, 4, 9, 1, 7];
const first = input[0]; // 2
const len   = input.length; // 5
input.push(18); // [2, 4, 9, 1, 7, 18]
// n/a
input.push(4, 8)
input.slice(2, 1)
// n/a
```

Pseudocode

```
input          = [2, 4, 9, 1, 7]
constant first = input[0] // 2
constant len   = input.length // 5
input[5]       = 18 // [2, 4, 9, 1, 7, 18
// n/a
// n/a
// n/a
// n/a
```

> Even though the array is declared to be constant, that doesn't mean no elements can be added or removed. It only means that the constant itself cannot be reassigned!

4.5.4.3 Objects

JS has an object type which allows modelling objects with their properties. If we add quotes around the property names, we have the same object described in **J**ava**S**cript **O**bject **N**otation, JSON. As an example, the following could be a very simple object representing a person:

JavaScript

```javascript
const alice = {
  name: "Alice",
  spouse: { name: "Bob" },
  children: [
    { name: "Eve" },
    { name: "Trudy" },
  ],
}
```

JSON

```json
{
  "name": "Alice",
  "spouse": { "name": "Bob" },
  "children": [
    { "name": "Eve" },
    { "name": "Trudy" },
  ],
}
```

We can get an array of an object's keys by calling `Object.keys(foo)` on an object called `foo`.

Note that object keys can be of type `string`, `number` (, or `symbol` - but that's irrelevant here).

Time to exercise!

4.5.4.4 Maps

Originally, there wasn't a separate concept of maps in JS, but rather JS *objects* offered much of their functionality. Today, most modern JS interpreters know `Map`.

JavaScript

```javascript
const corners = new Map();
corners.set("Triangle", 3);
corners.set("Rectangle", 4);
corners.set("Pentagon", 5);

console.log(corners.get("Triangle")); //
    3
```

Pseudocode

```
constant corners = {
  "Triangle":  3,
  "Rectangle": 4,
  "Pentagon":  5,
}
print(corners["Triangle"]) // 3
```

Time to exercise!

4.5.5 Control Structures

4.5.5.1 Conditions

JS expresses conditions as follows:

JavaScript

```javascript
if (x < 5) {

  console.log("smaller");
} else {
  console.log("greater or equal");
}
```

Pseudocode

```
if x < 5 then

  print("smaller")
else
  print("greater or equal")
end if
```

Be very careful: conditions take into account so-called `truthy` (everything for which `ToBoolean()` evaluates to `true`) and `falsy` (everything for which `ToBoolean()` evaluates to `false`) values, not just `true` and `false`. Here's a complete list of falsy values (everything else is `truthy`):

- `false`
- `undefined` / `null`
- `""`
- `+0` / `-0` / `0n`
- `NaN`

Time to exercise!

4.5.5.1.1 Conditionals

If we only have conditions, assigning different values to a variable is quite verbose (and assigning to a constant is impossible):

JavaScript

```javascript
let foo;
if (condition) {
  foo = "Great!";
} else {
  foo = "Oh no ;-(";
}
```

Fortunately, there is a simpler way of doing this!

JavaScript

```javascript
let foo = condition ? "Great!" : "Oh no ;-(";
```

4.5.5.2 Case Matching

JavaScript

```
switch(optionType) {
  case "plant":
    console.log("water it")
    break;
  case "animal":
    console.log("feed it")
    break;
  case "gas":
    console.log("inhale it")
    break;
  case "liquid":
    console.log("surf it")
    break;
  case "solid":
    console.log("decorate it")
    break;
  default:
    console.log("stare at it")
}
```

Pseudocode

```
if objectType
  matches plant  then
    print("water it")

  matches animal then
    print("feed it")

  matches gas    then
    print("inhale it")

  matches liquid then
    print("surf it")

  matches solid  then
    print("decorate it")

else
    print("stare at it")
endif
```

Time to exercise!

4.5.5.3 Loops

In JS, a loop can take one of three basic forms:

- A “while” loop:

JavaScript

```
var i = 1;
while (i < 5) {

  console.log(i++);
}
```

Pseudocode

```
variable i = 1;
loop
  exit if i >= 5;
  print(i++);
end loop
```

Output

```
1
2
3
4
```

- A “do while” loops:

JavaScript

```
var i = 1;
do {
  console.log(i++);

} while (i < 5);
```

Pseudocode

```
variable i = 1;
loop
  print(i++);
  exit if i >= 4;
end loop
```

Output

```
1
2
3
4
5
```

- A “for”-loop:

JavaScript

```
for (var i = 1; i <= 4; i++) {


  console.log(i);
}
```

JavaScript

```
variable i = 1;
loop
  exit if i > 4;
  print(i++);
end loop
```

Output

```
1
2
3
4
```

So what’s the difference between these three ways of expressing loops?

- A `for` loop is really just a shorthand for a `while` loop, allowing you to express *loop variable* declaration (`var i`), initialization (`i = 0`), entry condition (`i < 5`) and loop variable modification (`i++`) in one line.
- Have you noticed the difference in outputs between the “while” and the “do while” loops? The “do while” loop always executes the contained code at least once and then evaluates what we could call a “re-entry condition”. The “while” loop on the other hand evaluates an “entry condition”, which means it may never execute the contained code at all!

There are two more types of loops:

- `for (let item of arr) { ... }` loops over all elements in `arr`, assigning the current one to `item`.
- `for (let key in obj){ ... }` loops over all property keys of an object. This is equvalent to `for(let item of Object.keys(obj)) { ... }`

Time to exercise!

4.5.6 Program Structures

4.5.6.1 Methods

In our pseudocode, we declared methods like variables or constants. In fact, the same is possible in JS (our syntax draws largely from JS) - they call it *arrow function*.

JavaScript

```
const calculateFactorial = (input) => {
  let result = 1;


  for(let cnt = 1; cnt != input; cnt++) {
    result *= cnt;

  }

  return result;
}
```

Pseudocode

```
method calculateFactorial = (input) => {
  variable result = 1
  variable cnt = 1

  loop
    result = result * cnt
    exit if cnt++ == input
  end loop

  return result
}
```

In JS, we can also declare this the "traditional" way:

JavaScript

```
function calculateFactorialTraditional(input) {
  let result = 1;
  for(let cnt = 1; cnt != input; cnt++) {
    result *= cnt;
  }
  return result;
}
```

This is called a "named function" as opposed to an "anonymous function". As an example, arrow functions don't have their own names: they are anonymous. They can be assigned to constants, which do have a name, but that's not necessarily the case.

> Modern browsers are able to show you the name of the variable a function is assigned to in a stack trace even if it's an anonymous function. In some environments, that may not work and you may just see e.g. `<anonymous>`.

Time to exercise!

4.5.6.2 Modules

Programs tend to become rather unreadable if all their code is stored in one file. JS adresses this via *modules*. You can split your code up and distribute it over different files, which will each define a module. To glue them together, we use *exports* and *imports*.

Let's say we have two files, one a library which offers different utilities (`util.js`) and one with code (`foo.js`) that uses one of these utility functions.

There are two main ways how libraries can organize their exports and three ways (depending on the library's organization) to import these into our own programs.

4.5.6.2.1 Default Exports

The original - and thus probably most used - way to export functions is by using `export default` on an object containing the functions offered for re-use. Note that the object contains one property with key `foo` and value of type function (`() => { console.log("foo called"); }`); thus there's a colon (`:`) and not an equals sign (`=`) as you might have expected between key and value.

JavaScript

```javascript
// util.js
export default {
  foo: () => {
    console.log("foo called");
  }
}
```

In order to use `foo`, we can simply choose a name for the entire object exported as `default` by the library and call `foo` from that object as if we had declared it locally:

JavaScript

```javascript
// bar.js
import Utils from "./util";

Utils.foo();
```

For this to work in our code runner extension, we need to enable module syntax. To do this, we need to install the `esm` package and configure code runner to use it:

1. in a terminal at the folder where you've created your files run `npm i esm` to install the package using the **n**ode **p**ackage **m**anager.
2. Reopen `.vscode/settings.json` and add `-r esm $fullFileName` directly after `node`.

4.5.6.2.2 Named Exports

The more recent way of exporting functionality is by exporting individual constants and functions, for example:

JavaScript

```javascript
// util.js
export const QUESTION = undefined;
export const ANSWER = 42;
export const foo = (answer) => {
  console.log(`The answer is ${answer}`);
}
```

In order to import these, we have choose one of two ways. We can import selected exports:

JavaScript

```javascript
// bar.js
import { ANSWER, foo } from "./util";

foo(ANSWER);
```

Alternatively, we can import all the exports:

JavaScript

```javascript
// baz.js
import * as Utils from "./util";

Utils.foo("blowing in the wind");
```

4.6 Language Features

4.6.1 Template Strings

JS allows to expand constants / variables inside a special string type called *template string*. You may hear the term *string interpolation* to describe this behavior. The way it works is as follows:

JavaScript

```javascript
function foo(input) {
  console.log(‘I’ve been called with argument ${input}!‘);
}
```

The template string is surrounded by backticks (`) and any variable to be expanded is surrounded by `${}`. Another cool feature of template strings is that they can contain line breaks, so the following is valid code:

JavaScript

```javascript
function foo(input) {
  console.log(‘Knock, knock!
Who’s there?
Alex.
Alex who?
Alex-plain when you open the door!‘);
}
```

4.6.2 Spread Operator

The *spread operator* (`...`) allows us to say “each key/value pair” in an object or array (where keys are the indexes). We can use it for various things:

- replacing one property in an object:

  JavaScript

  ```javascript
  const single = { forename: "Meghan", surname: "Markle" };
  const married = { ...single, surname: "Mountbatton-Windsor" };
  ```

  We use the spread operator to refer to all properties of an object. Because a second mention of the same key overwrites the value of the first mention, the property “surname” will be overwritten.
- Transforming all items in an array. Remember how we manually declared an array with numbers from 0 to 5 above? We can now do this more elegantly:

  JavaScript

  ```javascript
  const firstSix = [...Array(6).keys()];
  ```

Time to exercise!

4.6.3 Variable as Property Key

Let's revisit Alice.

JavaScript

```javascript
const alice = {
  name: "Alice",
  spouse: { name: "Bob" },
  children: [
    { name: "Eve" },
    { name: "Trudy" },
  ],
};
```

If we want to know Alice's spouce's name, we can access it by using `alice.spouse.name`. Because objects are also maps, we can also use `alice["spouse"]["name"]` or even mix it up: `alice["spouse"].name`. What about if the object can have the keys in different langauges?

JavaScript

```javascript
function spouseName(person, nameKey) {
  return person.spouse[nameKey];
}

console.log(spouseName({ spouse: { nombre: "juan" } }, "nombre"));
```

> Note that in reality we would have to deal with cases where the object passed into the function is not a well defined person or doesn't have a spouse or has an anonymous spouse or . . .

In the same way, we can now build an object using dynamic keys:

JavaScript

```javascript
function addSpouse(person, nameKey, spouseName) {
  return { ...person, spouse: { [nameKey]: spouseName} };
}

console.log(addSpouse({ }, "nombre", "juan"));
```

Time to exercise!

4.7 Algorithms

4.7.1 Searching

4.7.1.1 Arrays

JS makes it very easy to search in an array by offering the `find` function on arrays. The way it works is the following: you call `find` on an array and pass it a function which evaluates as `truthy` or `falsy` (also called a *predicate*) as an argument, e.g.

JavaScript

```javascript
[5, 6, 8, 12, 23].find((e) => e > 9); // will return [12]
```

Note that `find` only returns the first match if called with a simple predicate!

4.7.1.2 Strings

To search in a string, we can use a variety of methods. Here are some examples:

Function	Description	Example
`indexOf`	finds the first match of the passed string	`"thanks".indexOf("hank")` $\rightarrow$ 1
`search`	finds the first match of the passed regex	`"foo(bar)".search(/\(.*\)/)` $\rightarrow$ 3

Time to exercise!

4.7.2 Sorting

Arrays can be sorted very easily by passing a *comparator* to `sort` called on the array. Here's how:

JavaScript

```javascript
// default sort: will return ["an", "apple", "be", "must", "red"]
["an", "apple", "must", "be", "red"].sort();
// will return ["red", "must", "be", "apple", "an"]
["an", "apple", "must", "be", "red"].sort((a, b) => a === b ? 0 : (a < b ? 1 : -1));
// will return ["an", "be", "red", "must", "apple"]
["an", "apple", "must", "be", "red"].sort((a, b) => a.length - b.length);
```

Time to exercise!

4.7.3 Mapping

One cool thing in JS is how simple it is to map each element of an array to something else. To do that, one can simply call the `map` function on the array and pass a function that transforms each individual element.

JavaScript

```javascript
[0, 1, 2, 3, 4, 5].map((e) => e * e); // will return [0, 1, 4, 9, 16, 25]
```

Time to exercise!

4.7.4 Aggregating

Often times we want to aggregate individual values, e.g. sum up numbers, concatenate strings, or create an object from the contents of an array. For this purpose, there's the `reduce` function. We have to pass it a function which aggregates the previous result with the current element and a starting value.

JavaScript

```javascript
[3, 5, 28, 7, 96].reduce((result, current) => result * current, 1); // => 282240
["an", "apple", "must", "be", "red"].reduce((result, current) => result + " " + current
    , "").trim(); // => "an apple must be red"
```

Time to exercise!

5. Shell Scripts

5.1 Introduction

A shell is a command processor: you type, it executes. Shell scripts make life bearable. Without them, we would be doomed to repeat ourselves over and over again.

We are generally going to use the `bash` (Bourne Again Shell [1]) and are going to aim for syntax that works on bash 3 - that version is still around in lots of places because it was the last version released under GPL v2 [2] , later versions were distributed under less permissive licenses. Wherever we use features not available in that version, we will be explicit about it.

We will separately look at Unix tools that help us do things more easily, but we will in some cases use these in the exercises, so feel free to jump around between the two chapters. We won't have exercises for everything, but please try the examples given and play around a bit to try and get a hang of everything!

5.1.1 Code Repository

The code snippets for this chapter can be found here [3] . If you can't access it, please ask!

5.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Syntax Mapping	30	85	***115***
Language Fetures	40	115	***155***
Total	**70**	**200**	**270**

[1] https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29
[2] https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html
[3] https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/ShellScripts

5.3 Syntax Mapping

In this section, we will map our pseudocode syntax to the syntax used in Unix shells in order to give a quick start into reading the language. We will of course "get our hands dirty" with some small exercises.

5.3.1 Input and Output

5.3.1.1 Printing

To output information to the user, we'll use `echo` instruction. The mapping to our pseudocode is as follows:

Bash

```bash
echo "Hello, Shell"
```

Pseudocode

```
print("Hello, Pseudocode")
```

There are two relevant options to the `echo` command:

Options	Impact
-e	Interpret backslash-escaped space characters: (e.g. `\n`, `\t`)
-n	Don't print a newline at the end

5.3.1.2 Reading

The most basic form of reading a user input into a variable is as follows:

Bash

```bash

read -p "Enter your choice: " x
```

Pseudocode

```
print("Enter your choice: ")
constant x = read()
```

The `read` command accepts some parameters:

Options	Impact
-s	Don't show characters typed (e.g. for passwords)
-p $prompt	Output `$prompt` without following newline before reading
-t $seconds	Time out after `$seconds`
-d $eoi	Use `$eoi` as end of input delimiter instead of the return key

5.3.2 Comments

In bash, there are only really single line comments: `#` marks the rest of the line as a comment. They have to either stand at the beginning of the file or be preceded by a whitespace character.

Bash

```bash
# Line comment

```

> Hashes without a preceding space can mean other things as you'll see below.

5.3.3 Data Structures

5.3.3.1 Constants and Variables

All variables are stored as strings in the background. Variables generally don't need to be declared, they can just be used. Note that in order to access a variable (as we do when printing it on the console), we need to prefix it with `$`.

Constants have to be declared a bit more explicitly by prepending `declare -r` to the assignment.

Bash

```bash
x=9
y=5
declare -r z=2
echo $x # 9
```

Pseudocode

```
variable x = 9
variable y = 5
constant z = 2
print(x) // 9
```

> Note that there cannot be any spaces around the equal (`=`) sign! This is due to the fact that spaces separate commands.

Time to exercise!

5.3.3.2 Arrays

Bash

```
input=(2 4 9 1 7)
first=${input[0]} # 2
len=${#input[@]} # 5
input[5]=18 # (2 4 9 1 7 18)
input+=(9) # adds an element at the end
input+=(4 8)
input=("input[@]:0:2””{input[@]:3}")
# n/a
```

Pseudocode

```
input          = [2, 4, 9, 1, 7]
constant first = input[0] // 2
constant len   = input.length // 5
input[5]       = 18 // [2, 4, 9, 1, 7, 18
// n/a
// n/a
// n/a
// n/a
```

An array can also be declared explicitly (see below) by prepending `declare -a`.

There are three ways to access the array as a “structure”:

- `echo ${foo[@]}` prints all array elements
- `echo ${!foo[@]}` prints all array indexes
- `echo ${#foo[@]}` prints the number of elements in the array

The curly braces are required for array access!

5.3.3.3 Maps

Real maps have only been around since bash version 4. This means that you may not be able to use this in some environments - nevertheless, it’s good to know the syntax!

Bash

```
declare -A corners=(
  [Triangle]=3
  [Rectangle]=4
  [Pentagon]=5
)
echo ${corners[Triangle]} # => 3
```

Pseudocode

```
constant corners = {
  "Triangle":  3,
  "Rectangle": 4,
  "Pentagon":  5,
}
print(corners["Triangle"]) // 3
```

For maps, the following provide access to the structure as a whole:

- `echo ${shapeCorners[@]}` prints all map values
- `echo ${!shapeCorners[@]}` prints all map keys
- `echo ${#shapeCorners[@]}` prints the number of items in the map

- Maps cannot be declared implicitly (without `declare -A`).
- Maps are not ordered!

5.3.4 Control Structures

5.3.4.1 Conditions

In shell scripts, conditions are expressed as follows:

Bash

```bash
if [ $x -lt 5 ]
then
  echo "smaller"
else
  echo "greater or equal"
fi
```

Pseudocode

```
if x < 5 then

  print("smaller")
else
  print("greater or equal")
end if
```

This is a place where whitespace matters! You need

- Spaces around the opening bracket (`[`)
- A space preceding the closing bracket (`]`)
- Line breaks before `then`, `else`, and `fi`
  - alternatively, you can replace these by semicolons, the following works as well:
    `if [ $x -lt 5 ]; then echo "smaller"; else echo "greater or equal"; fi`

> Note that `[` is actually a program! You can check for yourself by running `which [`!

There are different operators used to compare strings and numbers:

Operator (Number)	Operator (String)	Meaning	Operator (Number)	Operator (String)	Meaning
`-eq`	`=`	**E**qual	`-ne`	`!=`	**N**ot **e**qual
`-lt`	`<`	**L**ower **t**han	`-gt`	`>`	**G**reater **t**han
`-le`	n/a	**L**ower or **e**qual	`-ge`	n/a	**G**reater or **e**qual

There are also specific operators that are useful when dealing with input and files, e.g.:

Operator	Meaning	Operator	Meaning
`-z $string`	True if `$string` is empty	`-n $string`	True if `$string` is not empty
`-f $path`	True if `$path` points to a file	`-d $path`	True if `$path` points to a directory

Time to exercise!

5.3.4.2 Case Matching

Here’s how we match different cases in bash:

Bash

```
case $type in
  plant)    echo "water"    ;;


  animal)   echo "feed"     ;;


  gas)      echo "inhale"   ;;


  liquid)   echo "surf"     ;;


  solid)    echo "decorate" ;;


  *)        echo "stare"    ;;

esac
```

Pseudocode

```
n/a
```

> Like with `if` and `fi`, `esac` is an anadrome of `case` (the word written in reverse letter order).

5.3.4.3 Loops

In bash, a loop can take one of two basic forms:

- A “while” loop:

Bash

```
i=1
while [ $i -lt 5 ]
do
  echo $((i++))
done
```

Pseudocode

```
variable i = 1;
loop
  exit if i >= 5;
  print(i++);
end loop
```

Output

```
1
2
3
4
```

> `$((expr))` evaluates an arithmetic expression `expr`

- A “for”-loop:

Bash

```
for i in {1..4}

do

  echo "$i"
done
```

Pseudocode

```
variable i = 1;
loop
  exit if i > 4;
  print(i++);
end loop
```

Output

```
1
2
3
4
```

For loops simply loop through all the elements provided to them.

To loop over an array’s contents, we need a somewhat strange syntax:

Bash

```
input=(1 2 3 5 8)
for t in ${input[@]}; do
  echo $t
done
```

We can also loop forever and use `break` to exit the loop, just as we would using `exit` in our pseudocode:

Bash

```
i=3;
while true
do
  echo $((i++));
  if [ $i -ge 4 ]; then
    break;
  fi
done;
```

Pseudocode

```
variable i = 3;
loop

  print(i++);

  exit if i >= 4;

end loop
```

Output

```
3
```

Often times, you will see a colon (`:`) instead of `true`, such as in `while : ; do ...; done`

Time to exercise!

5.3.5 Program Structures

5.3.5.1 Methods

Methods are defined without a signature in bash - that's why it's good practice to have reusable methods output their parameters if called with e.g. `--help` or `-h` (such as most programs do that are available in Unix systems).

Bash

```bash
 1 power_usage() {
 2   echo "Usage: power BASE EXPONENT"
 3 }
 4 
 5 power() {
 6   base=$1 # first argument passed
 7   exponent=$2 # second argument passed
 8 
 9   if [ $base = "-h" ] || [ $base = "--help" ]; then power_usage && exit 1; fi
10   if [ -z $base     ] || [ -z $exponent     ]; then power_usage && exit 1; fi
11 
12   echo "$base ^ $exponent = $(($base**$exponent))"
13   return 0
14 }
15 
16 binary_usage() {
17   echo "Usage: binary LIMIT"
18 }
19 
20 binary() {
21   if [ -z $1 ];                          then binary_usage && exit 1; fi
22   if [ $1 = "-h" ] || [ $1 = "--help" ]; then binary_usage && exit 1; fi
23 
24   limit=$1
25   i=1
26   while [ $i -le $limit ]; do
27     power 2 $((i++)) # call "power" with arguments 2 and "i" (then increase i)
28   done
29 
30   return 0
31 }
```

- In most programming languages (and in our pseudocode syntax), `return` is used to transport information to the caller. In shell scripts, what is returned is the exit status of a method.
  - Exit statuses must be between 0 and 255, where only 0 means "success".
- Methods are called the same way programs / commands are called in the shell - note the call to `power` on line 27 in the above code.
- If you define the above program in its own file,
  - a new shell (a *subshell*) process is spanwned (*forked*) when it's executed. It inherits the environment of its parent shell.
  - you'll have to add a separate call to e.g. `binary` and pass the arguments received.

Time to exercise!

5.3.5.1.1 Variable Scope

Variables declared *explicitly* (using the keyword `declare`) are visible only within the method they're declared in and in any methods called from these (directly or indirectly). Using `declare -g` (or no `declare` at all, creating the variable *implicitly*) makes them visible globally. Consider the following example:

Bash

```bash
x1=1
declare x2=2
declare -g x3=3

f() {
    fx1=1
    declare fx2=2
    declare -g fx3=3

    g

    echo "f - x1: $x1"   # f - x1: 1
    echo "f - x2: $x2"   # f - x2: 2
    echo "f - x3: $x3"   # f - x3: 3
    echo "f - fx1: $fx1" # f - fx1: 1
    echo "f - fx2: $fx2" # f - fx2: 2
    echo "f - fx3: $fx3" # f - fx3: 3
    echo "f - gx1: $gx1" # f - gx1: 1
    echo "f - gx2: $gx2" # f - gx2:
    echo "f - gx3: $gx3" # f - gx3: 3
}

g() {
    gx1=1
    declare gx2=2
    declare -g gx3=3

    echo "g - x1: $x1"   # g - x1: 1
    echo "g - x2: $x2"   # g - x2: 2
    echo "g - x3: $x3"   # g - x3: 3
    echo "g - fx1: $fx1" # g - fx1: 1
    echo "g - fx2: $fx2" # g - fx2: 2
    echo "g - fx3: $fx3" # g - fx3: 3
    echo "g - gx1: $gx1" # g - gx1: 1
    echo "g - gx2: $gx2" # g - gx2: 2
    echo "g - gx3: $gx3" # g - gx3: 3
}

f

echo "x1: $x1"   # x1: 1
echo "x2: $x2"   # x2: 2
echo "x3: $x3"   # x3: 3
echo "fx1: $fx1" # fx1: 1
echo "fx2: $fx2" # fx2:
echo "fx3: $fx3" # fx3: 3
echo "gx1: $gx1" # gx1: 1
echo "gx2: $gx2" # gx2:
echo "gx3: $gx3" # gx3: 3
```

5.4 Language Features

5.4.1 Builtin Parameters

There are quite a few builtin shell parameters (or just *parameters* for short) which give you information about your current working environment. Here are a few particularly relevant ones.

Parameter	Description
`$PWD`	Current working directoy
`$OLDPWD`	Previous working directoy
`$PATH`	Default search locations for binaries
`$UID`	Current user's ID
`$SHLVL`	"Depth" of the shell. Calling `bash` from within `bash` will increase this value
`$IFS`	Separator(s) used between "words". Default is `$' \t\n'`
`$0`	The name of the command / script called
`$1` .. `$n`	First to nth argument passed to a command / script
`$#`	Number of command line arguments a command / script was called with
`"$*"`	Single string of all parameters, joined by the first character in `$IFS`
`"$@"`	All parameters, each of them double quoted, joined by the first character in `$IFS`
`$?`	Exit status of the previously called command
`$$`	Current process' PID
`$!`	Last started backgrond job's PID

- You can treat `"$@"` or `"$*"` like an array in many respects. If you need an actual array consisting of the elements of `"$@"` or `"$*"`, you can assign `("$@")` or `("$*")` to a variable.
- Without quotes, `"$@"` and `"$*"` behave the same.

Time to Exercise!

5.4.2 Expansions

5.4.2.1 Strings

Single-quoted strings (e.g. `'Hi,\n$there'`) are treated differently from double quoted ones (e.g. `"Hi,\n$there"`), and signle-quoted strings prefixed with a dollar sign (e.g. `$'Hi,\n$there'`) are special again:

- In double-quoted strings, parameter expansion (see below) happens
- In dollar-prefixed single-quoted strings, backslash-escaped characters are interpreted as specified in the *ANSI C* [4] standard.
- Other (non-dollar-prefixed) single-quoted strings are left as is, they are used literally

Try the three examples above!

5.4.2.2 File Literals

File literals are called *Here Documents* or *HereDocs* for short. They are blocks of text that are treated like files. One application that is often seen is to write multi-line strings to a file:

Bash

```bash
1 cat <<EOF > ./script.sh
2   read -p "What's your name?" name
3   echo "Hello, \$name!"
4 EOF
```

Let's break this down

- `cat` prints a file's (in this case the HereDoc's) content to the console
- `<<EOF` states that a HereDoc will start on the next line. `EOF` could be anything (e.g. `MyFooBlubb123!`), as long as the terminator (line 3 above) is the same word
- `> ./script.sh` instructs writing the output generated by `cat` into `script.sh` in the current directory
- `EOF` on line 3 terminates the HereDoc

> - There must not be any space before the end marker (here: `EOF`)
> - To avoid string expansion, put the start marker (here: `<<EOF`) in single quotes (`<<'EOF'`)
> - To trim (ignore) leading spaces / tabs, use `<<-EOF`.
> - The dollar sign on line 3 has to be escaped to avoid variable expansion.

5.4.2.3 Output

Using `$($command)`, we can re-use the `$command`'s output instead of having it print to the console. A stupid example to demonstrate this is `echo $(pwd)` - run its components to understand what happens!

Time to exercise!

[4] https://www.gnu.org/software/bash/manual/html_node/ANSI_002dC-Quoting.html

5.4.2.4 Parameters

Shell parameters are variables valid for a given shell (a user session). They are simply declared by stating a name (e.g. `PAR`) and assigning them a value (e.g. `PAR=world`). They can be *expanded* prefixing them with the dollar sign (e.g. `$PAR`, which could be used in `echo "hello, $PAR"`). They can also appear surrounded by curly braces (e. `echo "${PAR}ly pleasures"`).

There are a few very cool things you can do using parameter expansion that typically require much more typing (and there are lots more - you now know to look for these if you need them):

- `${!PAR}`: expands `$PAR` and treats the expanded value as another variable to be expanded.
- `${!PREFIX*}`: returns all parameters starting with `$PREFIX`.
  - To expand them all, use e.g. `for i in ${!PREFIX*}; do echo "$i: ${!i}"; done`
- `${PAR1:-PAR2}`: expands `$PAR1`. If it is unset or empty, `$PAR2` will be expanded instead.
  - `${PAR1-PAR2}` (without the colon) will only expand `$PAR2` if `$PAR1` is unset, but not if it is empty.
- `${PAR1:=PAR2}`: expands `$PAR1`. If it is unset or empty, `$PAR2` will be expanded and assigned to `$PAR1`.
  - `${PAR1=PAR2}` (without the colon) will only expand and assign `$PAR2` if `$PAR1` is unset, but not if it is empty.
- `${PAR: $start: $num}` will expand `$PAR` and return its substring of length `$num` between position `$start` and `$start + $num`.
  - The space after the colons is optional for positive numbers or non-numbers, but necessary for negative numbers - otherwise it could be confused with `${PAR1:-PAR2}`.
  - `:$num` can be omitted to get the entire substring starting from `$start`.
  - Negative numbers for `$start` start counting from the end of the string.
  - Negative numbers for `$num` remove that amount of characters from the end of the string.
  - This works for arrays too, with `$num` denoting a number of items
- `${#PAR}` returns the number of characters in the expanded `${PAR}`.

Time to exercise!

5.4.2.5 Curly Braces

Using curly braces, we can express different options with less code. As an example, to create multiple directories, we can use `mkdir -p ~/examples/{loops,conditions,cases}`. It's also possible to use the form `{m..n}` to expand a sequence of numbers: `rm ./chapters/{2..9}.pdf`.

Even better, we can provide a step width as a third parameter: `{m..n..s}` will create a sequence from `m` to `n` with step width `s`. As an example, `{A..G..2}` results in the sequence `A C E G`.

Braces can be nested, so we can e.g. state `mkdir -p ./repos/{a,b/{src,test}}`

Time to exercise!

5.4.2.6 Filenames

5.4.2.6.1 Tilde

A tilde (`~`) and the characters folloing it up to the first unquoted slash (if any) has a special meaning if it's the first character of a file path.

`~`	The currently logged in user's home directory
`~-`	The previous working directory (equivalent to `$OLDPWD`)

5.4.2.6.2 Patterns

Remember regular expressions? A very similar (but also very limited) syntax can be used when matching file names.

Pattern	Effect
`*`	Matches any number (including zero) of characters
`?`	Matches a single character (but not zero characters)
`[]`	Matches one of the characters inside the brackets
`[^...]` or `[!...]`	Matches one character NOT inside the brackets

Using `[a-c]` may yield unexpected results as the evaluation depends on the defined locale's sorting. It could e.g. be the same as putting `[aBbCc]`.

Character classes in the format `[:...:]` can also be used inside brackets, the most relevant here again as a reminder: `alpha` / `alnum` / `digit` / `blank` / `lower` / `upper` / `space` / `word`

To match a single digit, we thus need to use `[[:digit:]]`.

5.4.3 Commands

5.4.3.1 Built-In

The call to a shell command has three parts:

- the command name (also simply “the command”)
- options, typically prefixed with `-` (for single letter options) or `--` (for spelled out options)
- arguments

We’ve already met the builtin commands `echo` and `read` - let’s look at a few more we’re likely to use.

- `alias $name='$command $args'`: defines an alias `$name`, which can be typed to execute the `$command`, passing `$args`. As an example, I always define `alias ll='ls -al'`.
- `cd $path`: changes the current working directory to the given `$path`
  - `cd -` is equivalent to `cd $OLDPWD`
- `eval`: concatenates all arguments to one expression and executes it as if it was entered directly into the shell
- `exit $status`: exits the current shell, returning status code `$status`
- `pwd`: prints the current working directory
- `pushd $dirname`: navigates to `$dirname` and keeps a record of the directory history (pushes the previous directory on a *stack*).
- `popd`: goes back to the previous working directory (the one before the last `pushd`, the last one pushed on the *stack*)
- `dirs`: lists the directory *stack*
- `source $file` (or simply `. $file`): executes the commands in the file `$file` specified
- `test $expression`: evaluates the logical expression `$expression`
- `type -t $name`: evaluates whether the given `$name` refers to…
  - a builtin command; e.g. `type -t type`
  - a file (a binary); e.g. `type -t bash`
  - an alias; e.g. `type -t ll` (if you’ve defined it as I do)
  - a function; e.g. `type -t pyenv` (if you have python installed: probably)
  - a keyword; e.g. `type -t for`
- `unset $PAR`: undefines `$PAR`

5.4.3.2 Chaining

We can run commands sequentially by adding `;` between them, e.g. `$command1 ; $command2`

We can make the execution of one command conditional on the exit status of a previous command.

- To only run `$command2` if `$command1` was successful (exit status 0), we can use `$command1 && $command2`.
- To only run `$command2` if `$command1` was NOT successful (non-zero exit status), we can use `$command1 || $command`

Time to exercise!

5.4.3.3 Background Execution

Adding a single ampersand (`&`) after a command executes it in the background (asynchronously in a subshell). The exit status of `$command1 &` is always 0, so `($command1 &) && $command2` will always execute `$command2` immediately after scheduling `$command1` to be executed in a subshell.

To get the actual exit code of a background process, one needs to take note of that process' PID (held in `$!`) and once we know it's exited (e.g. by looking at whether `ps` still lists it), calling `wait $pid` (where `$pid` is the PID of the background process). The exit status of `wait` will be the one of the background process we're interested in.

> Executing `wait` before the process has exited will block the shell until it exits.

5.4.3.4 History

To reuse a previously typed command, we can use different *event designators*. Here are a few relevant use cases:

Event	Description	Example
`!!`	Rerun the previous command	`sudo !!`: rerun the last command as su
`!-3`	Rerun the command three invocations back	
`!foo`	Rerun most recent command starting with `foo`	
`$command !*`	run `$command` with the previous arguments	

To search the history for a previous command:

1. Type `CTRL+R`
2. Start typing the command you're looking for
3. Use `CTRL+R` to look for earlier matches
4. Press `ENTER` (or `CTRL+O` to execute the command found) or `CTRL+G` (or `CTRL+C`) to cancel.

> You can also view your complete history by executing `history`.

5.4.4 Streams

5.4.4.1 Piping

Very often, we want to do something with the output of a command. How cumbersome (well, actually: impossible in many cases) would it be, though, if we always had to copy output from the terminal and paste it as the argument of another command! Of course we could write to a temporary file, but fortunately, there's an easier solution: piping solves this problem. The pipe (`|`) can be added after any command, which hands that command's (standard) output as input (NOT arguments) to the following command. To also redirect standard error, ues `|&`.

If for example we want to find all files in a directory which end in `.log`, we can use

Bash

```bash
ls | grep ".log"
```

This passes the output of `ls` to the `grep` command

5.4.4.2 Standard Streams

There are three *standard streams* in terminals:

Name	Acronym	File Descriptor
standard input	stdin	0
standard output	stdout	1
standard error	stderr	2

Commands can read input from stdin and write output to either stdout (if all goes well) or stderr (in case of errors).

File descriptors are one-digit abbreviations used to represent the streams in commands - again, never type more than necessary!

5.4.4.3 Redirection

Streams can be redirected, e.g. into a file. The following operators are typically relevant:

Operator	Description	Example	Explanation
`>`	Send to	`ls / > dirs.txt`	Writes output to `dirs.txt`, overwriting content
`>>`	Append to	`ls ~ >> dirs.txt`	Appends output to `dirs.txt`, retaining content
`<`	Read from	`wc -l < a.log`	Sends content of `a.log` to stdout, read by `wc`

5.4.4.4 Stream Redirection

We can use the file descriptors to only redirect either stdout or stderr to a file, so if we're only interested in seeing info on the screen, but would like errors else preserved in a file in case we want to review them later, we can redirect a command's output as follows:

Bash
```
foobar 2> err.txt
```

The `2` is the file descriptor of stderr, so that will be redirected into "err.txt". To do the same for stdout, we don't even have to type the `1` as `>` is shorthand for `1>`:

Bash
```
foobar > out.txt
```

We can also redirect one stream to where another is currently directed by using `m>&n`. This redirects the stream with file descriptor `m` to where the stream with file descriptor `n` is currently pointed.

Let's explain the two examples the manual [5] gives:

Bash
```
ls > dirlist 2>&1
```

This first redirects the standard output (which `ls` writes non-error information to; this initially goes to the console) to the file named `dirlist`. In a second step, it redirects the standard error (to which `ls` writes error output) to where it's standard output is directed, which is now the file named `dirlist`.

Bash
```
ls 2>&1 > dirlist
```

First redirects the standard error to where the standard output points: the console. In a second step, it redirects the standard output (remember, `>` means `1>`) to a file named `dirlist`.

> To simply suppress output, you can redirect it to `/dev/null`

5.4.4.5 File Redirects

Sometimes we have programs that only read from files, not e.g. from stdin, or we want to further process something that is written to a file by a program. We can of course use multiple steps to do that, but there's a shortcut (officially called *process substitution*):

`<($expression)` executes `$expression` and provides its output as a file descriptor. This allows us to e.g. use `diff <(ls dirA) <(ls dirB)` to compare two directories' contents.

On the other hand, we can also use `>($expression)` to run `$expression` on output a program writes to a file. We can e.g. use `cat x.log | tr -d " \t\n\r" > >(sort)` to output the lines sorted after trimming some leading / trailing white space from a log file. The first `>` redirects stdout, whereas the second `>` is part of `>(...)` - they're both needed!

[5] https://www.gnu.org/software/bash/manual/html_node/Redirections.html

6. Unix Tools

6.1 Introduction

This chapter introduces (some of) the most important tools and shows you how to use them. This is definitely content you'll use! You're strongly encouraged to try the examples as we go along - and of course to do the exercises. There's not much to understand, but a lot to remember, so it definitely helps to build muscle memory!

6.1.1 Code Repository

6.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Command Hoisting	5	n/a	***5***
Filesystem & Files	10	n/a	***10***
Infomation Gathering	5	15	***15***
Content	10	30	***30***
Text Manipulation	15	45	***45***
Miscellaneous	5	15	***15***
Total	**50**	**160**	**210**

The time allocated for the journeyman's piece is used combined with the chapter "Shell Scripts". There is a total of 120 minutes allocated to it.

6.3 Tools

The full documentation for many of these tools (and many more) can be found here [1]

6.3.1 Command Hoisting

6.3.1.1 xargs

Used to pass output of one program to another one as arguments, calling it once for every output element.

As an example, we can pass the file names in a directory (returned by `ls`) to `wc` to get a count of each file's lines (both tools: see below):

Bash
```bash
ls | xargs wc -l
```

Without `xargs`, `wc -l` would be called once, getting the output of the directory listing as input, returning the number of elements in the directory.

- To use the string passed on by `xargs` explicitly, we can define a placeholder (e.g. `%`) using `-I %` and then use it to call the command we're looking to execute: `ls | xargs -I % wc -l %`
- `eval` cannot be called from `xargs`. Instead, we can use e.g. `bash -c` (or shorter: `sh -c`).

6.3.1.2 tee

To write output both to stdout and a file, `tee` can be used. This is particularly useful to persist intermediate results where the output is going to be further used.

Bash
```bash
ls -al | tee listing.txt
```

If the option `-a` is passed, then output will be appended to the given file (otherwise the file will be overwritten).

6.3.1.3 sudo

Execute a command as another user (default: `root`).

Option	Description
`-u $user`	Execute as `$user` instead of `root`
`-E`	Keep existing environment variables
`-i`	Opens an interactive shell for the given user (you're that user until you `exit` the shell)

[1] https://www.gnu.org/software/coreutils/manual/html_node/index.html

6.3.2 Filesystem & Files

6.3.2.1 ls

Let’s look at a first example everyone most likely knows already: the command `ls`, which is short for “list” (yep, never type too much!). We have different ways of calling the command:

Bash
```
ls            # list the contents of the current directory
ls /          # list the contents of the root directory
ls -alh ~     # list the contents of the home directory,
              # showing "hidden" (starting with ‘.‘) files (‘a‘)
              # using "long format" (‘l‘, incuding permissions, file size,...) and
              # unit suffixes (‘h‘).
ls -a -l -h ~ # same as above
```

6.3.2.2 chmod

`chmod` changes file / directory permissions. There are two ways permissions can be modified: by setting them absolutely or by adding / removing them.

To set permissions absolutely, they are encoded in a number which can either be three or four digits long. The last three digits encode permissions for the owner, the group and others. Each such digit is made up by adding:

- 4 for read (`r`) permission
- 2 for write (`w`) permission
- 1 for execute (`x`) permission

As an example, we can call `chmod 740 $file` to grant r/w/x (4+2+1) to the owner, r (4+0+0) to the (owner) group and nothing (0+0+0) to others for `$file`.

The first of the four numbers can be used to grant permission to run a program with the permissions of it’s owner (4, *setuid*) or group (2, *setgid*) or to protect files in a directory by only allowing file / directory owners or root to rename or delete files (1). This is commonly used if the script can be executed by users who don’t have the permissions to access resources the script needs to read or write.

Alternatively, we can use another code to add / remove permissions. In this case, we use an identifier (**u**ser, **g**roup, **o**ther, **a**ll), an operator (`+`, `-`, `=`) and a permission (**r**ead, **w**rite, e**x**ecute, **s**et ID on execution - setuid / setgid), such as e.g. in `chmod ug+x $file` or `chmod a=rw`.

Try the different commands and see what happens by calling `ls -l`.

6.3.2.3 mkdir

To create directories, use `mkdir $path` to create the file with the name which is the last segment of `$path`. A most useful option is `-p`, which allows creating entire directory hierarchies.

6.3.2.4 mv

To move or rename a file or directory, use `mv $source $destination`.

6.3.2.5 ln

`ln $existing $new` by itself creates a hard link [2] , a file system entry named `$new` which points to same data `$existing` points to.

The (probably) more relevant use is to create a symbolic link [3] : `ln -s $existing $new`. Note that when providing a relative path in `$new`, moving the symbolic link will break it: the path is hard coded and will not change.

6.3.2.6 rm

`rm` removes files. Using `rm -d $directory` will remove a directory and `rm -r $directory` will remove an entire directory structure. If the directory is not empty, `rm -d` will refuse to delete, use `rm -r` instead.

6.3.2.7 tar

To compress / uncompress files, we typically use `tar`.

- Compress files: `tar -zcf $archive $files`, `$files` will be expanded and can be multiple `$IFS` separated files
- List files: `tar -ztf $archive`
- Decompress archive: `tar -zxf $archive`

This is what the different options mean:

Option	Meaning
`c`	Create a new archive
`t`	List archive contents
`x`	Extract from archive
`f $archive`	Read from / write to `$archive`
`z`	Use `gzip` to compress / the archive

And here are a few we haven't used but may prove useful:

Option	Meaning
`r`	Add to an existing archive
`v`	List names of file read or written
`-C $directory`	Add from / extract to `$directory` instead of `$PWD`

Try it: create an archive, move it somewhere else, unpack it there and verify that everything worked!

[2] https://en.wikipedia.org/wiki/Hard_link
[3] https://en.wikipedia.org/wiki/Symbolic_link

6.3.3 Information Gathering

6.3.3.1 find

To find a file based on pretty much any information, use `find`. The most common use is probably `find . -name "$pattern"` to find a file named `$pattern` anywhere in or below the current directory (`$pattern` according to file name expansion syntax).

Another common use is to find files accessed, created or changed more / less than a given period of time ago. To do this, one can use `find` with the following options:

Option	Description
`-mmin +$minutes`	Find files **m**odified more than `$minutes` minutes ago
`-mmin -$minutes`	Find files **m**odified less than `$minutes` minutes ago
`-amin +$minutes`	Find files **a**ccessed more than `$minutes` minutes ago
`-amin -$minutes`	Find files **a**ccessed less than `$minutes` minutes ago

- To use regular expressions to match filenames, use `-regex $pattern`. The pattern must match the entire file name.
- Add option `-E` (directly after `find`, before the path to search in) to use regular expressions as we know them. Without this option, commands typically use *basic regular expressions*, *BRE*s.

Time to exercise!

6.3.3.2 which

Use `which $name` to find out where a program named `$name` is to be found. `which` checks all locations in `$PATH` and returns the first match. To find all matches, use `which -a $name`.

6.3.3.3 ps

`ps` shows the running processes. Use argument `a` to include processes other than your own, `x` to include processes which are not connected to a terminal and `u` to show extended information: `ps aux`. Note that the options don't use a dash for historic reasons

6.3.4 Content

6.3.4.1 cat

`cat $file` writes the contents of a file to the console. It can also concatenate multiple files, using `cat $file1 $file2` (with pretty much as many files as one might please). One useful option is `-s` (for "squeeze"), which removes any duplicate empty lines.

Using curly brace expansion, we can also write e.g. `cat {a,b,c}.log`.

Time to exercise!

6.3.4.2 less

Displays the contents of a file and allows navigating within them. Man pages are displayed using `less`.

To navigate, there are lots of commands. The most important ones are probably the following:

Command	Effect
`SPACE`	Scrolls down one "window" - (initially) as much as fits into the visible console
`$n SPACE`	Scrolls down `$n` "windows"
`ENTER`	Scrolls down one line
`$n ENTER`	Scrolls down `$n` lines.
`b`	Scrolls backwards one line
`$n b`	Scrolls backwards `$n` lines.
`g`	Scrolls to the top
`$n g`	Scrolls to line `$n`
`G`	Scrolls to the bottom
`/$pattern ENTER`	Searches for `$pattern`. Subsequent `n` will find the next occurrance, `N` the previous one
`&$pattern ENTER`	Shows only lines containing `$pattern`. `& ENTER` will revert to showing all lines
`q`	Quits `less`

Navigate around the man page of `less` itself to get better acquainted with its commands. Put particular focus on searching / filtering information!

6.3.4.3 tail

Shows the last lines of a file. Mainly useful for listing file content as a file - such as a log file - is being updated by using `tail -f $file` (**f**ollow). Adding `-n $num` will show up to `$num` lines instead of the default 10. Using a `$num` starting with the plus sign (`+`, so e.g. `-n +10`) will list all except for the top `$num` lines.

Time to exercise!

6.3.4.4 head

Shows the first `-n` number of lines (default 10) of a file.

6.3.4.5 column

Formats a file's content in column layout. Use `-t` to have `column` create a nice tabular layout based on the input's content. The default separator used is a space; this can be changed by passing `-s $s` with `$s` single character to use as separator.

Time to exercise!

6.3.4.6 wc

Count the lines, words and bytes (or characters) in a file / `stdin`.

The following options can be provided at once:

- `-l`: count the number of lines (`wc -l`)
- `-w`: count the number of words (`wc -w`)
- `-c` / `-m`: count the number of bytes / characters (`wc -c` / `wc -m`)
  - `-c` is short for "single-byte **c**haracter, whereas `-m` is short for"**m**ulti-byte character"

Time to exercise!

6.3.4.7 diff

Shows the difference between two files. Provide the option `-i` to ignore case and `-b` to ignore white space.

6.3.5 Text Manipulation

6.3.5.1 rev

Outputs the reversed characters in each line of a file (last character to first character).

6.3.5.2 sort

Bash

```bash
sort $FILE     # sorts the lines of $FILE and writes the result to ‘stdout‘
sort -u $FILE # sorts the lines of $FILE, removes duplicates
               # and writes the result to ‘stdout‘
```

Time to exercise!

6.3.5.3 grep

To output lines containing a given string, use `grep`.

Bash

```bash
grep ERROR event.log
```

Some relevant options:

Option	Impact
`-E`	Use regex patterns as we know them for all patterns passed to `grep`.
`-e $pattern`	Add a regex pattern `$pattern` - sensible if more than one is added. Patterns will be combined using `OR`.
`-v`	Invert match, show lines **not** matching (any) pattern.
`--color=auto`	Highlight matching text (wanna create an alias?).
`-B$number`	Also show `$number` lines before each match.
`-A$number`	Also show `$number` lines after each match.
`-C$number`	Also show `$number` lines before and after each match.

Time to exercise!

6.3.5.4 cut

`cut` can remove everything except for specified ranges of each of a file's lines. Three mutually exclusive different options, which each take a comma separated list of ranges as input:

Option	Range unit
`-b`	Bytes
`-c`	Characters
`-f`	Fields (lines are split into fields at each separator)

If `-f` is used, the tab (`\t`) character is used as field separator by default. This can be overridden by passing `-d $separator` additionally.

Ranges are 1-indexed, comma-separated and can be:

- A simple number, which makes the range one unit wide
- Two numbers separated by a dash, which means "from-to"
  - One of the numbers can be left blank (e.g. `2-` or `-5`)

Time to exercise!

6.3.5.5 awk

One of the most useful tools to evaluate column based output is `awk`. The tool parses each line of a file (or `stdin`), matches them to an (optionally) provided regex and performs a given action (default: print the line) for each matching line. Columns of the input can be referred to by `$n`, where `n` is the 1-indexed column number.

The separator used is **NOT** `$IFS`, but `awk`'s own `FS`, which defaults to the space character.

The argument passed to `awk` is enclosed in single quotes and consists of

- an optional pattern to match - if not provided, all lines will match. It can be:
  - a regex to match (enclosed by forward slashes (`/`))
  - a *relational expression*, which in turn can be
    - `$a $op $b`, with `$op` an operator in (`==`, `!=`, `<`, `>`, `>=`, or `<=`)
    - `$a $matchop $regex`, with `$matchop` either `~` (matches) or `!~` (does not match)
    - `$a in $array`
  - a boolean combination of the two (using operators `!` (NOT), `&&` (AND) and `||` (OR))
- an optional action to execute (enclosed by curly braces (`{}`)); if not provided, matching lines will be printed

One common use case is to execute a command on the content of one column, e.g. kill all java processes:

Bash
```bash
ps | awk '/java/ { print $1 }' | xargs kill -9
```

It's interesting to observe that the pattern matching capability of `awk` is often not used - instead `grep` is used to match even though that means typing more:

Bash
```bash
ps | grep java | awk '{ print $1 }' | xargs kill -9
```

`awk` also allows coding more complex aggregations, such as e.g.

Bash
```bash
tail /usr/share/dict/words | awk 'BEGIN{x=""}{x=x","$1}END{print x}' | cut -c 2-
```

The `BEGIN` and `END` blocks are executed exactly once, before the first line or last line are processed, respectively.

> To exclude (instead of include) matches, add an exclamation mark (`!`) in front of the regex to be matched.

Time to exercise!

6.3.5.6 sed

`sed` is used to edit “streams”. Some common edits are:

- Regular expression replaces
- Inserting / Appending text
- Deleting lines
- Changing lines

As an initial (stupid) example, we use the `s` (substitute) edit command to change every “a” to an “ä”:

Bash
```
head /usr/share/dict/words | sed 's/a/ä/g'
```

Without the `g` flag, only the first occurrance per line would be substituted. In the replacement text, `&` has a special meaning: it refers to the entirety of the text matched.

To do multiple edits, we can add further instructions separated by a semicolon.

Bash
```
head /usr/share/dict/words | sed 's/a/ä/g;s/A/Ä/g'
```

Edits can be restricted to specific lines / line ranges by providing so-called *addresses* - in the above examples, we didn’t specify any. If we provide one address, it specifies the lines to be edited. If we provide two separated by a comma, they limit the edits to the range of lines between (and including) them. Such an address can either be a line number, the dollar sign to denote the last line or a regular expression. The address(es) can also be excluded from instead of included in the editing by appending an exclamation mark (`!`) after the address(es).

Let’s look at some examples, at the same time introducing the other edits. First off, we can use the edit command `a` to append text after each match or `i` to insert text before. We’re going to insert “# Look out:” on every line that contains a double “a”:

Bash
```
head /usr/share/dict/words | sed '/aa/ i\
\# Look out: '
```

out: ’ \end{code}

- the newline after the edit command (`a` or `i`) is required
- `a` always adds a new line after the text appended, while `i` is prepended to the line matched.

To delete every line between the one matching exactly “aal” to the last line, we use the `d` edit command as follows:

Bash
```
head /usr/share/dict/words | sed '/^aal$/,$ d'
```

And finally, to redact lines 6 trough 8, we can use the `c` (**c**hange) edit command:

Bash

```
head /usr/share/dict/words | sed '6,8 c \
(redacted: L6-8)
'
```

\end{code}

> To redact each line individually, use the `s` edit command instead of `c`.

Option	Impact
`-E`	Use “extended” regular expression syntax - add this to use the regular expression syntax we looked at
`-i $extension`	Edit the provided file directly (**in**-place) instead of printing to the console. Save a backup with the given `$extension` first - use `''` to omit backing up
`-n`	Do not write to `stdout`
`-f $scriptname`	Execute the edits specified in `scriptname` (instead of inline)

Time to exercise!

6.3.5.7 base64

To encode or decode text in *base 64*, we can call `base64 $file` (or `base64 -d $file` to decode). If `$file` is omitted, the the program will transform standard in (e.g. `echo test | base64`).

6.3.6 Miscellaneous

6.3.6.1 seq

`seq` generates a sequence of numbers. There are up to three parameters:

1. start of sequence (optional, default 1)
2. increment (optional, default 1)
3. end of sequence / maximum

Time to exercise!

6.3.6.2 date

Prints the current date and time (or can set it if called by a superuser).

Can be passed a formatting pattern as an argument:

Format String	Meaning
`%a`	Name of the day of the week, e.g “Mon”
`%d`	Day of the month, e.g.”25”
`%V`	Week of the year
`%b`	Name of the month, e.g. “Dec”
`%m`	Month of the year, e.g. “12
`%Y`	Year, e.g. “2020”
`%H`	Hours, e.g. “23”
`%M`	Minutes, e.g. “42”
`%S`	Seconds, e.g. “59”
`%z`	Time zone offset, e.g. “+0100”
`%n`	Inserts a newline character

The format string needs to start with a plus sign, e.g. `date "+DATE: %Y-%m-%d%nTIME: %H:%M:%S"`.

Using `date`, we can also do date arithmetic by providing offsets for certain parts of the date using option `-d`. Some examples:

- to add one week, we pass `-d +1 weeks`
- to go back in time by three hours, we specify `-d -3 hours`

Other possible units are `years`, `days`, `minutes`, and `seconds`.

6.3.6.3 curl

We’ll use this to also introduce you to the `man` command. Type `man curl` to read up on the command - and use the information to solve the exercises!

Time to exercise!

7. Python

7.1 Introduction

Python is probably the language most widely used by security professionals. Fans will certainly be able to give you lots of reasons why that is the case, but most likely it boils down to the language being pre-installed on most unix distributions, where it is usable as a scripting language (if you don't like bash syntax, maybe this is for you!) as well as a fully fledged general purpose programming language.

Python is somewhat special (as compared to most other programming languages) in that it is *whitespace aware*: whitespace has meaning in Python. Code blocks that are delimited using e.g. braces in other language are defined by indenting.

Fun fact: it has nothing to do with the snake, but is named after Monty Python [1] .

7.1.1 Code Repository

The code snippets for this chapter can be found here [2] . If you don't yet have access, please ask to have your user added!

7.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Tooling	10	n/a	***10***
Syntax Mapping	30	75	***105***
Language Features	40	145	***185***
Algorithms	20	45	***65***
Libraries	20	95	***115***
Total	**120**	**360**	**480**

[1] https://en.wikipedia.org/wiki/Monty_Python
[2] https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/Python

7.3 Tooling

VSCode offers a great environment for working with *Python*. Before we can get started, please make sure you have Python itself installed - we need a 3.x version. Here's one way to check:

Bash

```bash
which -a python  | xargs -I % bash -c "echo -n %: && % --version"
which -a python3 | xargs -I % bash -c "echo -n %: && % --version"
```

If you don't, best follow the instructions here [3] .

An easy way to execute your code is to use *Code Runner* again. Make sure the correct version of python is configured to be run in the settings.

To configure code runner to execute the correct version of python, adapt `.vscode/settings.json` to reflect the executable you found above (aliased `$python` below):

JSON

```json
{
  "code-runner.executorMap": {
    "python": "$python -u"
  }
}
```

Alternatively, you can spawn a python shell by calling the program you found above (e.g. `python3`).

7.4 Syntax Mapping

We're again going to be mapping Python's syntax to our pseudocode, but we'll progress in a slightly different order this time.

7.4.1 Comments

Comments in Python start with a hash character. They can either be at the start of the line or follow some code. If a hash occurs within a string, it will not denote a comment. There is no notion of inline comments, so anything that follows the hash on a line will be considered a comment. There are also no multi-line comments (using the "docstring" format we will see later can be used as a workaround, though).

Python

```python
# This is a comment
print("1") # this is also a comment
print("2")#and this
print("3#but this isn't")
```

[3] https://wiki.python.org/moin/BeginnersGuide/Download

7.4.2 Program Structures

7.4.2.1 Methods -> Functions

The first thing to note is that Python says "function" where we say "method". It uses the word "method" specifically to refer to a method defined as part of a *class* (don't worry about it now, we'll look at classes in the "language features" section). Where not otherwise noted, we will use Python's terminology in this chapter.

Python

```python
def calculateFactorial(input):
  # calculations
  return result

```

Pseudocode

```
method calculateFactorial = (input) => {
  // calculations
  return result
}
```

7.4.3 Data Structures

7.4.3.1 Constants and Variables

Python does a lot by convention instead of enforcing constraints. One example is that there are no constants - programmers are requested to adhere to these conventions and "play nice". A value that should not be changed in python is simply declared using only upper case letters and the undersccore (_) character (also called all caps snake case [4]).

Python

```python
x = 9
y = 5
# No constants in Python
print(x) # 9
```

Pseudocode

```
variable x = 9
variable y = 5
constant z = 2
print(x) // 9
```

[4] https://en.wikipedia.org/wiki/Snake_case

7.4.3.2 Arrays -> Lists

The data structure built in to Python which best maps to our pseudocode's arrays are *lists*. Lists are handled very much the same as arrays in our pseudocode syntax, and are also 0-indexed. There are a few handy shortcuts, though:

- `$list.pop($index)` will remove the element at the given index.
- `$list.remove($item)` will remove the item `$item` first found. This is a shortcut to looping through the array indexes in ascending order.
- `$list.index($item)` will return the index at which the first occurrence of `$item` is found.
- `$list.sort(key=$fnc, reverse=$descending)` will sort the elements of a list using `__lt__` to compare pairs of items.
  - `$fnc` is an optional function which converts each item in `$list` to something comparable by using the `__lt__` operation.
  - `$descending` has a default of `False`, resulting in the list being sorted in ascending order. Setting it to `True` will sort descending.

On the other hand, where our pseudocode syntax allows adding array elements at arbitrary positions (extending arrays as needed), Python only allows appending to the end.

Python

```python
foo = [2, 4, 9, 1, 7]
first = foo[0]
len = len(foo)
# does not exist in Python
foo.append(9) # appends at the end
foo.extend([4, 8]) # appends items
foo.pop(2) # removes the third item
foo.remove(4) # removes the first "4"
```

Pseudocode

```
input          = [2, 4, 9, 1, 7]
constant first = input[0] // 2
constant len   = input.length // 5
input[5]       = 18 // [2, 4, 9, 1, 7, 18
// n/a
// n/a
// n/a
// n/a
```

A data structure called array does exist in an extension library. It more efficiently stores data and supports additional operations, but it's also more restrictive in some areas.

Time to exercise!

7.4.3.3 Maps -> Dictionaries

Maps are called *dictionaries* in Python. They offer three special methods that give us access to either all keys, all values or all item *tuples* (see further below) of a dictionary. Please note that these are not copies, but rather "views" of the original dictionary, so if that changes, our views will change as well. Here is how to map them to our pseudocode syntax:

Python

```
shapeCorners = {
  "Triangle":  3,
  "Rectangle": 4,
  "Pentagon":  5,
}
print(shapeCorners["Rectangle"])
x = len(shapeCorners) # 3
k = shapeCorners.keys()
v = shapeCorners.values()
i = shapeCorners.items()
```

Pseudocode

```
constant shapeCorners = {
  "Triangle":  3,
  "Rectangle": 4,
  "Pentagon":  5,
}
print(shapeCorners["Rectangle"])
// does not exist in our pseudocode
// does not exist in our pseudocode
// does not exist in our pseudocode
// does not exist in our pseudocode
```

A word about the methods we don't know from our pseudocode:

- `len(...)` simply returns the number of items in the dictionary - this is the same as the number of keys
- `keys()` returns an *iterator* over all the keys. This is a special structure that can be looped over (see below), but doesn't give direct access to individual elements. If you need that, you can simply surround it by `list(...)` (e.g. `list(shapeCorners.keys())`). The above example returns `dict_keys(["Triangle", "Rectangle", "Pentagon"])`.
- `values()` returns an iterator over all values. In the above case, it's `dict_values([3, 4, 5])`
- `items()` returns an iterator over all item *tuples* (see below for details), which here is `dict_items([("Triangle", 3), ("Rectangle", 4), ("Pentagon", 5)])`.

> The method `get` is a nice addition which allows us to pass the key we want and a second parameter which will be returned if the key doesn't exist in the dictionary.

Time to exercise!

7.4.4 Input and Output

7.4.4.1 Printing

To output information to the user, we'll use the `print` function. The mapping to our pseudocode is as follows:

Python

```
print("Hello, Python")
```

Pseudocode

```
print("Hello, Pseudocode")
```

7.4.4.2 Reading

Python

```

x = input("Enter your choice: ")
```

Pseudocode

```
print("Enter your choice: ")
constant x = read()
```

7.4.5 Control Structures

7.4.5.1 Conditions

In Python, conditions are expressed as follows:

Python

```
if x < 5:
  print("smaller")
else:
  print("greater or equal")

```

Pseudocode

```
if x < 5 then
  print("smaller")
else
  print("greater or equal")
end if
```

Python has an additional syntax element which allows expressing multiple alternatives: `elif`. You can for example write the following:

Python

```
if x < 0:
  print("negative")
elif x > 0:
  print("positive")
else:
  print("exactly zero")
```

7.4.5.1.1 Containment

A special syntax exists to check whether or not an item exists in a list or a key in a dictionary:

Python

```python
if "Shabbat" in days:
  print("Oh, you're jewish? Nice to meet you!")
```

7.4.5.1.2 Conditionals

If we only have conditions, assigning different values to a variable is quite verbose:

Python

```python
if condition:
  foo = "Great!"
else:
  foo = "Oh no ;-("
```

Fortunately, there is a simpler way of doing this!

Python

```python
foo = "Great!" if condition else "Oh no ;-("
```

This syntax hasn't always been around, so you may find some workarounds if you read code. A particularly interesting example is the following:

Python

```python
foo = ["Oh no ;-(", "Great!"][condition]
```

This works because `True` maps to 1 and `False` maps to 0.

Time to exercise!

7.4.5.2 Case Matching

There isn't really a case expression as such in Python, but we can map our pseudocode using a dictionary:

Python

```python
cases = {
    plant: "water",
    animal: "feed",
    gas: "inhale",
    liquid: "surf",
    solid: "decorate",
}
print(cases.get(objectType, "stare"))
```

Pseudocode

```
if objectType
  matches plant   then print("water")
  matches animal  then print("feed")
  matches gas     then print("inhale")
  matches liquid  then print("surf")
  matches solid   then print("decorate")
  else                 print("stare")
end if
```

> The disadvantage (or advantage, if you ask the enthusiasts) is that the `cases` variable can be changed at runtime, so you're never quite sure nobody meddled with your code. . .

7.4.5.3 Loops

- A "while" loop:

Python

```python
i = 0
while True:
  if i >= 5:
    break
  print(i)
  i = i + 1

```

Pseudocode

```
variable i = 0;
loop
  exit if i >= 5;

  print(i++);

end loop
```

Output

```
0
1
2
3
4
```

The above syntax, which most closely maps to our pseudocode syntax, is not the most common one. We could write the following code:

Python

```python
i = 0
while i < 5:
  print(i)
  i = i + 1
```

But in fact, we wouldn't even do that, we would use the `range` function as follows:

Python

```python
for i in range(5):
  print(i)
```

- A “for”-loop:

Python

```python
NUMBERS = [1, 2, 3, 5, 8]

for i in range(len(NUMBERS)):

  print(NUMBERS[i])

```

Pseudocode

```
constant numbers = [1, 2, 3, 5, 8];
variable i = 0;
loop
  exit if i >= numbers.length;
  print(numbers[i++]);
end loop
```

Output

```
1
2
3
5
8
```

As before, the above is a close match of our pseudocode syntax. In reality, the following code would be used to achieve the same result:

Python

```python
for n in [1, 2, 3, 5, 8]:
  print(n)
```

If we want to loop over a map, we can use the same syntax, but the loop variable (`n` above) will be assigned each key in the map, so we need to access values separately:

Python

```python
alpha = { "a": "A", "b": "B" }
for n in alpha:
  print(n + ": " + alpha[n])
```

Python has an `else` keyword in loops, which can be used to do something once the loop exits “normally” - that means not via a `break` statement. It’s very unlikely you’re ever going to use this, but maybe you’ll encounter this somewhere in the wild...

Time to exercise!

7.5 Languages Features

7.5.1 F-Strings

F-strings (formatted strings) allow us to refer to code from within text:

Python

```python
def introduce(forename, surname):
  print(f"Hi, my name is {forename} {surname}.")
```

7.5.2 Modules & Packages

Modules are units of (reusable) code. They can be bundled into packages.

A module `foo` is defined in a file named `foo.py`. If it is directly located in the root of a python project, it can be imported into other modules using `import foo`. If `foo.py` contains functions `doThing` and `printItem`, then these can be imported

- collectively using `from foo import *` and then used e.g. as `foo.doThing()`
- individually using `from foo import printItem` and the used as `printItem(...)`

Multiple modules can be bundled into a package. This is simply done by creating a directory with the package name (e.g. `bar`) containing the module files and a special file called `__init__.py` (can be empty, its content will be executed when the package is imported - this can e.g. be use to load configuration values such as logging levels). When importing, packages (which can be hierarchical) are mentioned as well. We can thus do the following (assuming there's a module `foo` with function `doThing` in a package `bar`):

Import Package

```python
import bar

bar.foo.doThing()
```

Import all Modules from Package

```python
from bar import *

foo.doThing()
```

Note that this way of importing (called "wildcard import") doesn't work in classes or functions. This is due to an optimization by the compiler which greatly increases execution speed. Even in places you **can** use wildcard imports, you shouldn't, though: it makes code harder to read and can easily lead to name clashes (same name used locally as defined in the imported package).

Import one Module from Package

```python
from bar import foo

foo.doThing()
```

Import Function from Module in Package

```python
from bar.foo import doThing

doThing()
```

A package can also contain a `__main__.py` file. If it does, the module can be executed using `python -m $packageName`.

Time to exercise!

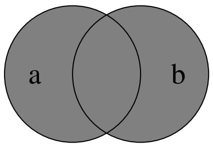
7.5.3 Sets

Sets are collections that contain *unique* items - there cannot be any duplicates. They are not ordered, so there is no way to access individual items, only to loop over them.

Python

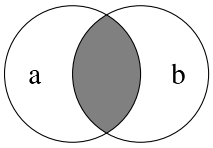
```python
foods = { "coconuts", "spam" }
for i in foods:
  print(i)
```

Sets additionally offer the following methods (we're going to asssume sets `a` and `b` are defined as `a = { "apples", "oranges" }` and `b = { "bananas", "oranges" }`):



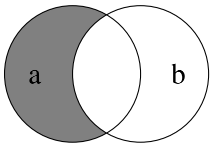
Python

```python
c = a.union(b) # c = { "apples, "bananas", "oranges" }
```



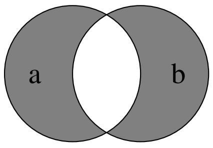
Python

```python
c = a.intersection(b) # c = {"oranges" }
```



Python

```python
c = a.difference(b) # c = { "apples }
```



Python

```python
c = a.symmetric_difference(b) # c = { "apples, "bananas" }
```

- You can also get the maximum and minimum item in a set by using `min(...)` and `max(...)`.
- To create an empty set, use `set()` - just using `{}` would create an empty dictionary.

Time to exercise!

7.5.4 Tuples

Tuples can be looked at as unmodifiable lists. Their elements are accessed and iterated over the same way as lists', but any operation that attempts to modify a tuple will fail.

Python

```python
tup = (1, 2, 3, 5, 8)
first = tup[0]
print(first)
last = tup[-1]
print(last)
mid = tup[1:-1] # mid will be (2, 3, 5); "to" (behind the colon) is exclusive
print(mid)
for i in tup:
  print(i)
```

We can also check if an element is part of a tuple, e.g.

Python

```python
tup = (1, 2, 3, 5, 8)
if 4 in tup:
  print("got it!")
else:
  print("not here...")
```

Any comma separated items are considered elements of a tuple:

Python

```python
pair = "m", "f" # same as ("m", "f")
single = "I", # same as  tuple("I") or ("I",)
```

7.5.5 Strings

Strings can be inside either

- single quotes, which can then contain double quotes, e.g. `'He said "woof"'`
- double quotes, which can contain single quotes, e.g. `"It's great!"`
- three double quotes, which can span multiple lines and contain single and double quotes, e.g.

Python

```python
from textwrap import dedent
print(dedent("""\
  So haben sie mit dem Kopf und dem Mund
  Den Fortschritt der Menschheit geschaffen.
  Doch davon mal abgesehen und
  bei Lichte betrachtet sind sie im Grund
  noch immer die alten Affen."""
))
```

\end{code}

We're using the `dedent` function to allow indenting the multiline string in the source. The backslash right after the three quotes is there to avoid a leading newline. Try it!

Strings in Python are sequence types like lists and tuples, so we can use many of the same operations on them as we would on e.g. tuples:

Python

```python
word = "foobar"
first = word[0]
print(first)
last = word[-1]
print(last)
mid = word[1:-1] # mid will be "ooba"
print(mid)
for i in word:
  print(i)
```

We also have quite a few additional useful methods, such as

- `$string.count($substring, $start, $end)`: number of occurrences of `$substring` in `$string[$start:$end]`
- `$string.find($substring, $start, $end)`: index of the first character of the occurrence of `$substring` in `$string[$start:$end]`
- `$string.split($separator)`: creates an array containing the parts of `$string` that are separated by `$separator`.
  - If `$separator` is not provided, spaces will be used as separators
- `$separator.join($iterable)`: joins items in `$iterable` using `$separator`
  - As an example, `", ".join(["hi", "you"])` outputs `"hi, you"`.
- `$string.translate(str.maketrans($from, $to))`: translates a string by applying a character to character mapping from the translation table constructed by `str.maketrans($from, $to)`
  - An example would be to decode a cesar cypher:

Python

```python
from string import ascii_lowercase, ascii_uppercase
alph = ascii_lowercase + ascii_uppercase
print("Kl, wkhuh!".translate(str.maketrans(alph[3:] + alph[:3], alph))) # => Hi,
    there!
```

We've already met f-string. There are also **b**inary strings (`b"..."`) containing binary data and **r**aw strings (`r"..."`), in which escape characters will be included literally. We will see examples of both further below.

Time to exercise!

7.5.6 Unpacking

Collections can be “unpacked”, which means taking them apart and treating each item individually:

Python
```python
first, *middle, last = collection
outermost = last, first
rotated = [ *middle, *outermost ]
```

`*middle` takes all the items that are not individually assigned.

We can use tuple unpacking to get both key and value of each item (which is a tuple) of a dictionary:

Python
```python
opposites = { "lost": "found", "alive": "dead" }
for key, value in opposites.items():
  print("The opposite of " + key + " is " + value)
```

Note that if `collection` is a dictionary, the unpacked items (`*collection`) will be the keys, not the items as we might expect. To merge two dictionaries, we need to use two stars to get the individual items (tuples):

Python
```python
vertical = { "top": "2px", "bottom": "5px" }
horizontal = { "left": "5px", "right": "5px" }
padding = { **vertical, **horizontal }
```

Dictionaries know another kind of unpacking, which we can use to call a function with the values of a dictionary for which the keys correspond to the function parameters.

Python
```python
tom = { "surname": "Dooley", "forename": "Tom" }
def introduce(forename, surname):
  print(f"Hi, my name is {forename} {surname}!")
introduce(**tom)
```

Methods themselves can also be declared to accept variable numbers of parameters (`*jobs`) and also *named parameters* (`**family`).

Python
```python
def introduce(forename, surname, *jobs, **family):
  print(f"Hi, my name is {forename} {surname}.")
  if len(jobs) == 1:
    print(f"I am a {jobs[0]}.")
  elif len(jobs) > 1:
    print("My jobs are:")
    for j in jobs:
      print(f"- {j}")
  if len(family) > 0:
    print("My family consists of ")
    for i in family:
      print(f"- {i} ({family[i]})")

introduce("Tom", "Dooley", "Gardener", "Architect", Tessa = 42, Adam = 5, Eve = 3)
```

Time to exercise!

7.5.7 Enumerable

In some cases, when we loop over a list, we don't only need the item itself, but also the index it is stored at. Python gives us a method `enumerate`, which allows us to write the following:

Python

```python
days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
for index, item in enumerate(days, start = 1):
  print("Day " + str(index) + " of the Week is " + item)
```

- `start = 1` tells `enumerate` to start with 1 instead of 0
- `str` is used to convert the integer value `$index` to a string so it can be appended. Alternatively, we could use an f-string.

For dictionaries, we can even combine `enumerate` with a call to `item()` to get index, key and value:

Python

```python
opposites = { "lost": "found", "alive": "dead" }
print("Some opposites")
for index, (key, value) in enumerate(opposites.items()):
  print(str(index) + ": " + key + " - " + value)
```

7.5.8 File Handling

Opening a file is very simple in python, we simply call the `open` function with the name of a file to open and optionally the editing mode and whether the file is to be considered to contain text or binary data.

Parameter	Meaning
`x`	Create file only
`r`	Read from file - default editing mode
`w`	Overwrite file
`a`	Append to file
`t`	Text content - default content mode
`b`	Binary content

Python

```python
input = open("names.txt")
output = open("shortnames.txt", "w")
for name in input:
  if len(name) < 7:
    output.write(name)
input.close()
output.close()
```

7.5.9 Classes

The following is a very high level overview of a rather complex topic. Feel free to skip it if you don't feel up to the task at the moment.

Classes are a concept of object oriented [5] languages. The basic idea is that classes specify a blueprint for the creation of *objects*, which represent real world things (e.g. cars or people).

Very basically, classes define the data relevant to objects (e.g. horse powers for cars, their name or birth date for people) and methods (e.g. "marry", defining a family name) to modify this data (an object's *state*) or to perform calculations based on it as a response to messages passed between objects.

Python

```python
 1 class Human():
 2   NUM_LEGS = 2
 3
 4   def __init__(self, name):
 5     self._name = name
 6
 7   @classmethod
 8   def numLegs(cls):
 9     return cls.NUM_LEGS
10
11   def sayHello(self):
12     return f"Hello, I'm {self._name}"
13
14 tom = Human("Tom Dooley")
```

So, what's going on here? Let's look at the relevant lines:

- Line 1: a class with name `Human` is declared.
- Line 2: declares a class wide variable - it exists only once for the entire class instead of once for each object (class instance)
- Line 4: the class' initializer `__init__`, which is called when a new class instance (an object) is created.
  - the first parameter (the typical name is `self`) will be implicitly added when the class' *constructor* is called. It refers to the newly created object itself.
  - `name` has to be provided explicitly to create a new instance.
- Line 5: the parameter `name` is assigned to the instance variable `_name`, which is created implicitly.
- Line 7: `@classmethod` is a method *annotation*, which states that the following method (`numLegs`) should only exist once for the entire class instead of once for each object (class instance).
- Line 8: the class wide method `numLegs` has to declare at least one parameter. This mandatory first parameter refers to the class and will also be added automatically when the method is called. The convention is to call the parameter `cls`.
- Line 11: the instance wide method `sayHello` must declare at least one parameter. This mandatory first parameter refers to the instance and will also be added automatically when the method is called. The convention is to call the parameter `self`.
- Line 14: the *constructor* of the class `Human` is called, which leads to a new instance, on which the `__init__` method will be invoked.

[5] https://en.wikipedia.org/wiki/Object-oriented_programming

Classes can *extend* other classes - they can *inherit* their state and methods. In Python, a class can extend multiple *superclasses*.

Python

```python
 1 class Horse(object):
 2   NUM_LEGS = 4
 3 
 4   @classmethod
 5   def numLegs(cls):
 6     return cls.NUM_LEGS
 7 
 8   def sayHello(self):
 9     return "whinny"
10 
11 class Human():
12   NUM_LEGS = 2
13 
14   def __init__(self, name):
15     self._name = name
16 
17   @classmethod
18   def numLegs(cls):
19     return cls.NUM_LEGS
20 
21   def sayHello(self):
22     return f"Hello, I'm {self._name}"
23 
24 class Centaur(Horse, Human):
25   def __init__(self, name):
26     Horse.__init__(self)
27     Human.__init__(self, name)
28 
29   def sayHello(self):
30     return Human.sayHello(self)
31 
32 tom = Centaur("Tom Dooley")
33 print(f"{tom.sayHello()}, I'm a {type(tom).__name__}. I have {tom.numLegs()} legs.")
```

Let’s see what this extended piece of code does.

- Line 24: a class `Centaur` is declared, which extends both `Horse` and `Human`.
- Line 25-27: the `__init__` method of `Centaur` calls the `__init__` methods of both `Horse` and `Human` explicitly.
- Line 33: prints “Hello, I’m Tom Dooley, I’m a Centaur. I have 4 legs.”
  - `Centaur`’s method `sayHello` calls the same method in `Human`.
  - `type(tom).__name__` returns the name of the `Centaur` class.
  - `Centaur`’s method `numLegs` is inherited from `Horse`. This is because `Horse` was mentioned first in line 24.

Time to exercise!

7.5.10 Docstrings

Python has a special syntax to document modules, classes, functions and methods, which is called *docstrings*. A docstring is enclosed in six double quotes (three each, `"""..."""`) and is the first statement in the documented block. During program execution, it will be ignored - just like a comment. Doctrings can e.g. be displayed in IDEs.

Time to exercise!

7.5.11 Exceptions

Sometimes things go wrong. If we already know they can, we can deal with them. In Python, this is how this works:

Python

```python
 1 try:
 2   raise NotImplementedError("nothing here") # just for the sake of demonstration
 3   # raise ValueError("what value?")
 4   # raise RuntimeError("this is unexpected!")
 5   # pass
 6 except NotImplementedError:
 7   print("Not implemented...")
 8 except (ValueError, NameError) as err:
 9   print(err)
10 else:
11   print("Oh, no exception?")
12 finally:
13   print("This runs in any case!")
```

This is what's happening:

- Line 1: the following code block is considered to potentially raise an exception
- Line 2: we manually raise an exception - this would usually happen in some other code we call
- Line 5: `pass` is a "no-op" (no-operation) command. It's there to avoid having to change indentations if commenting parts of your code.
- Line 6: "catches" an exception of type `NotImplementedError`
- Line 7: executed if an exception was caught on line 6
- Line 8: "catches" an exception of either type `ValueError` or `NameError` (which of course will never happen in this case) and assigns the exception instance to variable `err`
- Line 9: prints a string representation of the exception
- Line 10 / 11: code will only be executed if no exception was caught.
- Line 12 / 13: code will always be executed, regardless of whether or not an exception was raised - regardless of whether or not it would have been caught

Try what happens if you uncomment another line of 2 - 5.

Time to exercise!

7.5.12 Operator Overloading

Python allows defining methods that will be executed when an operator is used on objects. A class can overload an operator by defining a specifically named method. An example of this is already implemented in arrays, allowing you to write `arr += [5, 6]` instead of `arr.extend([5 6])`. This specifically works because arrays implement the method `__iadd__`.

> The methods starting with `i` are designed to work ***in**-place*: modifying the original object. If such a method doesn't exist, but it's "basic" equivalent does (in the above case, if `__add__` is implemented), that will be executed, resulting (by convention) in a new object to be created - as if `arr = arr + [5, 6]` had been written in the first place.

The full list of all such special methods can be found here [6] - it's not that simple to read, so just store the link away safely for future reference ;-)

Time to exercise!

[6] https://docs.python.org/3/reference/datamodel.html#basic-customization

7.6 Algorithms

7.6.1 Filtering

The built-in `filter` function allows us to create an iteator containing just those elements for which a given function returns `True`:

Python

```python
print(list(filter(lambda x: x % 2 == 0, [1,2,3,5,8,13])))
```

- a lambda [a] is an anonymous function. We can also pass a normally declared function or method to `filter`.
- `filter` returns an iterator. We have to convert it to e.g. a list to print it.
- The `%` operator means “modulo”. It returns the remainder of a division. As an example, `13 % 3 == 1` because **13** is $4 \times \mathbf{3} + \mathbf{1}$.

Time to exercise!

7.6.2 Mapping

Transforming iterables can be done via the `map` function:

Python

```python
print(list(map(lambda x: x**2, range(1, 5))))
```

Like with `filter`, the result of the `map` operation is an iterator.

Time to exercise!

[a] https://en.wikipedia.org/wiki/Lambda%28programming%29

7.7 Built-In Libraries

7.7.1 functools

The functools [7] library contains different functional programming utilities. One that's particularly useful is `reduce`.

Python

```python
1 from functools import reduce
2
3 def sum(inputs):
4   return reduce(
5     lambda aggregate, item: aggregate + item,
6     inputs
7   )
8
9 print(sum([2, 18, 22]))
```

What's happening here?

- Line 5: a lambda function which takes as input the previously calculated aggregate (`aggregate`) and the current item (`item`) and returns their sum
- Line 6: the sequence over which the `reduce` function will iterate

If you need the index of a given item as well, you can use `enumerate`:

Python

```python
 1 from functools import reduce
 2
 3 def positionWeightedAverage(values):
 4   tup = reduce(
 5     lambda aggregate, item: (aggregate[0] + item[0] * item[1], aggregate[1] + item[0]),
 6     enumerate(values, start=1),
 7     (0, 0)
 8   )
 9   if len(tup) > 0:
10     return tup[0] / tup[1]
11   return 0
12
13 print(positionWeightedAverage([4, 2, 7]))
```

This one's a bit more complex - let's see what's going on!

- Line 7: the last argument to the `reduce` function is optional. We need it to start with a tuple containing two zeros
- Line 6: we use `enumerate` to create tuples containing the index along with each item in `values`, starting with index value 1.
- Line 5: this lambda function creates a new tuple of
  - the product of index and value added to the previous sum of products
  - the index itself added to the previous sum of indexes

Time to exercise!

[7] https://docs.python.org/3/library/functools.html

7.7.2 sys

The sys[8] library gives you access to the running code's environment.

When running a Python script, `sys.argv` will contain its arguments, with `sys.argv[0]` set to the name of the script executed.

At the end of a script, we should normally exit, providing an exit code. This is done using `sys.exit($code)`.

7.7.3 argparse

To simplify life when dealing with Python scripts, the argparse[9] library offers a convenient wrapper creating *usage* information and parsing options provided.

> You may also want to check out the 3rd party click[a] library.

7.7.4 os

In the os[10] libary, we find wrappers for the operating system's tools in a portable way.

A few examples:

- `getcwd`: current working directory
- `sep`: path separator
- `getuid`: current user's ID

Additional utilities for file path manipulation are separated into the library os.path[11]. We can e.g. use `exists($path)` to check if there is a file at `$path`.

[8] https://docs.python.org/3/library/sys.html
[9] https://docs.python.org/3/library/argparse.html
[a] https://click.palletsprojects.com/
[10] https://docs.python.org/3/library/os.html
[11] https://docs.python.org/3/library/os.path.html

7.7.5 datetime

When dealing with real world systems, we often require date and time handling. The datetime [12] library offers lots of convenient methods for that.

To convert dates to string representations or vice versa, we can use `strftime` (**str**ing-**f**ormat-**time**) and `strptime` (**str**ing-**p**arse-**time**).

They each take a format string as an argument (in the case of `strptime`, the `datetime` object comes first) consisting of the following (and some other) format expressions:

Format String	Meaning
`%a`	Abbreviated name of the day of the week, e.g "Mon"
`%A`	Full name of the day of the week, e.g "Monday"
`%d`	Day of the month, e.g."25"
`%V`	Week of the year
`%b`	Abbreviated name of the month, e.g. "Dec"
`%B`	Full name of the month, e.g. "December"
`%m`	Month of the year, e.g. "03" or "12"
`%Y`	Year, e.g. "2020"
`%H`	Hours, e.g. "05" or "23"
`%M`	Minutes, e.g. "08 or"42"
`%S`	Seconds, e.g. "07" or "59"
`%f`	Microseconds, e.g. "004394" or "127100"
`%z`	Time zone offset, e.g. "+0100"

We can also use the method `timedelta` to create an offset to add or subtract from dates. It accepts the following named arguments:

- `weeks`
- `days`
- `hours`
- `seconds`
- `minutes`
- `milliseconds`
- `microseconds`

Python

```python
from datetime import datetime, timedelta

start = datetime.strptime("04-Dec-2020, 11:00:00", "%d-%b-%Y, %H:%M:%S")
print(f"From {start} to {start + timedelta(days=8)}")
```

Time to exercise!

[12] https://docs.python.org/3/library/datetime.html

7.7.6 json

Python's json [13] library takes care of transforming JSON from and to Python objects.

Python
```python
import json
alice = json.loads("""{
  "name": "Alice"
}""")
print(alice)
couple = alice
couple["spouse"] = { "name": "bob" }
print(json.dumps(couple))
```

Time to exercise!

7.7.7 base64

The base64 [14] library offers functions for encoding and decoding in different BaseXX alphabets, among them Base64, but also e.g. Base32.

Python
```python
import base64
print(base64.b64decode(b"V2VsbCwgaGVsbG8gdGhlcmUh"))
```

7.7.8 hashlib

The hashlib [15] library offers implementations of hashing algorithms. An example:

Python
```python
from hashlib import sha1
print(sha1(b"Content").hexdigest()) # => 4f9be057f0ea5d2ba72fd2c810e8d7b9aa98b469
```

Have a look at cryptography [a] for encryption / decryption.

[13] https://docs.python.org/3/library/json.html
[14] https://docs.python.org/3/library/base64.html
[15] https://docs.python.org/3/library/hashlib.html
[a] https://cryptography.io/en/latest/

7.7.9 random

For simple pseudo-random numbers, we can use the random [16] library. In one of the exercises, I suggested you use `random.randrange` to draw a random number. We could also have used `random.choice` to select an element out of the list of foods.

> If you need more random random numbers, have a look at the secrets [a] library.

7.7.10 re

There are three particularly relevant methods in the re [17] (**r**egular **e**xpression) library:

- `search`: find the first match
- `finditer`: find all non-overlapping matches
- `sub`: replace all non-overlapping matches

There are two ways of using these methods:

- standalone, with the pattern provided as string
- called on a compiled, reusable *pattern object*

Let's start with an example for the standalone case, searching for the frist match:

Python

```python
import re
match = re.search(r"\bspam\b", "Spam, spam, spam, spam, spam!")
print(f"First match found between position {match.start()} and {match.end() - 1}")
# => First match found between position 6 and 9
```

We can also compile a pattern and then use it to find matches:

Python

```python
import re
pattern = re.compile(r"\bspam\b")
for match in pattern.finditer("Spam, spam, spam, spam, spam!"):
  print(match.start(), end=" ") # No linebreak due to ‘end=" "‘
# => 6 12 18 24
```

We can also pass the usual flags. In Python, they are combined with a logical OR (`|`). The most relevant flags are

Short Form	Long Form	Meaning
`re.I`	`re.IGNORECASE`	Case insensitive
`re.M`	`re.MULTILINE`	Match across line breaks
`re.S`	`re.DOTALL`	Single line - `.` also matches newline
`re.A`	`re.ASCII`	ASCII instead of Unicode

[16] https://docs.python.org/3/library/random.html
[a] https://docs.python.org/3/library/secrets.html
[17] https://docs.python.org/3/library/re.html

Python

```python
import re
pattern = re.compile(r"\bspam\b", re.I)
for match in pattern.finditer("Spam, spam, spam, spam, spam!"):
  print(match.start(), end=" ") # No linebreak due to `end=" "`
# => 0 6 12 18 24
```

Finally, here's an example for replacements:

Python

```python
import re
print(re.sub(r"^(?:https?:\/\/)?(.*)$", r'<a href="https://\1">\1</a>', """
www.google.com
www.ch.ch
""".strip(), flags=re.M))
# => <a href="https://www.google.com">www.google.com</a>
# => <a href="https://www.ch.ch">www.ch.ch</a>
```

Python has a neat addition that allows us to format and comment regular expressions nicely: adding the flag `re.VERBOSE` will lead to whitespace being ignored as long as they're not inside brackets and to commenting being enabled. Here's a nice example almost directly from the documentation (slightly tweaked):

Python

```python
import re
pat = re.compile(r"""
  \s*                 # Skip leading whitespace
  (?P<header>[^:]+)   # Header name
  \s* : \s*           # Whitespace, a colon and possibly more whitespace
  (?P<value>.*?)      # The header's value -- *? used to
                      # lose the following trailing whitespace
  \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
print(pat.search(" xx:  yyy  ").groups())
```

Time to exercise!

7.7.11 urllib.request

Finally, to interact with remote systems, we have the urllib.request[18] library.

Python

```python
import urllib.request
req = urllib.request.Request(url='https://www.ost.ch', method="GET")
with urllib.request.urlopen(req) as response:
  print(response.status)
# => 200
```

There's also the 3rd party requests[a] library, which simplifies life further.

Time to exercise!

[18]https://docs.python.org/3/library/urllib.request.html

[a]https://requests.readthedocs.io/en/master/

8. C

8.1 Introduction

The C programming language was developed in 1972 (!) to provide higher level abstractions for assembly instructions. At the time of writing - 48 years later - C interestingly is still (one of) the most widely used programming languages!

C is translated to machine language in four steps:

- preprocessing, which includes e.g. removing comments
- compiling, which translates from C to assembly
- assembling, which translates from assembly to object code
- linking, which creates a single executable file

The resulting program can be executed directly on our machines - there is no interpreter required. The disadvantage of this is that our code has to be compiled for a specific operating system (version) and cannot simply be run on another machine.

So why are we looking at such an old and rather low-level language? The reason is that when we reverse engineer a program running natively on our machines, we have a somewhat decent chance to decompile it to C, which might just save us from having to look at assembly code (well, we'll look at that next, but in real life, you may sometimes get around it).

At the same time, we're not going to look into C with as much detail as we did with other languages. The reason is simply that it's very often enough to be able to read the language and do some minor adjustments. This - and the fact that there's just so much new - is also why we're allocating a bit more time to concepts rather than method application. You will notice in the timetable that we're allocating more than 1/3 of the time to concepts rather than the usual 25%.

8.1.1 Code Repository

The code snippets for this chapter can be found here [1] . Please ask to have your user added (if you haven't already)!

[1] https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/C

8.2 Timetable

This chapter is slightly more complex conceptually, the time is allocated a bit differently between concepts and methods (practice).

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Tooling	10	15	***25***
Syntax Mapping	75	110	***185***
Total	**85**	**125**	**210**

8.3 Tooling

To translate C code to an executable file, we require a toolchain which compiles, assembles and links our source code. On Unix systems, the typical choice is `gcc`, which does all these steps and comes preinstalled on many systems.

An easy way to execute your code is to use *Code Runner* in Visual Studio Code. It calls `gcc` and executes a temporary executable file it creates.

Make sure code runner is configured to execute `gcc`, and adapt `.vscode/settings.json` if necessary:

JSON

```json
{
  "code-runner.executorMap": {
    "c": "cd $dir && gcc $fileName -o $fileNameWithoutExt && $dir$fileNameWithoutExt"
  }
}
```

To execute a program which needs to read user input, you can't use Code Runner directly (there's no way to read input in Code Runner) simply compile and run the code manually:

- go to the directory where your C source file is located
- execute `gcc $filename -o $outputName` (e.g. `gcc loops.c -o loops`)
- run the compiled program by calling `./$outputName` (e.g. `./loops`)

8.4 Syntax Mapping

8.4.1 Comments

In original C, there were only block-comments, delimited by `/*` at the start and `*/` at the end of the comment - comments thus delimited are called *c-style comments*. Only in 1999 (ok, today, that seems like ages ago), in the *C99* standard, were single line comments officially added to the syntax, allowing a comment to be started by `//` and ended by a newline (line feed) not directly preceded by a backslash (`\`, called *line-continuation character*).

C

```
/* this is a comment, now discouraged for single line comments */
// this is also a comment, now preferred for single line comments
// this comment continues on the next line \
still part of the comment - highly discouraged.
/*
 * this is a regular
 * multi-line comment
 */
```

\end{code}

The comment delimiters will not have any special meaning within quotes - they will be part of a string.

8.4.2 Program Structures

8.4.2.1 Methods

With C, we really have to look at methods first since any C program requires at least a method named `main` to run. This doesn't mean you couldn't define standalone methods (and use them later), but you won't be able to run them standalone. Let's first map our default pseudocode:

C

```
int calculateFactorial(int input) {
  // calculation
  return result;
}
```

Pseudocode

```
method calculateFactorial = (input) => {
  // calculations
  return result
}
```

- The semicolon in `return result;` is mandatory. All statements have to be terminated by a semicolon in C - this is not the case for code blocks surrounded by braces (`{` and `}`), though.
- C requires us to specify the type of the parameters!

The `main` method mentioned initially looks as follows:

C

```c
int main(int argc, char* argv[]) {
  // code
  return 0;
}
```

What does this do? The first line tells us the following:

- It's a function (if it wasn't, it would start with `void` instead of `int`) which returns an integer value.
  - The value returned by `main` is the exit code, so if everything went well, we should return zero.
- It receives two arguments, namely:
  - `argc` - **arg**ument **c**ount - this is the number of arguments the program was called with: an integer value. Includes the program name.
  - `argv` - **arg**ument **v**alues - this is the pointer to an array (we'll see what that means further below) containing the argument values as character sequences. `argv[0]` contains the name of the program called (see below to learn about arrays).

The arguments `int argc, char* argv[]` are not mandatory, you can also simply declare `int main()` if you don't require input arguments.

Time to exercise!

8.4.3 Data Types

As we already saw in methods, we have to define the data types that will be assigned to a variable or constant. The below list shows the originally available data types along with their size in memory. Why is that relevant, you may wonder? Well, C is a low level language in which you have to take care of things like memory management yourself!

Type	Aliases	Min bits	Min range	Actual bits / range
`char`		8	ASCII	`CHAR_BIT`
`signed char`		8	-127 : +127	`SCHAR_MIN` : `SCHAR_MAX`
`unsigned char`		8	0 : 255	0 : `UCHAR_MAX`
`short`	`short int`, `signed short`, `signed short int`	16	-32’767 : +32’767	`SHRT_MIN` : `SHRT_MAX`
`unsigned short`	`unsigned short int`	16	0 : +65’535	0 : `USHRT_MAX`
`int`	`signed`, `signed int`	16	-32’767 : +32’767	`INT_MIN` : `INT_MAX`
`unsigned`	`unsigned int`	16	0 : +65’535	0 : `UINT_MAX`
`long`	`long int`, `signed long`, `signed long int`	32	-2’147’483’647 : +2’147’483’647	`LONG_MIN` : `LONG_MAX`
`unsigned long`	`unsigned long int`	32	0 : 4’294’967’295	0 : `ULONG_MAX`

- Why “Min bits” / “Min range”, you may wonder? Compiler developers are allowed to use “wider” types, but are required (by the specification) to foresee at least these minima.
- The constants used in “Actual bits / range” are defined in the builtin header [a] `<limits.h>` - header files define the signatures of library functions we can use.

[a] https://en.wikipedia.org/wiki/Header_file

Floating point values represent real numbers [2]. The format allows representing very big and very small numbers using relatively few bits by representing each number in scientific notation [3], separating them into a ***significand***, a ***base*** and an ***exponent*** ($n = s \times b^e$). In computing, the base will be 2 instead of the familiar decimal 10, but to make this more easily accessible, I'll give you an example in decimal notation:

$15'843.93482 \rightarrow 1584393482 \times 10^-5$

Type	Min bits	Significant Digits	Exponent Range
`float`	32	`FLT_DIG`	`FLT_MIN_EXP` : `FLT_MAX_EXP`
`double`	64	`DBL_DIG`	`DBL_MIN_EXP` : `DBL_MAX_EXP`
`long double`	80	`LDBL_DIG`	`LDBL_MIN_EXP` : `LDBL_MAX_EXP`

> The constants used are defined in the builtin headers `<float.h>`

The C99 standard added a few more types, among them `_Bool`, which with the help of `<stdbool.h>` can be used as `bool` as well. This new type is in fact an integer type, but it conveniently stores any value other than `0` as `1`.

8.4.3.1 Strings

In many other languages, we can use single and double quotes interchangeably when we declare strings. In C, that's different:

- Single quotes can only contain one character, e.g. `'a'` or '`\n'`.
- Double quotes contain strings, e.g. "Hi!" or `"Hello, World!\n"`.

> Strings are arrays of `char` which are terminated by a Zero-Byte (`\0`). This means if we allocate space for a string manually, we always need that one more byte!

[2] https://en.wikipedia.org/wiki/Real_number
[3] https://en.wikipedia.org/wiki/Scientific_notation

8.4.4 Input and Output: Printing

C

```
printf("Hello, C\n");
```

Pseudocode

```
print("Hello, Pseudocode")
```

Writing output in C is both relatively basic and rather powerful. In short, the procedure `printf` takes as arguments a string containing placeholders and values that will be expanded in these placeholders. The way the values are formatted will depend on the placeholder. Note that if we want a line break, we need to explicitly add `\n` at the end.

The most relevant format codes are as follows:

Code	Meaning
`%c`	Character
`%s`	String (character array)
`%i`	Integer
`%li`	Long
`%o`	Octal representation
`%x`	Hexadecimal representation
`%f`	Float
`%lf`	Double
`%e`	Scientific notation
`%a`	Binary hexadecimal representation
`%%`	Literal `%` character

The `%` character can be immediately followed by an integer value and `$` (e.g. `1$`) to indicate which input argument should be expanded. `1$` means the second argument to `printf`, which is typically the first value argument after the format string. This allows us to avoid repeating the same value multiple times.

These codes can be augmented by the following flags:

Flag	Effect
`#`	`o`, `x`: prefix with “0” / “0x”; `f`, `lf`, `e`: always print decimal point
`+`	Print `+` sign for positive (signed) numbers
Space	Prepend a space for positive (signed) numbers
`0`	Pad numbers with leading zeros
`'`	Print thousands separators
`-`	Align left (see below for width)

Any non-zero positive integer number directly before the format code is interpreted as a “width”, which allows aligning numbers. This value is also relevant for zero-padding.

The last possible addition is a dot followed by an non-negative integer (e.g. `.3`) directly before the format code (and, if provided, after the width), specifying

- the decimal precision for floating point numbers
- prefixing non-floating point numbers with zeros
- truncating strings

Here are some examples that should make this easier to understand:

C

```c
#include <stdio.h>
int main() {
  char c = 67;
  printf("%1$c is\n    decimal   %1$3d,\n      octal   %1$o and\nhexadecimal %1$#.3x\n",
    c);
  /* =>
  C is
      decimal    67,
        octal   103 and
  hexadecimal 0x043
  */
  double n = 345938.2349234934;
  printf("%1$44.17lf\nwould be rounded to %2$0.17f,\nis %1$41.13e in scientific
    notation,\nand %1$#40a in binary hex\n", n, (float)n);
  /* =>
                      345938.23492349340813234
  would be rounded to 345938.25000000000000000,
  is                         3.4593823492349e+05 in scientific notation,
  and                      0x1.51d48f08fc8c5p+18 in binary hex
  */
  char* s = "Hi there!";
  printf("'%1$s', '%1$2s'", s);
  // => 'Hi there!', 'Hi'
  return 0;
}
```

The first line imports the library that makes the `printf` function available.

Time to exercise!

8.4.5 Data Structures

8.4.5.1 Constants and Variables

C allows us to declare constants, making the initially assigned value unmodifiable.

C

```
int x = 5;
const int y = 2;
```

Pseudocode

```
variable x = 5
constant y = 2
```

8.4.5.2 Arrays

By now you've probably realized that C is a rather low-level language. And we'll really see this in arrays and how we work with them.

C

```
int foo[] = { 2, 4, 9, 1, 7 };
int first = foo[0];
int len = sizeof(foo) / sizeof(foo[0]);
// not so simple in C - see below
```

Pseudocode

```
variable foo   = [2, 4, 9, 1, 7]
variable first = foo[0]
variable len   = foo.length
foo[5]         = 18
```

In C, an array is basically a number of memory locations that get reserved when it's created. By default, arrays are local to the current code block - and code called from within it - and are not safe to be accessed from outside. This means that if a method "a" calls method "b" and "b" declares an array, "a" will not be able to access this array.

To create the same structure, but make it survive the code block it's been created in, we will have to allocate (and free) memory on the *heap* [4] explicitly using `malloc` (**m**emory **alloc**ate). We will also have to manually keep track of its size, the calculation above (`sizeof(foo) / sizeof(foo[0])`) doesn't work any longer.

What does this `sizeof` operator do, anyway? It tells us how many bytes a variable uses (in memory). Once you've read the chapter about assembly, you'll hopefully be able to deduce why `sizeof(foo)` can tell us how much the entire array uses, whereas the same applied to a pointer to an array allocated using `malloc` cannot.

If we need to increase the size of such an "array", we can use `realloc` (**re-alloc**ate). We also have to be careful to release memory when we don't need it any longer using `free` - C doesn't do this for you!!

[4] https://en.wikipedia.org/wiki/Heap_%28data_structure%29

Let's look at some sample code:

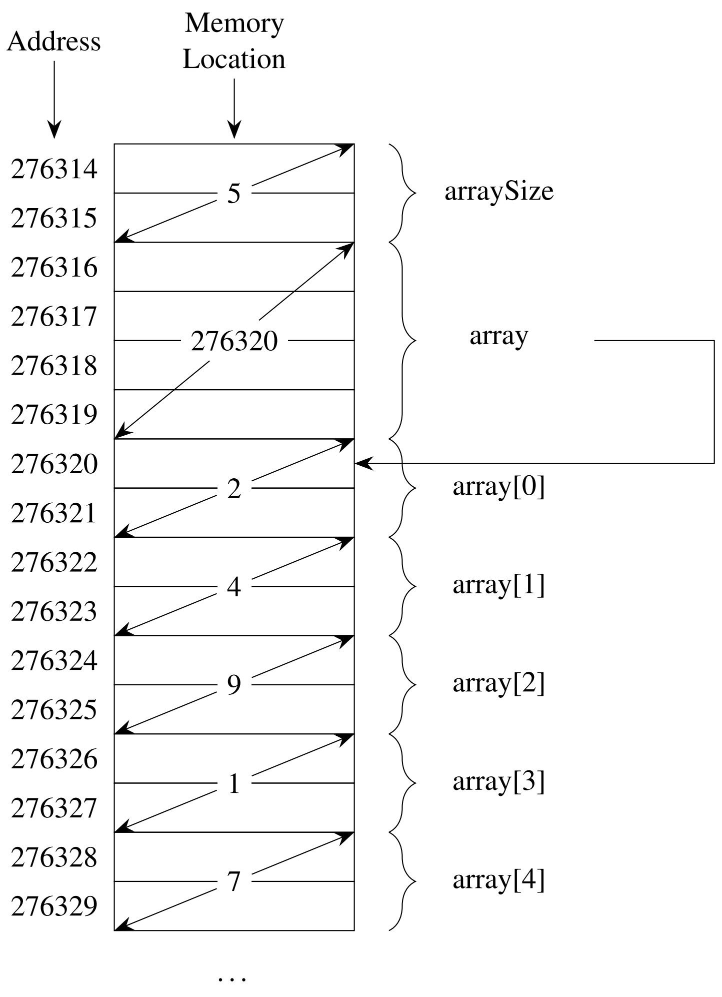
C

```
 1 #include <stdlib.h>
 2 #include <string.h>
 3 int main() {
 4   int arraySize = 5 * sizeof(int);
 5   int* array = malloc(arraySize);
 6   memcpy(array, (int []){ 2, 4, 9, 1, 7 }, arraySize);
 7   // do something with your shiny array
 8   int newArraySize = 2 * arraySize;
 9   int* newArray = realloc(array, newArraySize);
10   if (newArray) {
11     array = newArray;
12     arraySize = newArraySize;
13   } else {
14     // if memory can't be allocated, we'll have to do handle that here
15   }
16   // now we have space for 10 integer values! Do something with it...
17   free(array);
18 }
```

Ok, this must be confusing. Let's try to understand what's going on here:

- Line 1: Import the definitions of functions provided by the standard library, `stdlib`.
- Line 2: `string` provides `memcpy`, which works for any structure that has units with a size that is a multiple of 4 bytes.
- Line 4: Calculate the space taken up by five integers (in bytes)
- Line 5:
  - Declare a pointer (type `int*`, see below) to a memory location holding the first integer of the array
  - Allocate the space required for an array of integers
- Line 6: copy content from a local array to our structure
- Line 9: There are two things done in the background,
  - first, either...
    - ...allocate more memory directly following the previously allocated memory (if possible), or
    - ...allocate new memory and copy the contents of the previously allocated memory to the new location
  - second, if memory could not be allocated immediately following the original location, free up the originally allocated memory.
- Line 10ff: Check if memory could be allocated.
  - If so, the original pointer may have become invalid. Re-assign `array` to point to new memory location.
  - If not, the original array is retained and the problem has to be handled.
- Line 17: Free up memory allocated.

Unfortunately, it doesn't get easier here since we have to explain a very important low level structure: a *pointer*. Pointers contain memory addresses; they point to that memory location. We can visualize a possible memory layout in a 32-bit system:



`array` is initialized to the memory address where space is reserved for 5 integers, `276320`. We also see the first of these integers (`2`) at this memory location and the second one two bytes (the size of an integer in memory) further down.

You can get the memory address of any variable by prepending `&`, e.g. for `int foo` you can write `&foo` to get the *pointer* to `foo`. And if you store that pointer in a variable `bar` (`int* bar = &foo;`), you can get at the value at that location by prefixing a asterisk (`int xyz = *bar;`).

C

```c
int foo = 6;
int* bar = &foo;
int xyz = *bar; // contains the value 6
```

Time to exercise!

8.4.5.3 Maps

C does not know any data structure like the maps we know. There are libraries that provide this functionality, and of course you can (but probably shouldn't) write your own implementation.

8.4.6 Structs

Using a construct called *Struct*, data can also be **struct**ured in C, and memory can be allocated for an entire group of variables at once. As an example, we can declare the following *Struct*:

C

```c
struct Coordinate {
  int x;
  int y;
};
```

We can then declare a variable to be of type `Coordinate` - the nice thing is that it handles like a simple type, we don't have to explicitly allocated memory - even if we're now dealing with multiple values!

C

```c
struct Coordinate point1;
struct Coordinate point2;
```

We can now access the properties of a struct using a dot (`.`) between variable name of the struct and property name: `point1.x` will access the int value `x` within the struct instance named `point1`.

That's a bit tedious though, having to each time write `struct Coordinate`, isn't it? There's a shortcut! If we use the keyword `typedef` in our declaration, we can assign an alias which we can subsequently use:

C

```c
typedef struct Coordinate {
  int x;
  int y;
} coordinate;

coordinate point1;
coordinate point2;
```

We can even allocate arrays of structs, using the syntax we already know - we just need to remember to use `sizeof` to calculate the required memory. And there's one further goodie: if we have a pointer to a struct, we can use the arrow notation `pointPointer->x` as a shortcut for writing `(*pointPointer).x`.

C

```c
int numPoints = 10;
coordinate *points = (coordinate*) malloc(numPoints * sizeof(coordinate));

points[0]->x = 3;
points[0]->y = 7;

// Just for demonstration of equivalence, don't write inconsistent code like this!
printf("%i/%i; %i/%i", points->x, points[0].y, (*points).x, (&points[0])->y);
// => 3/7; 3/7
```

8.4.7 Input and Output: Reading

All right, now at least reading should be easy, right? If we need to read an input that has a fixed (or maximum) length, it's simple. Since that's unfortunately not the typical case, it's not.

In reality, user input is read character by character. How do we know they're done? Typically when they press ENTER. Since we're going to read input into a string, which is really a character array, we need to reserve space for these characters. And since we don't know how many there will be, we need to increase the array size as long as we don't read a newline.

We've seen above how to extend an array, and we'll see further below how to loop, so I'll just show you the parts that aren't covered elsewhere. Full code can be found here [5] .

C

```c
1 #include <stdio.h>
2 // other headers
3
4 int main() {
5   // allocate memory
6   printf("Enter your choice: ");
7   // loop
8     fgets(input, BUFFER_WIDTH, stdin);
9   return 0;
10 }
```

Pseudocode

```
1 //
2
3
4
5
6
7 print("Enter your choice: ")
8
9 constant x = read()
10
```

Let's analyze this:

- Line 8: `fgets` reads up to `BUFFER_WIDTH` characters from `stdin` into array `input`, terminates if `EOF` is encountered (typically CTRL+D)

8.4.8 Control Structures

8.4.8.1 Conditions

Finally something that works as we're used to!

C

```c
if (x < 5) {
  printf("smaller\n");
} else {
  printf("greater or equal\n");
}
```

Pseudocode

```
if x < 5 then
  print("smaller")
else
  print("greater or equal")
end if
```

[5] https://github.com/ti-ng/code-programming-languages-an-introduction/C/reading.c

8.4.8.1.1 Conditionals

C also knows conditionals (they call it *ternary operator*).

C

```c
void conditional(int x) {
  printf("%s\n", x < 5 ? "smaller" : "greater or equal");
}
```

8.4.8.2 Case Matching

C

```c
enum OBJECT_TYPE {
  PLANT, ANIMAL, GAS, LIQUID, SOLID
};
switch(objectType) {
  case PLANT:
    printf("water it\n");
    break;
  case ANIMAL:
    printf("feed it\n");
    break;
  case GAS:
    printf("inhale it\n");
    break;
  case LIQUID:
    printf("surf it\n");
    break;
  case SOLID:
    printf("decorate it\n");
    break;
  default:
    printf("stare at it\n");
}
```

Pseudocode

```
//


if objectType
  matches plant   then print("water")


  matches animal then print("feed")


  matches gas     then print("inhale")


  matches liquid then print("surf")


  matches solid   then print("decorate")


  else                 print("stare")

end if
```

In C, we can't use strings, but we have the concept of *enums* [a] . Enums basically map a name to an integer value.

[a] https://en.wikipedia.org/wiki/Enumerated_type

8.4.8.3 Loops

- A "while" loop:

C

```
int i = 0;
while (i < 5) {

  printf("%i\n", i++);
}
```

Pseudocode

```
variable i = 0;
loop
  exit if i >= 5;
  print(i++);
end loop
```

Output

```
0
1
2
3
4
```

- A "do while" loop:

C

```
int i = 5;
do {
  printf("%i\n", i++);

} while (i < 5);
```

Pseudocode

```
variable i = 5;
loop
  print(i++);
  exit if i >= 4;
end loop
```

Output

```
5
```

- A "for"-loop:

C

```
const int numbers[] = {1, 2, 3, 5};
const int LENGTH = 4;

for(int i = 0; i < LENGTH; i++) {

  printf("%i\n", numbers[i]);
}
```

Pseudocode

```
constant numbers = [1, 2, 3, 5];

variable i = 0;
loop
  exit if i >= numbers.length;
  print(numbers[i++]);
end loop
```

Output

```
1
2
3
5
```

There's actually a more clever way to loop over an array in C, which takes advantage of the fact that C knows *pointer arithmetic*. If you add one to a pointer of a given data type, the memory address pointed to will be increased by as many bytes as the data type requires. Thus, we can write the following:

C

```
1 const int numbers[] = { 1, 2, 3, 5, 8 };
2 for(int* i = (int*)numbers; i < numbers + 5; i++) {
3   printf("%i\n", *i);
4 }
```

The *type cast* (`(int *)numbers`) on line 2 is only there to avoid compiler warnings, which alert us to the fact that we're losing information about the array by assigning `numbers` to a "raw" pointer to integer values.

Time to exercise!

9. Assembly

9.1 Introduction

This will be the most low-level language we'll look at - I promise! You may be wondering why we even look at assembly in the first place? When would you ever have to write assembly code? Possibly never. You'll likely have to look at it at some point, though: when analyzing Malware running natively.

Many modern languages are interpreted, meaning we get easy access to their source code. That source code may be obfuscated, but we can directly look at what is run and can typically debug it quite simply. Other languages (such as e.g. C) are compiled to machine language - which unfortunately is not as easily understandable. If we want to see what's happening, we need to *reverse engineer* the executable instructions, which means that we create assembly code from them. Since it's not always possible to translate back to a higher level language, this is often the code we have to look at and understand.

There are different syntaxes of assembly language (like there are different higher level languages). We are going to look at the syntax used by the Netwide **Assem**bler [1] (NASM) intended to be run on 64-bit x86 architectures. The reason we choose this syntax is because it's close to what we'll see when reverse engineering code.

Our code listings will provide code examples definitely working in a recent Kali Linux, and likely in Linux systems in general. Code snippets provided in the repository (see below for the link) will usually include commented code for MacOS as well.

Assembly is translated to machine language in two steps:

- assembling, which translates from assembly to object code
- linking, which creates a single executable file

The resulting program can be executed directly on our machines - there is no interpreter required. The disadvantage of this is that our code has to be compiled for a specific operating system (OS) - or even specific OS version - and cannot simply be run on another machine.

A list of *instruction codes* can be found at the end of this chapter for your reference.

[1] https://www.nasm.us/

9.1.1 Code Repository

The code snippets for this chapter are often excerpts (and not the complete programs) - the full code can be found here [2] . If you don't yet have access, please ask to have your user added!

I've mentioned before that you should play around with the code, changing around things to see what works and what doesn't. This is more important than ever in a language such as assembly, because very often, things might not do what you expect them to at first!

9.2 Timetable

This chapter is somewhat more complex conceptually, so more time than usual is allocated to concepts.

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Tooling	15	30	***45***
Basic Concepts	45	n/a	***45***
Syntax Mapping	90	90	***180***
Instruction Codes		Reference only	
Total	**150**	**120**	**270**

9.3 Tooling

To assemble the programs we write, we need to make sure we have NASM installed. We are going to translate programs to run under UNIX, so if you're working with another platform, your easiest path forward is to use e.g. the Hacking-Lab Live CD [3] .

In a first step we have to run our assembler to translate our assembly code into object code. On most UNIX systems, this means translating to ELF [4] . To do this for a 64bit Linux platform, we can execute `nasm -f elf64 $prog.asm -o $prog.o`, e.g. `nasm -f elf64 hello.asm -o hello.o`. On a Mac, we need to use `-f macho64` instead of `-f elf64`.

In a second step, we need to bundle our object code and required libraries into an executable program. We could use `ld` to do this, but it's simpler to let `gcc` do everything that's needed: `gcc $prog.o -o $prog`, e.g. `gcc hello.o -o hello`. And that's it, we have an executable program we can run.

> If you tell GCC to keep the intermediate assembly files it generates (with extension `.s`), what you will see is *GAS* syntax. If you'd like, you can use e.g. c2nasm [a] to create NASM syntax from a C program to see how the code is translated.

[2] https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/Assembly
[3] https://livecd.hacking-lab.com/
[4] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[a] https://github.com/diogovk/c2nasm

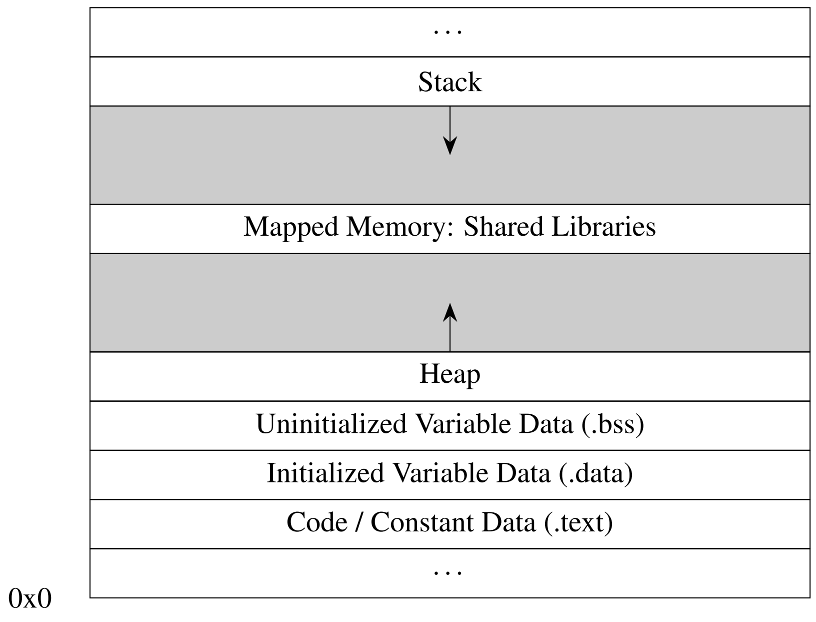
9.4 Basic Concepts

Before we get started mapping our pseudocode syntax to NASM, we should probably cover some basic concepts - without these, it'll likely be very hard (well, even harder than it probably already is) to understand what's going on.

9.4.1 Memory

Today, pretty much all memory is *virtual memory*, which means that any memory address we see is mapped by the OS to a physical memory address. This means that every program can have the same memory addresses and even if multiple programs run in parallel, they won't have any issues.

Memory for programs is allocated as shown in the following picture:


...
Stack
Mapped Memory: Shared Libraries
Heap
Uninitialized Variable Data (.bss)
Initialized Variable Data (.data)
Code / Constant Data (.text)
...
0x0

We will see most of these sections referenced in our code later, details aren't that important at this point. One main thing to remember is that the stack grows towards smaller addresses, so as over time we allocate memory on the stack, memory addresses used will become smaller and smaller.

9.4.2 CPU Registers

Our computers' CPUs work with data in *registers* [5] - any data in memory (generally) first has to be transferred into registers to be used in calculations. There is a limited amount of registers available with which we have to work. In the X86 [6] platforms for which we're going to write code, the names of the integer registers we are going to mainly use may seem a bit strange: they were meant to be used for predefined tasks.

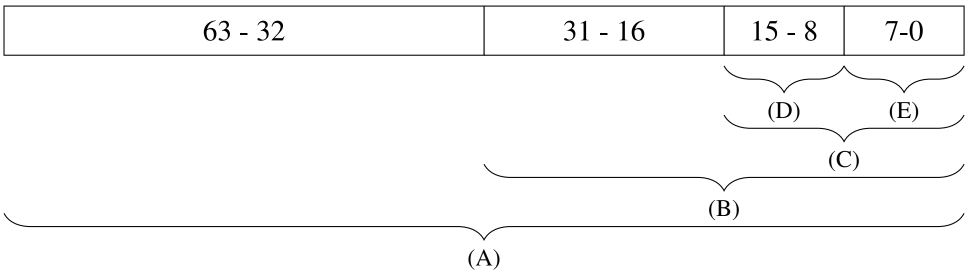
We typically access the entire register's contents, but it is important to know that we can also work with only certain areas (bytes) of them. This is mainly due to compatibility reasons (e.g. allow 32-bit code to execute on 64-bit machines), which also explains why the names of some register areas are the same as they were in previous architectures.

[5] `https://en.wikipedia.org/wiki/Processor_register`
[6] `https://en.wikipedia.org/wiki/X86`

9.4.2.1 Integer Registers

Integer registers hold whole numbers (as opposed to real numbers, which are held in floating point registers).

Let's look at the different areas of a 64-bit integer register and see what these areas are called. In order to help us with that, the following image shows the memory layout of a 64-bit register and labels the different areas from `(A)` to `(E)`. Following that, we will assign the register names for the different registers.



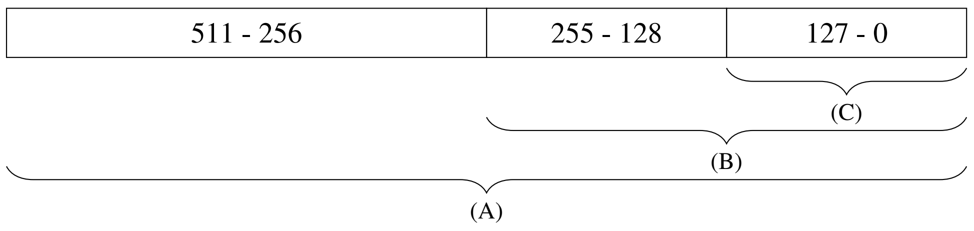
Below we see the mapping between the name of the entire register in column A and its individually addressable areas in columns B to E. As an example, `rdi`'s lower 32 bits are addressable via the name `edi`, its lowest 16 bits as `di` and it's lowest 8 bits as `dil`.

Registers `r8` to `r15` were added when the x86 architecture was extended for 64-bit processors; in 32-bit systems, only the first eight registers existed - and their main name was the one in column B.

A	B	C	D	E
`rdi`	`edi`	`di`	n/a	`dil`
`rsi`	`esi`	`si`	n/a	`sil`
`rax`	`eax`	`ax`	`ah`	`al`
`rbx`	`ebx`	`bx`	`bh`	`bl`
`rcx`	`ecx`	`cx`	`ch`	`cl`
`rdx`	`edx`	`dx`	`dh`	`dl`
`rsp`	`esp`	`sp`	n/a	`spl`
`rbp`	`ebp`	`bp`	n/a	`bpl`
`r8` - `r15`	`r8d` - `r15d`	`r8w` - `r15w`	n/a	`r8b` - `r15b`

9.4.2.2 Floating-Point Registers

For floating-point registers, there are three address areas:



The registers are very simply named, with the following prefix mapping for the areas, `MM` as infix, and numbers 0 to 15 as postfix.

- (A): `Z`, e.g. `zmm5`
- (B): `Y`, e.g. `ymm8`
- (C): `X`, e.g. `xmm1`

There are also registers `zmm16` to `zmm31`, which are not addressable through narrower areas. Additionally, the original floating-point registers `mm0` through `mm7` are still available - they are 64bits wide.

9.4.3 Command Line Arguments

When the program is executed, the command line arguments will be passed via registers `rdi` (holding the count of arguments - including the program name) and `rsi` holding the memory address of where of the program name is stored. The addresses of the actual arguments will be at `rsi + 8`, `rsi + 16`, . . .

9.5 Syntax Mapping

In general, assembly code - in the `.text` section - consists of instructions, which are generally executed linearly (one after the other), and labels, which allow non-linear code execution (labels can be "jumped" to). Memory allocation / initialization is done in the `.bss` and `.data` sections.

9.5.1 Comments

In NASM, line comments start with a semicolon (`;`). There's also a "clever" way which can be used to comment multiple lines: using macros. A macro will be processed by NASM's preprocessor, replacing all occurrences of a defined "variable" by their definition. We're not going to look at macros at this time, but suffice it to say that anything between `%ifdef $foo` and `%endif` is removed if `$foo` is not defined. An example of this is that everything between `%ifdef COMMENT` and `%endif` will be ignored (assuming you don't have a coworker who likes to pull your leg and defines `COMMENT` somewhere before your intended comment).

NASM

```
; nothing here to see!
%ifdef COMMENT
  nothing here either
  still nothing
%endif
```

9.5.2 Program Structures: Methods

Let's start by understanding a very basic program which does - nothing. It basically starts and then terminates successfully (you can imagine it's just doing `exit 0`). Here it is:

NASM

```
1 global      main
2
3 section     .text
4     main:
5         mov         rdi, 0                ; exit code 0
6         mov         rax, 0x3c             ; code for "exit" system call
7         syscall                           ; call OS
```

From the top, this is what's happening:

- Line 1: Makes the label `main` available externally.
- Line 3: Lets the assembler know the following lines are program code.
- Line 4: Declares a *label* `main`. Labels can be referenced and jumped to.
- Line 5: **Mov**es the value 0 into register `rdi`. Alternatively, this is often done by XORing its contents with themselves.
  - This is the exit code passed as argument to the system call.
- Line 6: **Mov**es the value `0x3c` (decimal 60) into register `rax` (the OS will check this register to see what to do).
  - In Linux, `0x3c` is the code used to tell the OS to terminate the (sys-)calling program.
- Line 7: Executes the system call defined in `rax`.
  - For a list of Linux system calls and which registers they take which argument from, see e.g. this unofficial source [7] .

[7] https://filippo.io/linux-syscall-table/

9.5.3 Input and Output: Printing

Our program cannot directly write to the console, but we can ask the OS to do it for us. To do so, we need to instruct `syscall`; but first, we have to initialize the appropriate registers with the arguments we need to pass in order to tell the OS what to do.

1. `rax`: we need to define what we want the OS to do for us when we issue the instruction `syscall`. The code for the "write" system call is `0x1` (or just `1`) in Linux.
2. `rdi`: the first argument for the function call provides the file pointer to `stdout` (value `1` as you may remember).
3. `rsi`: the second argument provides the memory address of the message to print
4. `rdx`: rhe third argument defines the number of characters (bytes) to write

Let's write a "Hello, World!" program to see how we call methods.

NASM

```
 1 global     main
 2
 3 section    .text
 4     print:
 5         mov       rax, 0x1
 6         mov       rdi, 0x1
 7         lea       rsi, [rel message]
 8         mov       rdx, message.len
 9         syscall
10         ret
11     main:
12         call      print
13         mov       rax, 0x3c
14         xor       rdi, rdi
15         syscall
16
17 section    .data
18     message:
19         db        "Hello, World!", 0xA
20         .len:     equ $ - message
```

Let's have a look at the things we haven't seen yet!

- Line 4: define a label `print`.
- Line 7: **l**oads the **e**ffective **a**ddress of `message` (translated from the one **rel**ative to the program start address) into `rsi`. Memory locations have to be put into brackets.
- Line 8: **mov**es the value of `message.len` into `rdx`
- Line 10: **ret**urns control to caller (jumps back)
- Line 12: **call**s the code at label `print`
- Line 17: Lets the assembler know the following lines are variables.
- Line 19: Puts the characters "H", "e", "l", ... and the newline character into memory.
  - `db` stands for **d**efine **b**ytes.
  - One can specify each character individually (comma separated), as one string, or mix, as we do here, adding the newline character individually.
  - `\n` is **NOT** equivalent to `0xA` (or `10`)!
- Line 20: calculates a constant based on the current memory address (`$`) and the one of `message` and **equ**ates (sets equal) a symbol named `message.len` to it.
  - A label starting with a dot is a "child label" to the label defined just ahead of it. It will be referenced using the combined name.

9.5.4 Program Structures: Methods (Continued)

Instead of writing `[rel ...]` for each memory address, we can simply put `default rel` at the top of our file. This way, all addresses will be considered relative - we will do this going forward.

We will also focus on the code that shows new, relevant concepts in the code listings. The full, executable programs can always be found in the repository mentioned in the introduction.

9.5.4.1 Calling Conventions

There are some conventions we should know in the world of x86 64bit assembly. Let's start with registers.

When calling a function in x86 64bit, the "fastcall" convention is used by default. This means that the first few arguments (by convention) are passed in registers. The number and order depends on the operating system:

Position	*UNIX*	*Windows*
1	`rdi`	`rcx`
2	`rsi`	`rdx`
3	`rdx`	`r8`
4	`rcx`	`r9`
5	`r8`	n/a
6	`r9`	n/a

Any additional arguments are placed on the *stack* [8] (more about this later) from right to left (the first argument not on the stack is pushed last). If the function returns a value, this value is placed in register `rax` if it's not more than 8 bytes wide. If it is, it gets placed on the stack.

When our function is called, we're free to do with these registers (and additionally with `r10` and `r11`) what we want. On the flip side, when we call another function, the same is true for them, so if we need something in these registers, we have to preserve their values. On the other hand, if we want to overwrite any of the registers `rbx`, `rbp`, and `r12` to `r15`, we need to preserve their values in order to restore them before we return control to the caller - but how? We can put values on the stack.

[8] https://en.wikipedia.org/wiki/Stack-based_memory_allocation

The following are the typical "prolog" and "epilog" of a function, preserving `rbp`, the "stack **b**ase **p**ointer".

NASM

```nasm
label:
  ; prolog:
  push    rbp        ; put the value in rbp on the stack
  mov     rbp, rsp   ; set the (stack) base pointer to the current stack pointer
  ; here goes your code
  ; epilog:
  mov     rsp, rbp   ; reset the stack pointer to the (stack) base pointer
  pop     rbp        ; restore the previous value in rbp from the stack
```

The two registers mentioned point to two different memory locations on the stack: `rbp` points to the stack's *base*. `rsp` points to the current extent of the stack. It is initially set to the base, but moves as we add more local variables to the stack.

The `push` operation does two things:

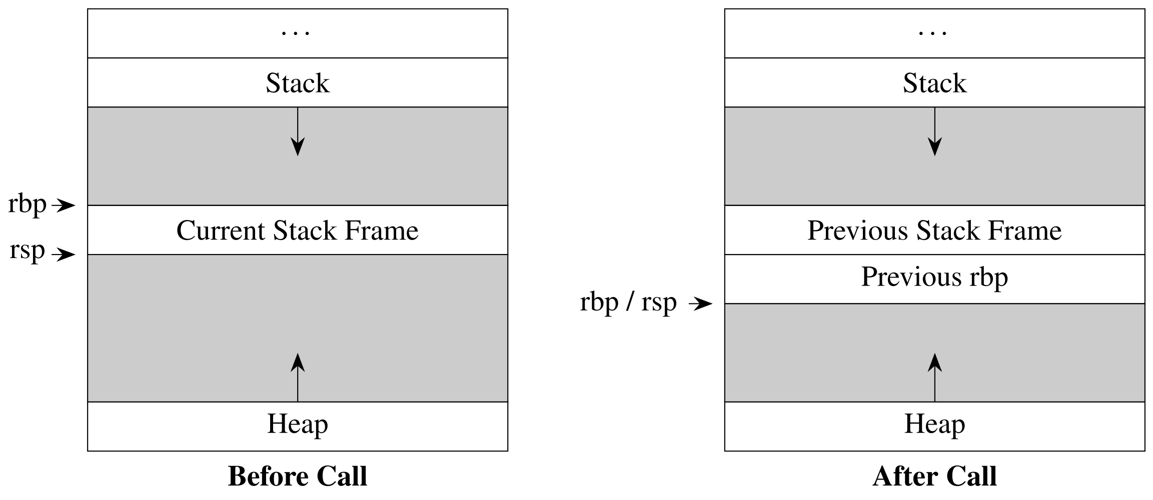
- writes the content of the stack referred to onto the stack (memory).
- decreases the stack pointer by the size of the data written - in this case 8 bytes.

The `pop` operation does the inverse:

- restores data of the size of the register referred to from the stack to that register.
- increases the stack pointer by the size of the data restored.

9.5.4.2 Stack Frames

Let's have a closer look at what happens on the stack when we call a method using the above described calling convention.



After the call, we have an new empty stack frame (`rsp` points to the same memory location as `rbp`).

9.5.4.3 Calling External Code

We can also call external methods from our assembly program - we just have to make sure they are available when we link the program. Here's an example calling the C function `printf`:

NASM

```
 1 section      .text
 2     main:
 3         lea          rdi , [fmt]         ; format for printf
 4         lea          rsi , [msg]         ; first variable argument for printf
 5         mov          rdx , qword [num]   ; second variable argument for printf
 6         mov          rax , 0             ; # of fp variables passed in variable arguments
 7         call         printf wrt ..plt    ; external function call
 8
 9 section      .data
10     msg:
11         db           "A number", 0
12     fmt:
13         db           "%s: %lld", 10, 0 ; Format string
14     num:
15         dq           1234567890123456789
```

- Line 5: To move the value from memory location `num` into register `rdx`, we have to specify the memory width. See table below for details.
- Line 6: Because `printf` can also work with floating-point arguments, we need to let it know how many of the variable arguments (`printf` can accept a variable number of arguments after the initial "format" argument) are in floating-point `xmmN` registers.
  - If our number was e.g. `1.6` stored in `xmm0`, we would have to set `rax` to 1.
- Line 7: GCC in Linux as a default builds a **P**osition **I**ndependent **E**xecutable (PIE). In order for this to work, less assumptions about memory layout can be made and an explicit expression `wrt ..plt` (**w**ith **r**espect **t**o **p**rocedure **l**inkage **t**able) has to be added to tell NASM to calculate addresses relative to the procedure linkage table which will be created by the linker.
  - An alternative to this is to tell GCC not to generate a PIE by adding the option `-no-pie`. If we do that, we don't need write `call printf wrt ..plt`, but can simply state `call printf`.
- Line 11/13: the last byte we're allocating is a zero, which `printf` uses to identify the end of a string.
- Line 15: So far, we only saw `db` used, but there are also further types. See the table below for details.

Term	Define	Access	Width
Byte	`db`	`byte`	1 byte / 8 bit
Word	`dw`	`word`	2 bytes / 16 bits
Double word	`dd`	`dword`	4 bytes / 32 bits
Quadword	`dq`	`qword`	8 bytes / 64 bits

Don't confuse the term word[a] with words in natural language. "Word" here just means "a unit of data" and has nothing to do with characters in the alphabet. Also note that the width of a word is specific to the computer architecture we're dealing with, often representing the width of integer registers. For x86 64 bit, the naming used back in 16bit days has been retained for backwards compatibility, so here, we still use "word" for 16 bits.

[a] https://en.wikipedia.org/wiki/Word_%28computer_architecture%29

9.5.5 Data Structures

9.5.5.1 Constants and Variables

There are two basic types of constants we can use:

- *Immediate* values - values that are directly coded into an instruction
- Values in memory that are in the `.text` (code) part of the file and will be write protected by modern OS'.

NASM

```
 1 section      .text
 2     main:
 3         mov          dword [message], "Hi!"          ; write "Hi!" to memory
 4         mov          byte [message + 3], 0xA         ; write 0xA to memory
 5         lea          rsi, [message]                  ; load memory address
 6         mov          rdx, 4                          ; number of characters to write
 7         call         print
 8         lea          rsi, [constmsg]
 9         mov          rdx, 15
10         call         print
11     constmsg:
12         db           "Hi right back!", 0xA
13     print:
14         ;...
15
16 section     .bss
17     message:
18         resb         10
```

The relevant lines are the following:

- Line 3: Writes the immediate value "Hi!" (three bytes) to memory location identified by label "message". Writes into four bytes of memory (a **d**ouble **word**).
- Line 4: Appends a fourth byte to an offset of three bytes from the memory location identified by label "message". Writes one byte of memory at an offset of 3 bytes from the location of `message`
- Line 12: A string declared in the code (`.text`) section.
  - Since code is meant to be unmodifiable, this memory area is protected by the OS.
- Line 14: section `.bss` is used for uninitialized memory we want to reserve
- Line 16: we reserve 10 bytes of memory - this area will be initialized to all zeros by the OS.

> An alternative to lines 4 and 5 would be to write all four bytes at once: `dword [message], 0x0A216948`. Note that the order of the bytes is reversed! Memory is written in little endian [a] format in x86 architectures. This means that the first byte is written last. To write a string to memory in the form of a hexadecimal number, we have to first invert it.

Variable values can be written to in four places:

- directly in registers
- in the data section
- on the stack
- on the heap

[a] https://en.wikipedia.org/wiki/Endianness

9.5.5.2 Arrays

There's not much more to say about arrays - they are continuous locations in memory. We've already seen how we read from and write to memory.

9.5.5.3 Maps

Assembly does not know any data structure like the maps we know. We can always write our own (wouldn't recommend it), or we can call functions in external libraries we link to. Given it's highly unlikely you'll write complex programs in assembly, we'll leave it at that.

9.5.6 Input and Output: Reading

To read data from the input, we also have two options: use a system call to read from `stdin` directly or call a library function. We'll only look at the first option here.

NASM

```
 1 section    .text
 2     main:
 3         mov         rax, 0x0
 4         mov         rdi, 0
 5         lea         rsi, [buffer]
 6         mov         rdx, buffer.len
 7         syscall
 8         mov         rax, 0x1
 9         mov         rdi, 1
10         lea         rsi, [buffer]
11         mov         rdx, buffer.len
12         syscall
13         mov         rax, 0x3c
14         xor         rdi, rdi
15         syscall
16
17 section    .bss
18     buffer:         resb 10
19         .len:       equ $ - buffer
```

What's new here?

- Line 3: code for the system call to "read" (`0x0`).
- Line 4: file descriptor for `stdin`.
- Line 17: section `.bss` is used for uninitialized memory we want to reserve - this is initialized to all zeros by the OS.
- Line 18: **res**erving 10 **b**ytes of memory to read input into.

9.5.7 Control Structures

9.5.7.1 Conditions

Whenever an instruction is executed, the CPU sets *flags*[9] based on the result of the instruction. These flags can be evaluated to decide whether or not to transfer control flow to a new address (typically by jumping to a defined label). There are instructions which are available explicitly to set these flags only:

- `cmp`: compares a register or memory location with a register, memory location or immediate value by subtracting the second operand from the first.
- `test`: calculates a bitwise AND between the operands.

Jump instructions are based on flags set by the CPU. The ones we reference are

- `ZF`: zero flag. Set when the previous calculation resulted in a zero value.
- `OF`: overflow flag. Set when the previous calculation resulted in an overflow, meaning the resulting value did not fit into the target register.
- `SF`: sign flag. Set when the previous calculation resulted in a negative number.

Based on the flags set, we can use different jump instructions. Some particularly relevant ones are:

Command	Description	ZF	OF	SF	Opposite
`je`	Jump if equal	1	?	?	`jne`
`jz`	Jump if zero	1	?	?	`jnz`
`jg`	Jump if greater	0	=SF	=OF	`jle`
`jge`	Jump if greater or equal	?	=SF	=OF	`jl`

As an example, we can use `je` to jump to a label if the previous operation was e.g. `cmp` with two equal operands. If we want to jump if the two operands were not equal, on the other hand, we use `jne`.

Because evaluation is based on flags, we may not need an explicit `cmp` or `test`: if we last used an instruction which resulted in a value of zero (e.g. decrementing a counter), we could immediately use `jz` to jump somewhere in our code.

The instruction `jmp` does not check anything, it just jumps to the label indicated.

NASM

```
compare:
    cmp         rax, 5
    jge         greaterOrEqual
smaller:
    lea         rdi, [rel msgLT]
    jmp         print
greaterOrEqual:
    lea         rdi, [rel msgGE]
print:
    xor         rax, rax
    call        _printf
```

Pseudocode

```

    if x < 5 then


      print("smaller")

    else
      print("greater or equal")
    end if


```

9 https://en.wikipedia.org/wiki/FLAGS_register

9.5.7.2 Case Matching

There is no high level concept like our case matching, but we will have a look at constructing a *jump table*, which is the basic concept which underlies case matching. We're only going to look at the simplest case, in which choices are sequential and make an example where we have cases for numbers 1, 2 or 3 and a default for any number that's greater.

NASM

```
 1 section     .text
 2     jump:
 3         cmp         rax, 3
 4         ja          dflt
 5         lea         rbx, [tab]
 6         jmp         [rbx + (rax - 1) * 8]
 7     C1:
 8         mov         rsi, 2
 9         jmp         print
10     C2:
11         mov         rsi, 3
12         jmp         print
13     C3:
14         mov         rsi, 5
15         jmp         print
16     dflt:
17         mov         rsi, 8
18     print:
19         ; ...
20
21 section    .data
22     tab:
23         dq          C1, C2, C3
```

How does this work?

- Line 3: check if the input we're matching (in `rax` from previously executed code not listed here) exceeds the maximum possible value
  - if so, jump to the default "case"
- Line 4: `ja` is the equivalent of `jg` for unsigned values
- Line 5: loads the absolute address of the *jump table* [10] into `rbx`
- Line 6: jumps to the address stored in the jump table at index `rbx - 1`
- Line 22/23: defines the jump table with the addresses of the labels `C1`, `C2`, and `C3` in subsequent 8 byte (`dq` means 8 bytes) memory locations

By loading the addresses at `tab[rax - 1]` (via `[rbx + (rax - 1) * 8]`) and continuing execution there, we effectively have implemented our case matching algorithm. `rbx` points to the base address of `tab`, and by multiplying our input value (decreased by one) by the 8 byte units we allocated memory for (using `dq` to declare quad words), we point to the address at which we put the symbols `C1` to `C3`, which contain the memory addresses of the code locations labelled.

[10] https://en.wikipedia.org/wiki/Branch_table

9.5.7.3 Loops

NASM

```
 1 initialize:
 2     xor        rcx, rcx
 3 loop:
 4     cmp        rcx, 5
 5     jge        break
 6     push       rcx
 7     lea        rdi, [rel fmt]
 8     mov        rsi, rcx
 9     xor        rax, rax
10     call       printf wrt ..plt
11     pop        rcx
12     inc        rcx
13     jmp        loop
14 break:
```

Pseudocode

```
 1 //
 2 variable i = 0;
 3 loop
 4   exit if i >= 5;
 5
 6
 7
 8
 9
10   print(i++);
11
12
13
14 end loop
```

Output

```
0
1
2
3
4
```

- Line 6: we need to preserve the value in `rcx` before calling `printf` - `printf` can (and will) overwrite the register.
- Line 11: we use `pop` to restore the value of `rcx` after `printf` has returned.
- Line 12: `inc` increases its operand by one.

There is a jump instruction we haven't looked at yet, one that specifically checks if the `rcx` register is zero and jumps if it is: `jrcxz`. This means that if we use `rcx` for loop counters, we have a shortcut instruction for our jumps.

9.6 Instruction Codes

We're not going to list all instruction codes [11] , but as a reference, some of the most often used integer register instructions are listed below. For the operands, we're going to use the following codes:

- `R`: register
- `M`: memory address
- `I`: immediate value

We're not going to specify `rel` for memory addresses - we will assume we've declared `default rel`.

9.6.1 Moving Data

- `mov` RM, RMI, e.g. `mov rbx, 15`
  - Puts the value the right operand into the left one
  - Only the left **OR** the right operand can be a memory address

9.6.2 Arithmetic

- `add` RM, RMI, e.g. `add [spi + 8], rax`
  - Adds up the left and right operand and puts the result into the left operand
- `inc` RM, e.g. `inc rcx`
  - Increases the operand by one
- `sub` RM, RMI, e.g. `sub rbx, rax`
  - Subtracts the right from the left operand and puts the result into the left operand
- `dec` RM, e.g. `dec rcx`
  - Decreases the operand by one
- `imul` R, RMI, e.g. `imul rbx, ecx`
  - Multiplies the two operands.
  - Note that this generally only makes sense if the value in `rbx` is equal to the value of `ebx` (or if you otherwise know that the result will fit into 64 bits). Overflow will be truncated when storing into the first operand.
  - There is a one-operand version, which for e.g. `imul rbx` multiplies `rbx` with `rax` and stores the result in `rdx:rax`, meaning the higher order bits will be in `rdx` and the lower order bits in `rax`. This form also exists for unsigned multiplication with the instruction `mul`.
- `idiv` RM, e.g. `idiv rbx`
  - Will divide the implicit operand `rdx:rax` (higher order bits in `rdx`, lower order bits in `rax`) by the operand given.
  - The quotient will be saved in `rax`, the remainder in `rdx` such that `rax` * operand + `rdx` is equal to the original `rdx:rax`.
  - For narrower operands (e.g. `ebx`), the matching implicit operand will be chosen (e.g `edx:eax`). For `el` as an operand, the value in `ax` would be divided, not `dl:al`.
  - The unsigned version of `idiv` is `div`.

[11] https://en.wikipedia.org/wiki/X86_instruction_listings

9.6.3 Logic

- `cmp` RM, RMI, e.g. `cmp rbx, rdi`
  - Subtracts the second operand from the first and throws the result away but sets the resulting flags.
  - Only the left **OR** the right operand can be a memory address
- `test` RM, RI, e.g. `test rbx, rdi`
  - Calculates the bitwise `AND` between the operands and throws the result away but sets the resulting flags.
- `jmp` RMI, e.g. `jmp rbx`
  - Typically used with a label, e.g. `jmp print`.
  - The most relevant conditional jumps are listed in section "Conditions".
- `call` RMI, e.g. `call _printf`
  - Pushes the memory address of the currently executed instruction on the stack, enabling `ret`, and jumps to the given address
- `ret`
  - Returns to the location `call` was instructed from (to the immediately following instruction)
  - There is a form of the instruction which accepts an immediate operand, which is used to advance to an instruction at the given offset from the instruction following where `call` was executed from.
  - The actual instructions are `retn` (return near) and `retf` (return far), to which `ret` will be translated by the assembler based on the target system's memory model.

10. Java

10.1 Introduction

Java has been one of the top choices among programming languages for a long time now, and it's certainly still a strong contender. Importantly, most Android apps are written in Java, so you'll require some understanding of the language when you analyze mobile phone malware.

Java code is compiled to bytecode running on the Java Virtual Machine (JVM). At runtime, the bytecode is interpreted and can be further optimized by the *hot spot compiler*, turning often used code sections into native code.

New versions of the language specification are published every six months, but most of these releases are only supported and actively patched for a very short period of time. Every few releases are longer lived - **l**ong **t**erm **s**upport (LTS) - versions, which are supported for typically around nine years.

We will write code making use of *Java 11* [1] .

10.1.1 Code Repository

The code snippets for this chapter can be found here [2] . Please ask to have your user added if it isn't already!

10.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Tooling	30	60	***90***
Syntax Mapping	90	255	***345***
Language Features	45	n/a	***45***
Total	**165**	**315**	**480**

[1] `https://docs.oracle.com/javase/specs/jls/se11/html/index.html`
[2] `https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/Java`

10.3 Tooling

As a first step, let's make sure we have Java 11 installed and set as default. To do this, execute `java --version`. If it shows a version number starting with 11 and says "JDK" (**J**ava **D**evelopment **K**it) somewhere, all is well. If you've installed a later version, you're likely fine too.

If you have an older version installed (or only a JRE instead of a JDK), use your OS' package manager or go to e.g. this site [3] to install a JDK for your system manually.

Even though there are currently still better alternatives for Java development than VSCode (in my opinion), we'll stick to this environment. It allows us to quickly get started using code runner and for the small programs we're implementing it will be quite sufficient.

As just mentioned, an easy way to execute some simple lines of code is to use *Code Runner*. To configure code runner to execute simple Java code, adapt `.vscode/settings.json` to run `jshell`:

JSON

```json
{
  "code-runner.executorMap": {
    "java": "cd $dir && cat $fileName | jshell -",
  }
}
```

As soon as we get to more complex examples, it's worth installing the Language Support for Java [4] extension. This will allow you to easily run the snippets provided in the repository. To do so, simply open the "Java" directory directly in Visual Studio Code. When you open individual source files which can be executed, you'll get an inline "Run | Debug" link, which allows you to directly execute the code.

Finally, if you'd like to create an executable java class, you can compile a java source file using `javac $prog.java` and run the resulting class (`$prog.class`) file using `java $prog` (without the `.class` extension). In order for this to work, you need to either be in the directory `$file` is in (actually, the base directory, as we'll see when we look at package structures further below) or add `-cp $dir`. Since Java 11, it is also possible to directly execute `java $prog.java`, which will do the above in the background.

[3] https://adoptopenjdk.net/installation.html
[4] https://marketplace.visualstudio.com/items?itemName=redhat.java

10.3.1 Debugging

10.3.1.1 Setup

To set up debugging for Java in VSCode, we really only need to install the “Language Support for Java” extension mentioned above, and we’re ready to go! When you open individual source files which can be executed, you’ll get an inline “Run | Debug” link, which allows you to directly debug the code without any further setup steps.

10.3.1.2 Stepping Through Code

Now you’re ready to step through some code. Set a *breakpoint* by clicking left of a line number in the left margin. This defines where the code execution will be suspended. Once it does, the *Variables* panel will show the values of all variables that are part of your program’s context, and the *Call Stack* panel will display the hierarchy of methods which led to the current point in the program. You can start to step through the code using “Run > Step Over” (or “Run > Step Into” when execution is stopped at a method) to execute your code line by line. When execution is suspended inside a method. you can use “Run > Step Out” to return to the method’s caller. To resume program execution and have your program run to the next breakpoint (or until it terminates), use “Run > Continue”. To finish your debug session, choose “Run > Stop Debugging”.

You can also define a *watch expression* in the *Watch* panel. This allows you to add a Java expression which will be evaluated when code execution is suspended - very convenient if you’d like to inspect some deeply nested state!

> Breakpoints can be set to be conditional: to only suspend execution if a given condition is met. This can be useful e.g. when only the 73. loop iteration is of interest.

10.4 Syntax Mapping

10.4.1 Input and Output: Printing

To output information to the user, we'll use `System.out.print($string)` or `System.out.println($string)` - which adds a newline at the end - method. The mapping to our pseudocode is as follows:

Java

```
System.out.println("Hello, Java!");
```

Pseudocode

```
print("Hello, Pseudocode")
```

To make sure your setup is running, open the file `Printing.java` and (a) run it, (b) select only the line with the code listed here and use code runner to execute it, and (c) set a breakpoint on the line with the code listed and start debugging.

10.4.2 Comments

In Java, there are two different types of comments:

- `//` will make the rest of a line a comment
- `/*` denotes the start of a comment, whereas `*/` marks its end. Such comments can span multiple lines. These comments are often also used to temporarily comment parts of a program (while debugging).

Java

```
// Comment, moving on...
/*
 * More comment lines:
 * - one
 * - two
*/
System.out.println("// No /*comments*/ in strings" /*+ " but outside: inline"*/); // an example
```

10.4.2.1 JavaDoc

There is also a special type of comment which starts with `/**` (instead of `/*`), defining JavaDoc [5] , which is the primary means of documenting (library) code. This leads to the comment being parsed and - if it is syntactically correct - translated to HTML. JavaDoc can be applied to classes, interfaces, enums, fields, and methods (see below).

[5] https://en.wikipedia.org/wiki/Javadoc

10.4.3 Data Types

Java has 8 built-in *primitive* (non-object) data types, which each have a built-in corresponding "wrapper" type:

Type	Size	Wrapper	Description	Default
`byte`	1 byte	`Byte`	Whole numbers between -128 and 127	`0`
`short`	2 bytes	`Short`	Whole numbers between -2^{15} and $2^{15} - 1$	`0`
`int`	4 bytes	`Integer`	Whole numbers between -2^{31} and $2^{31} - 1$	`0`
`long`	8 bytes	`Long`	Whole numbers between -2^{63} and $2^{63} - 1$	`0L`
`float`	4 bytes	`Float`	Single precision floating point number	`0.0F`
`double`	8 bytes	`Double`	Double precision floating point number	`0.0D`
`boolean`	1 bit	`Boolean`	Either `true` or `false`	`false`
`char`	2 bytes	`Character`	Unicode character between `'\u0000'` and `'\uffff'`	`'\u0000'`

Wrapper types are the object oriented representation of the primitive types. They are used with structures (such as `Collections`s - see below) which reference objects. It's possible to assign primitive types to their wrappers (and vice versa) without any special operations. This feature is called *boxing* (primitive to wrapper) and unboxing (wrapper to primitive).

Have you noticed the `L`, `F` and `D` suffixes to `0` and `0.0`? They're used e.g. in calculations to force a specific type of primitive where another one would otherwise be used. As an example, if we were to divide `1` by `2` and assign the result to a variable, we would get `0` - because we're not using floating point arithmetic. We can fix this by computing `1.0 / 2` or `1D / 2`. If we want to explicitly only use the shorter `float` type, we would have to add the `F` suffix even if we used `1.0` - otherwise a double would be used. The same is true for integers, if we want to explicitly use the wider `long` format, we have to add the suffix `L`.

The official specification can be found here [a]

Further relevant built-in data types are:

- `Object` - the implicit base type of every class (see below), with default value `null`.
- `String` - a sequence of characters.
  - Strings are enclosed in double quotes, while chars are enclosed in single quotes.
  - Double quotes within a string have to be escaped using a backslash: `\"`.

[a] https://docs.oracle.com/javase/specs/jls/se11/html/jls-4.html#jls-4.2

10.4.3.1 Operators

Here is a table of the most relevant operators. We use “number” to subsume “byte”, “short”, “int”, “long”, “float”, and “double”.

Operator	Operand 1	Operand 2	Operation	Example
`=`	any	any	Assignment	`a = 3`
`==`	any	any	Equality check	`a == 4`
`+`	any	String	Concatenation	`"Hello " + "you"` → `"Hello you"`
`+`	String	any	Concatenation	`"It's " + 42` → `"It's 42"`
`+`	number	number	Addition	`2 + 4` → `6`
`-`	number	number	Subtraction	`5 - 3` → `2`
`++`	number		Incrementation	`5++` → `6`
`+=`	number	number	In-place addition	`a += 3` → `a = a + 3`
`--`	number		Decrementation	`5--` → `4`
`-=`	number	number	In-place subtraction	`a -= 7` → `a = a - 7`
`%`	number	number	Modulo	`11 % 2` → `1`
`>`	number	number	Greater than	`a > b`
`>=`	number	number	Greater or equal than	`a >= b`
`<`	number	number	Smaller than	`a < -1`
`<=`	number	number	Smaller or equal than	`a <= c`
`&&`	boolean	boolean	`AND` / $\wedge$	`a && b`
`||`	boolean	boolean	`OR` / $\vee$	`a || b`
`!`	boolean		`NOT` / $\neg$	`!a`

10.4.4 Data Structures

10.4.4.1 Constants and Variables

Java is a strongly typed language, so we have to declare variables and constants along with their types. Where other languages use some derivation of the word "constant", Java chooses to declare constant variables as "final" (not to be modified).

Java

```java
int x = 5;
final int y = 2;
```

Pseudocode

```
variable x = 5
constant y = 2
```

Time to exercise!

10.4.4.2 Arrays

Arrays in Java are low-level structures, they don't have methods to adjust their size or to add elements. On the plus side, they are very efficient if their size stays constant.

Java

```java
final int[] input = {2, 4, 9, 1, 7};
final int first    = input[0];
final int len      = input.length;
input[1]           = 5;
// => [2, 5, 9, 1, 7]
// does not exist

```

Pseudocode

```
input          = [2, 4, 9, 1, 7]
constant first = input[0]
constant len   = input.length
input[1]       = 5;
// => [2, 5, 9, 1, 7]
input[5]       = 18
// => [2, 5, 9, 1, 7, 18]
```

- Even though the array is declared to be final, that doesn't mean no elements can be changed. It only means that the constant itself cannot be reassigned!
- If we try to simply print an array, we won't get what we'd like. To get what we want, we can e.g. use `java.util.Arrays.toString(...)`.

Most people don't work with arrays directly, but rather choose to use higher level types such as `Lists`. There are many different types of `Lists` (what all `Lists` have in common is that they are ordered). The type that is closest to the behavior of an array is `ArrayList`, which is backed by an array, but offers functionality like automatically resizing when elements are added. A major difference is that `ArrayList` cannot hold *primitive types* (scalars) like `ints`, but only their object type equivalents, `Integers`.

Using an `ArrayList`, we have more functionality available:

Java

```java
List<Integer> foo = new ArrayList<>(List.of(2, 4, 9, 1, 7));
Integer first = foo.get(0);      // => 2
int len = foo.size();            // => 5
foo.add(18);                     // => [2, 4, 9, 1, 7, 18]
foo.addAll(List.of(4, 8));       // => [2, 4, 9, 1, 7, 18, 4, 8]
foo.remove(2);                   // => [2, 4, 1, 7, 18, 4, 8]
foo.remove(Integer.valueOf(4)); // => [2, 1, 7, 18, 4, 8]
```

The first line uses syntax we haven't encountered yet (we will see more detail about it further below):

- `List<Integer>` declares a new variable of type `List` to hold items of type `Integer`.
  - `ArrayList` is a subtype of `List`. It's more specific, which is why we can assign it to a variable of type `List`. See below for classes and inheritance.
  - We're not going to look at generics [6] here, but if you want to dive deeper, that's a topic to look at ;-)
- `List.of(...)` creates a new immutable list containing the given items.
- `new ArrayList(...)` create a new instance of the class `ArrayList` from the given argument. This creates a mutable list from the immutable one passed as parameter, enabling us to change its contents on the following lines.

> Note that `remove(...)` with an argument of type `int` will remove the item at the *index* the argument specifies, while `remove(...)` which is passed an argument of the type corresponding to the type of the list's items removes the first item found that is equal to the argument.

Time to exercise!

10.4.4.3 Sets

Sets are collections that contain *unique* items - there cannot be any duplicates. They are not ordered, so there is no way to access individual items, only to loop over them.

Java

```java
Set<String> foods = Set.of("blueberry", "flour", "egg", "sugar", "butter");
```

> The `.of(...)` methods create unmodifiable collections. To get a modifiable collection, their results have to be used to create another instance.

[6] https://en.wikipedia.org/wiki/Generics_in_Java

10.4.4.4 Maps

Java

```java
Map<String, Integer> shapeCorners =
  new HashMap<>(Map.of(
    "Triangle",  3,
    "Rectangle", 4,
    "Pentagon",  5
  ))
;
System.out.println(
  shapeCorners.get("Rectangle")
);
int x = shapeCorners.size(); // => 3
Set<String> keySet =
  shapeCorners.keySet();
// => [Triangle, Rectangle, Pentagon]
shapeCorners.put("Hexagon", 6);
Collection<Integer> values =
  shapeCorners.values()
;// => [6, 3, 4, 5]
shapeCorners.remove("Rectangle");
Set<Entry<String, Integer>> entrySet =
  shapeCorners.entrySet()
;// [Hexagon=6, Triangle=3, Pentagon=5]
```

Pseudocode

```
constant shapeCorners = {

  "Triangle":  3,
  "Rectangle": 4,
  "Pentagon":  5,
}

print(shapeCorners["Rectangle"])


// does not exist in our pseudocode



// does not exist in our pseudocode


// does not exist in our pseudocode
// does not exist in our pseudocode


// does not exist in our pseudocode
```

The methods we don't know from our pseudocode do the following:

- `size()` simply returns the number of entries in the map.
- `keySet()` returns a `Set` containing all the keys.
- `values()` returns a `Collection` of all values. A collection is a super type (see below for details of what that means) of e.g. `List` and `Set`.
- `entrySet()` returns a `Set` containing items of type `Entry` (which is a class - we'll learn about that below).

Time to exercise!

10.4.5 Input and Output: Reading

The simplest way - which doesn't require any additional libraries - is to use `System.console().readLine()`. There are other ways that are easier to test (because they don't require an actual console to be present).

Java

```java
System.out.println("Choose wisely: ");
String x = System.console().readLine();
```

Pseudocode

```
print("Choose wisely: ")
variable x = read()
```

10.4.6 Program Structures

10.4.6.1 Methods

Methods can either be procedures, which are declared to return `void` (but don't really return anything), or functions, which return any other data type.

Java

```java
int calculateFactorial(int input) {
  // calculations
  return result;
}
```

Pseudocode

```
method calculateFactorial = (input) => {
  // calculations
  return result
}
```

10.4.6.1.1 Overloading

In Java, there can be multiple methods with the same name in the same class. The decision which one to execute for a given method call is taken based on the static type of the variable passed. As an example, consider the following code:

Java

```java
private void guru(Object n) {
  System.out.println("Can change it? " + n);
}
private void guru(String n) {
  System.out.println("Cannot change it? - " + n);
}
String x = "Then why worry?";
Object y = x;
guru(x); // => Cannot change it? Then why worry?
guru(y); // => Can change it? Then why worry?
```

There are a few things to notice here:

- Variables `x` and `y` refer to exactly the same `String`.
- `String` (like every class in Java) is a subtype of `Object`, which is why we can assign `x` to `y`.
- The decision which method to call is not taken based on the type of the variable's *value*, but based on the variable's **declared** type!

10.4.6.2 Interfaces & Classes

Java is an object oriented programming language, and methods can only exist within interfaces or classes. *Interfaces* define a contract which defines what can be expected in terms of functionality from a class that implements them (stating that it conforms to the contract defined by the interface). *Classes* specify a blueprint for the creation of *object instances*, which represent real world things (e.g. cars or people). Very basically, classes define the data relevant to objects (e.g. horse powers for cars, their name or birth date for people) and methods (e.g. “marry”, defining a family name) to modify this data (an object’s *state*) or to perform calculations based on it as a response to messages passed between objects.

A class that represents a runnable program has a `main` method, which accepts an array of strings:

Java

```java
1 public class App {
2   public static void main(String[] args) {
3     // ...
4   }
5 }
```

- Line 1: a `public` class has to be declared in a file that has the same name as the class. So `App` would have to be declared in `App.java`.
- Line 2: the keyword `static` means the method it precedes is a class-wide method, which means it can be called on the class itself - without requiring an instance of the class.

Interfaces implicitly set some modifiers:

- methods are always `public`
  - non-`static` methods are cannot be `final`.
  - `static` methods are always `final`.
- properties are always `public`, `static`, and `final` (they are constants).

Classes have *constructors*, which are the methods called when a class is instantiated. They have to be named the same as the class they're declared in and they don't declare any return value. If we don't declare an explicit constructor (and only then), there is an implicit *no-args* constructor, a constructor which will be called when we create a class without providing any arguments.

Java

```java
 1 public class Foo {
 2   public static void main(String[] args) {
 3     Bar.foo();
 4     Bar bar = new Bar("Bar Barians");
 5     bar.baz();
 6   }
 7 }
 8
 9 class Bar {
10   private String name;
11
12   public Bar(String name) {
13     System.out.println("Creating bar \"" + name + "\"");
14     this.name = name;
15   }
16
17   public static void foo() {
18     System.out.println("Let's have some fun!");
19   }
20
21   public final void baz() {
22     System.out.println("Welcome to " + this.name);
23   }
24 }
```

- Line 14: assigns the parameter `name` to the field `name`. The keyword `this` refers to the object instance and is there for disambiguation between field and parameter.
- Line 17:
  - the `public` modifier means that this method can be called from any other object. Alternatives are:
    - `private`: only the object itself (and objects which are instances of classes defined within the object's class) can call the method.
    - no modifier, meaning "package private": only objects which are instances of classes in the same package (see below) can call the the method.
- Line 21: the `final` modifier means that the method cannot be overridden in subclasses (see below).

10.4.6.2.1 Equals

The operator `==` works well for primitive types, but it doesn't do what you'd expect for objects (instances): it only returns true for **identical** objects, but not for ones that are simply **equal**. To address this, there is the method `equals` defined on the basic `Object` class, which can be overridden (redefined) in each class defined. As an example, if we have a class `2dPoint` with coordinates represented as `x` and `y`, we could implement `equals` as follows:

Java

```java
class Point {
  private int x;
  private int y;
  // ...
  public equals(Object other) {
    return other != null
        && getClass() != other.getClass())
        && this.x == ((Point)other).x
        && this.y == ((Point)other).y
    ;
  }
}
```

The `equals` method is what is called e.g. by the `remove` method we saw for `List` when comparing items to the argument defining which item to remove.

`((Point)other)` casts [7] `other` to type `Point` - this is safe because we've checked that they are of the same `Class`.

10.4.6.2.2 hashCode

One often overlooked method is `hashCode`. It defines e.g. which *bucket* of a hash table [8] an object goes into. Due to this, whenever `equals` is overridden, `hashCode` should equally be overridden to makes sure that object which are `equal` also go into the same bucket. If this is not done correctly, duplicate objects may end up being inserted into e.g. `Set`s.

10.4.6.2.3 toString

When we print an object to the console or concatenate an object with a string, the `toString` method of the object is called in the background. Since the default implementation of `toString` defined in object is `getClass().getName() + "@" + Integer.toHexString(hashCode())`, it often makes sense to override `toString`, e.g. to provide helpful debug output.

10.4.6.3 Inheritance

In Java, every class can inherit at most from one other class. This greatly simplifies matters because there's no need to resolve any conflicts between methods with the same signature defined in two different parent classes - there can only be one! Of course that presents another problem: how are we going to model the fact that a Centaur is both a Human and a Horse (or at least a bit of both)? This is where *Interfaces* come in: they only declare the existence of a method, but don't implement it (at least that's how it originally was). Interfaces can be *implemented* by classes and *extended* by interfaces. Classes can *extend* other classes.

[7] https://en.wikipedia.org/wiki/Type_casting_%28computer_programming%29
[8] https://en.wikipedia.org/wiki/Hash_table

Interfaces have gained the ability to provide a default implementation of a method. Conflict resolution is very simple:

- methods in extended classes have precedence before those inherited from interfaces
- if multiple interfaces define the same method, a class inheriting from them has to explicitly implement the method.

Java

```
interface Horse {
  static final int NUM_LEGS = 4;

  default int numLegs() {
    return NUM_LEGS;
  }

  default String greeting() {
    return "whinny";
  }
}

interface Human {
  static final int NUM_LEGS = 2;

  default int numLegs() {
    return NUM_LEGS;
  }

  default String greeting() {
    return "Hi, there!";
  }
}

class Centaur implements Human, Horse {
  private String name;

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

  public int numLegs() {
    return Horse.super.numLegs();
  }

  public String greeting() {
    return String.format("Hello, I'm %s", name);
  }
}

public class Inheritance {
  public static void main(String[] args) {
    Centaur tom = new Centaur("Tom Dooley");
    System.out.println(String.format(
      "%s, I'm a %s. I have %d legs.",
      tom.greeting(),
      tom.getClass().getSimpleName(),
      tom.numLegs()
    )); // => Hello, I'm Tom Dooley, I'm a Centaur. I have 4 legs.
  }
}
```

So let's look into this code in more detail:

- Lines 2 / 14: declare a constant visible across all classes.
- Lines 4, 8, 16, 20: declare default implementations of a method. If an implementing class doesn't specify them, this implementation will be used.
- Line 25: `Centaur` implements both `Human` and `Horse`.
- Line 26: declares a `String` *field* which will only be visible within each instance of class `Centaur`.
  - "Field" is the name for a variable that lives within an object instance.
- Line 28: the *constructor* for class `Centaur`. Its return value is implicitly an object of type `Centaur`.
- Lines 32 / 36: we need to explicitly define the methods inherited from both interfaces implemented.
- Line 33: to call the method `numLegs` inherited from `Human`, we can use `Human.super.numLegs()`. `super` refers to the direct parent class or interface.
- Line 37: we're not reusing any existing method implementation, but defining our own one.
- Line 44: `String.format` offers an interface very similar to C's `print` method.
- Line 45: We can't use `%i`, but have to use the more specific `%d` instead - it doesn't make a difference for output, though.
- Line 47: each object can tell us which class it was created from using the method `getClass()`. Calling `getName()` on the class returns only its name.

> Re-defining a method inherited from a parent interface or class is called *overriding* the method. The decision which method to call is taken at runtime based on the instance assigned to the variable it is called on! - this is called polymorphism [a]

Time to exercise!

[a] https://en.wikipedia.org/wiki/Polymorphism_%28computer_science%29

10.4.6.4 Packages

Every class lives inside a package, and the location of the file the class is declared in must correspond to this package. There are two options:

- there isn't an explicitly declared package: the class implicitly is part of the `default` package and must live in the root of the "classpath".
- there is an explicit declaration of the form `package foo.bar.my.pkg`, which corresponds to the file living in `$root/foo/bar/my/pkg`.
  - Convention dictates that the package reads like a reverse URL, e.g. `com.mycompany.mymodule.mycomponent`.

10.4.6.5 Imports

When we want to use classes defined in another source file, we need to declare our intention explicitly. We do that by importing the *fully qualified* name of the class: the name prefixed with the package they're defined in. The only exception are built-in classes: those defined in `java.lang`, which do not need to be imported explicitly.

We can also import all the classes defined in a package by using the wildcard (`*`) notation, e.g. `import java.util.*`. to import all classes in package `java.util`[9] .

Here's an example we've encountered before:

Java

```java
import java.util.HashMap;
import java.util.Map;

public class Maps {
  public static void main(String[] args) {
    Map<String, Integer> shapeCorners = new HashMap<>(Map.of(
      "Triangle",  3,
      "Rectangle", 4,
      "Pentagon",  5
    ));
    System.out.println(shapeCorners.keySet()); // => [Triangle, Rectangle, Pentagon]
  }
}
```

Time to exercise!

[9] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-summary.html

10.4.7 Control Structures

10.4.7.1 Conditions

In Java, conditions are expressed as follows:

Java

```java
if (x < 5) {
  System.out.println("smaller");
} else {
  System.out.println("greater or equal");
}
```

Pseudocode

```
if x < 5 then
  print("smaller")
else
  print("greater or equal")
end if
```

Time to exercise!

10.4.7.1.1 Conditionals

To simplify conditionally assigning values or passing parameters, we can use conditionals, also called *ternaries* in Java.

Java

```java
System.out.println(x % 2 == 0 ? "How considerate!" : "Oh well...");
```

10.4.7.2 Case Matching

Java

```java
switch(objectType) {
  case "plant":
    // water it
    break;
  case "animal":
    // feed it
    break;
  case "gas":
    // inhale it
    break;
  case "liquid":
    // surf it
    break;
  case "solid":
    // decorate it
    break;
  default:
    // stare at it
}
```

Pseudocode

```
if objectType
  matches plant   then // water it


  matches animal  then // feed it


  matches gas     then // inhale it


  matches liquid  then // surf it


  matches solid   then // decorate it


else                   // stare at it

endif
```

- The value provided to `switch` can either be a numeric type (primitive or wrapper), an `enum` (see below) or a `String`.
- Without the `break` statement, the next `case` would also be executed. This is called “fall-through”.

Time to exercise!

10.4.7.3 Loops

In Java, there are the same basic types of loops as in many other languages:

- A “while” loop:

Java

```java
int i = 4;
while (i < 4) {

  System.out.println(i++);
}
```

Pseudocode

```
variable i = 4;
loop
  exit if i >= 4;
  print(i++);
end loop
```

- A “do while” loop:

Java

```java
int i = 4;
do {
  System.out.println(i++);

} while (i < 4);
```

Pseudocode

```
variable i = 4;
loop
  print(i++);
  exit if i >= 4;
end loop
```

- A “for”-loop:

Java

```java
for (int i = 4; i < 4; i++) {


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

Pseudocode

```
variable i = 4;
loop
  exit if i >= 4;
  print(i++);
end loop
```

There are two additional kinds of loops, though:

- *Enhanced* for-loops for arrays and `Collection`s:

Java

```java
int[] intArr = { 1, 2, 3, 5, 8 };
for (int i : intArr) {
  System.out.println(i);
}
```

- “forEach” in Iterables:

Java

```java
List.of("I", "am", "happy").forEach(System.out::println);
```

Writing `System.out::println` is equivalent to using a lambda[a] expression explicitly: `name -> System.out.print`

Time to exercise!

[a]https://en.wikipedia.org/wiki/Lambda_%28programming%29

10.5 Language Features

10.5.1 Enums

Enums define all possible instances for a given type. In the simplest case, that's basically just names. As an example, we can define the "object type" in our case matching example as follows:

Java

```java
public enum ObjectType {
  PLANT, ANIMAL, GAS, LIQUID, SOLID
}
```

Enums can be used to implement more complex things as well and are very useful in any case where the complete set of instances are known when writing the code. A simple example is a state machine such as the following:

Java

```java
 1 enum Hacktivity {
 2   HACK(null), SLEEP(HACK), EAT(SLEEP);
 3
 4   private final Hacktivity next;
 5   private Hacktivity(Hacktivity next) {
 6     this.next = next;
 7   }
 8
 9   public Hacktivity next() {
10     return next == null ? EAT : next;
11   }
12 }
```

We can't pass `EAT` as an argument to `HACK` because we can only refer to values that have already been defined, which is why we have to pass `null` and treat the case specifically in line 10.

10.5.2 Streams

Streams are the kid who was late to the party and forgot to dress up properly. They were introduced to add functional methods like `map`, `filter` and `reduce`. Unfortunately, language designers were keen to allow old programs to run in the new environment, and somehow (in my humble opinion) didn't think it through. This led to an overly complex syntax, requiring us to call `.stream()` on collections / iterables to create a *stream* on which we can call functional methods, and finally `.collect(...)` to translate back to a collection. Of course there are more favorable views of their efforts...

The following snippet maps a list of integers to their squares:

Java

```java
List.of(0, 1, 2, 3, 4, 5)
  .stream()
  .map(e -> e * e)
  .collect(Collectors.toList())
; // => [0, 1, 4, 9, 16, 25]
```

We can also apply multiple operations to directly sum up the squares:

Java

```java
List.of(0, 1, 2, 3, 4, 5)
  .stream()
  .map(e -> e * e)
  .reduce((previous, item) -> previous + item)
  .orElse(-1)
; // => 55
```

Reduce returns an `Optional`. Using `orElse`, we can return a default value if the optional value is `null`.

10.5.3 Exceptions

Sometimes things can work in ways that we haven't foreseen - or in ways we foresee, but can't (or don't want to) handle. In Java, we can work with exceptions in two basic ways: we can *catch* and handle them, or we can declare our method to *throw* them.

Let's start with the latter one:

Java

```
/**
 * @throws IllegalArgumentException for negative numbers.
 */
public void foo(int x) throws IllegalArgumentException {
  //...
}
```

The above method declares that it can throw an `IllegalArgumentException`. It also has the decency to add JavaDoc telling us in which case it will do that.

To handle an exception ourselves, we can write code like the following:

Java

```
 1 try {
 2   throw new UnsupportedOperationException("nothing here"); // just for demonstration
 3   // throw new NumberFormatException();
 4 } catch (UnsupportedOperationException e) {
 5   System.out.println("Not implemented: " + e.getMessage());
 6 } catch (NumberFormatException | ArithmeticException e) {
 7   System.out.println("Get your numbers straight!");
 8 } finally {
 9   System.out.println("This runs in any case!");
10 }
```

This is what's happening:

- Line 1: the following code block is considered to potentially raise an exception.
- Line 2: we manually raise an exception - this would usually happen in some other code we call.
- Line 4: "catches" an exception of type `UnsupportedOperationException` and assigns the exception instance to variable `e`.
- Line 5: prints a message including the text passed to the exception's constructor.
- Line 6: "catches" an exception of either type `NumberFormatException` or `ArithmeticException` (which of course will never happen in this case) and assigns the exception instance to variable `e`.
- Line 9: code in the `finally` blick will always be executed, regardless of whether or not an exception was raised - regardless of whether or not it would have been caught.

10.5.4 Reflection

We saw earlier that methods defined as `private` cannot be accessed from outside the class defining them. This is only partially true: they can be turned from `private` to `public` via *reflection* [10] (which can also do a lot more). Here's an example:

Java

```java
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Reflection {
  public static void main(String[] args) {
    try {
      Method m = Protected.class.getDeclaredMethod("print");
      m.setAccessible(true);
      m.invoke(null);
    } catch (Exception e) {
      // don't worry, it'll work ;-)
    }
  }
}

class Protected {
  private static void print() {
    System.out.println("How did you find me?");
  }
}
```

[10] https://en.wikipedia.org/wiki/Reflective_programming

11. VBA

11.1 Introduction

The reason we're looking at VBA (Visual Basic for Applications) is that office documents are one of the most common ways malware is introduced on someone's computer. There is another reason, though: Excel is one of the simplest ways to help us with automating certain tasks, e.g. to create similar commands based on some differing values. Unfortunately, due to the lack of some functionality like e.g. local variables, Excel formulas tend to become very complicated rather quickly even though what we want to do is really simple.

VBA can help us simplify some tasks and calculations and make our life more bearable... We're going to use code that works with VBA 7.1.

11.1.1 Code Repository

As always, the code snippets can be found here [1] . Please ask to have your user added (if it isn't already)!

11.2 Timetable

We will divide our time up a bit differently in this chapter with the goal of having you finish the book with something you can use and maybe even impress your coworkers (and bosses ;-)) with: our journeyman's piece will give you a starting point allowing you to more efficiently work with large Excel spreadsheets containing lots of formulas. This is why we will spend more time on the journeyman's piece and less time on individual exercises.

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Tooling	15	60	***45***
Syntax Mapping	105	300	***435***
Total	**120**	**360**	**480**

[1] https://github.com/ti-ng/code-programming-languages-an-introduction/tree/master/VBA

11.3 Tooling

In order to assure we work with as similar an environment as possible, this chapter is going to be based on *Microsoft Excel*. There are still some differences between the versions for different operating systems, but these should not impact us too much. If you don't yet have a version of Excel installed, you can get it via your school account:

1. Log in to office.com [2] using your school credentials.
2. Choose "Install Office" to download the installer package.
3. Follow the instructions shown.

To get started, open Excel and in "File" > "Options" > "Customize Ribbon" (or "Preferences" > "Ribbon & Toolbar") under "Customize the Ribbon" > "Main Tabs", enable the "Developer" tab. You will now see the "Developer" tab, from where you can click on "Visual Basic" to open the VBA editor, where you can define your macros.

To execute any of these macros during development, you can simply press the "Play" (▶) button. For "production", you can e.g. add a button to your *Excel* sheet (from the "Developer" Ribbon) and connect it to a macro.

If you would like to use the provided spreadsheet (in the code repository) to import newer versions of the provided code snippets over time (in the form of `.bas` files), you will need to make two adjustments to your setup:

- While viewing the workbook, in "Excel" > "Preferences" > "Security" > "Developer Macro Settings", enable "Trust access to the VBA project object model"
- While in the macro editor, in "Tools" > "References" enable "Microsoft Visual Basic for Applications Extensibility"

Also note that the first line of each snippet is added by *Excel* when exported. If you use "File" > "Import File", the line will disappear and the imported module will be named correctly. Thus, if you copy/paste the content, you will have to manually delete the first line.

> Note that if you're trying to use scripts in the Web version of Excel, you won't be able to work with VBA: the language used there is TypeScript. To work with VBA, use the desktop versions of Excel.

11.3.1 Macro Recorder

If you don't know how to do something in code, you can always record a macro and then edit the code manually. Sometimes that saves a lot of time! In the "Developer" tab, there's a button "Record Macro", which lets you start the recording. Try it out!

[2] `https://office.com`

11.4 Syntax Mapping

11.4.1 Comments

VBA only knows line end comments, which start when a single quote (`'`) is encountered. Note that while there are no block comments, a line can be manually wrapped by adding an underscore (`_` - the *line-continuation character*) at the end. This line-continuation character can be used whenever we want to wrap a line while having a statement continue on the next line.

VBA

```vba
' this is a comment
' and this one _
  is continued on this line
```

There are two alternative ways of commenting you may encounter - both of which are strongly discouraged:

- Putting the keyword `REM` right at the start of a line (possibly after whitespace) makes the entire line a comment.
- Using preprocessor directives (e.g. `#If comment = 1 Then this is a comment #Endif`) - you'll have to define the variable you use in "Project Properties" > "Conditional Compilation Arguments" (e.g. `comment = 0`) by right clicking on your VBA project in the VBA editor.

11.4.2 Input and Output: Printing

There isn't really a console as such: the UI for interacting with the user is Excel itself, we typically read from and write to excel cells. To provide more explicit output to the user, we can also use a popup dialog, a ***Message Box***. And if we just want to create some debug output, we can write some information to the "Immediate" window ("View" > "Immediate Window"). Let's have a look at these three alternatives in turn:

VBA: Write to Cell in Active Worksheet

```vba
Range("D5") = "Hi, there!"
```

VBA: Show a Dialog Box

```vba
MsgBox("Do I have your attention now?")
```

VBA: Write Debug Output to Immediate Window

```vba
Debug.Print("Look at me immediately!")
```

11.4.3 Program Structures

11.4.3.1 Modules

Modules are where VBA code lives. They are units of code organization and can be imported and exported. A module can be created by right clicking an open workbook and choosing “Insert” > “Module”.

We are going to largely ignore *class modules* in this chapter, and we’re not going to look at how to create our own classes in VBA. We will, however come across the `New` keyword used to create an object: an instance of a class.

11.4.3.2 Methods -> Procedures

VBA uses terminology which differs from ours: what we call procedure they call *subroutine* and what we call method they call *procedure* - very confusing, unfortunately! At least they agree with us with regards to naming functions. **Sub**routines are declared using the `Sub` keyword, whereas for functions we use `Function`. We will use this terminology going forward (in this chapter) - mainly in order to make it easier for you when you read through any of the referenced links.

By convention, procedure names start with an upper case character.

If you’re used to other programming languages, the way a result is returned will seem rather weird: the name of the function is assigned the result that will be returned. It can be used as a normal variable in calculations and can be reassigned multiple times. Whatever value is assigned when the function exits is what will be returned to the caller.

VBA

```
Function CalculateFactorial(value)
  ' calculations
  CalculateFactorial = result
End Function
```

Pseudocode

```
method calculateFactorial = (value) => {
  // calculations
  return result
}
```

`Function`s and `Sub`s are `Public` by default, meaning they can be used by procedures in other modules. Prepending `Private` to their declaration will make them invisible outside the module they’re declared in.

VBA

```
Private Sub Enchant()
  ' work magic
End Sub
```

> You may see procedures declared as `Static`. If they are, the values of their local variables are preserved between procedure calls. I would very strongly recommend not using this feature as callers will typically not expect such behavior.

11.4.3.2.1 Optional Parameters & Default Values

Parameters can be declared as optional by prepending the keyword `Optional` - they must all be listed after any mandatory parameters. If a parameter is declared as `Optional`, the caller doesn't have to pass a value for it. Optional parameters can additionally be provided with `= $default` (`$default` being the default value) which will be assigned to the parameter if it is not passed by the caller.

VBA

```vba
Sub OptionalParameters(a, Optional b, Optional c = 0)
  ' ...
End Sub
```

If no default value is provided and the parameter is not passed, there are two cases:

- if the type of the parameter is `Variant`, we will receive `True` from checking `IsMissing(...)`, and printing the variable will produce "Error 448".
- if the type is declared, there will be an implicit default value, which is the data type's default. `IsMissing` will always be `False` in this case.

11.4.3.2.2 ByVal & ByRef

Any arguments we pass to a procedure will by default be passed *by reference*, meaning that the procedure really receives a *reference* (a pointer) to the parameter. This means that if we pass a variable we've defined in one procedure and assign a new value to the argument received in the procedure called, the value of the caller's variable will also have changed! We can make this explicit by adding the keyword `ByRef` to our procedure parameters. If we do **not** want to change the value passed, we can declare the parameter as being passed `ByVal` (by value). If we do, we will receive a copy of the value passed.

VBA

```vba
Private Sub Foo(a, ByRef b, ByVal c)
  a = 5
  b = 5
  c = 5
End Sub

Sub Bar()
  a = 1
  b = 1
  c = 1
  Foo a, b, c
  Debug.Print a ' => 5
  Debug.Print b ' => 5
  Debug.Print c ' => 1
End Sub
```

11.4.3.2.3 ParamArray

There is a possibility to declare a specific type of parameter which will allow any number of arguments to be passed. To do so, we can declare the last parameter of a procedure to be of type array (by appending parentheses to its name - see further below) and prefix it by the keyword `ParamArray`.

VBA

```vba
Sub MyNumbers(ParamArray nums())
  ' ...
End Sub
```

This means that we can call the method with any number of arguments, e.g.

VBA

```vba
MyNumbers 1
MyNumbers 3, 5, 18
```

11.4.3.2.4 Procedure Calls

You probably already noticed above that we don't put parentheses around the parameters we pass to a procedure. This is by design - a very weird one, if you ask me, but a deliberate decision nonetheless. Sometimes, parentheses are required, sometimes they are tolerated and sometimes, they result in syntax errors. Here are the rules:

1. If the procedure call is **not** the only statement on a line (or delimited by the *statement delimiter character* - `:` - you can think of it as the opposite of the line-continuation character) or there are other keywords involved (as is the case when assigning the result of a function call to a variable), parentheses around the parameter list are required
2. Otherwise, no parentheses are allowed around the parameter list, **however**,
3. Parentheses may still be added around individual parameters to indicate that they are to be passed by value instead of by reference (regardless of what the procedure declares)
4. Parentheses are not allowed around individual parameters which result in a type mismatch when evaluated (as an example, an object passed will be serialized [3] , so if an object is expected, the resulting string will lead to a type error).

In other words, we have a strange difference between calling a function, where parentheses are normally required (because we assign the result to a variable or pass it on,) and calling a subroutine, where parentheses are normally not allowed (if we pass more than one parameter).

VBA uses the newline to delimit statements, but sometimes, we want to write multiple statements on one line. To do that, we can use the statement delimiter character (`:`). If we want to use the statement delimiter character to delimit a subroutine call with no arguments followed by a procedure call, we need to use the explicit call syntax, prepending `Call` to the procedure name and - in this case - putting parentheses around the parameter list (because now we have another keyword, disqualifying rule number 1 from being applied). If we fail to do that, the (first) procedure call will be interpreted as a label (a location which can be jumped to using `Goto` - we won't look at that in detail).

Time to exercise!

[3] https://en.wikipedia.org/wiki/Serialization

11.4.3.3 Error Handling

In VBA, we have two basic options of how to deal with runtime errors:

- We can ignore them
- We can jump to an error handler

To ignore errors, we add `On Error Resume Next` to our code. From that point out, all errors will be ignored. If we only want to ignore them in a certain section of code, we can add `On Error Goto 0` at the point when we want to stop ignoring errors.

VBA

```
' Some code - errors will lead to a dialog

On Error Resume Next

' Code section in which errors will be ignored

On Error Goto 0

' More code - errors will lead to a dialog again
```

Instead of simply ignoring errors, we can handle them, e.g. by showing a custom error message.

VBA

```
On Error Goto handler

  'Some code

handler:
  MsgBox "Something's wrong: " & Err.Description
```

> If we want to register another handler to deal with errors that might occur in our error handler code, we need to first clear the previously raised error. The syntax for this confusingly is `On Error Goto -1`...

To raise an error manually, we can use `Err.Raise`. The first argument is an error number which makes sense if we have to deal with many possible errors - this will make it easier to report errors in large projects. Generally, though, the important thing to provide is a description which will make sense to users.

VBA

```
Err.Raise 1000, description:="The first parameter must be a number!"
```

VBA allows us to pass *named parameters*, meaning that we can label each parameter by prepending the parameter name and `:=` to the value we're passing. This is particularly useful if there are many possible optional arguments and we only want to pass some of them, skipping the others. It's also nice if those who read our code are not necessarily familiar with the procedures we're using!

11.4.4 Data Types

VBA doesn't force you to define the type of variables, constants, procedure parameters and return values, but it allows you to make them explicit, in which case they will be enforced. Whenever we don't assign a specific data type, we're implicitly assigning the type `Variant`, which means "any type at any time".

There are 11 specific data types in addition to `Variant` (more detailed information about data types can be found here [4]):

Type	Abbr	Size	Description	Default
`Boolean`		2 bytes	`True` or `False`	`False`
`Byte`		1 byte	Whole numbers between 0 to 255	`0`
`Integer`	`%`	2 bytes	Whole numbers between -2^{15} and $2^{15}-1$	`0`
`Long`	`&`	4 bytes	Whole numbers between -2^{31} and $2^{31}-1$	`0`
`LongLong`		8 bytes	Whole numbers between -2^{63} and $2^{63}-1$	`0`
`Single`	`!`	4 bytes	Floating point numbers between $-3.402823e^{38}$ and $3.402823e^{38}$	`0`
`Double`	`#`	8 bytes	Floating point numbers between $-1.79769313486231e^{308}$ and $1.79769313486232e^{308}$	`0`
`Currency`	`@`	8 bytes	Floating point numbers between -922'337'203'685'477.5808 and 922'337'203'685'477.5807	`0`
`Date`		8 bytes	Dates between 1st January 100 and 31st December 9999	n/a
`Object`		4 bytes	Reference to an object	`Nothing`
`String`	`$`	n/a	Series of characters. Length is variable by default but can be fixed.	`""`

Knowing these data types, we can now add `... As $dataType`, with `$dataType` the data type we want values to be of. We can thus declare our function `CalculateFactorial` more specifically:

VBA
```vba
Function CalculateFactorial(value As Integer) As Integer
  ' calculations
  calculateFactorial = result
End Function
```

[4] https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/data-type-summary

We can also use “short-hand” declarations by appending the abbreviations shown above directly to the name of the thing we’re declaring types for:

VBA

```vba
Function CalculateFactorial%(value%)
  ' calculations
  calculateFactorial = result
End Function
```

Time to exercise!

11.4.4.1 Operators

11.4.4.1.1 Arithmetic Operators

The following arithmetic operators can be applied to any numeric types:

Operator	Operation	Example
`+`	Addition	`2 + 4` → `6`
`-`	Subtraction	`5 - 3` → `2`
`*`	Multiplication	`8.5 * 2` → `17.0`
`/`	Division (regular)	`7 / 5` → `1.4`
`\`	Division => quotient, discarding remainder	`7 \ 5` → `1`
`Mod`	Modulo	`7 % 5` → `2`
`^`	Exponent	`4 ^ -1` → `0.25`

11.4.4.1.2 Logical Operators

Finally, the most important logical operators, which apply to boolean values (they can be used on any value, resulting in applying the operation bitwise, with `0` interpreted as `False` and `1` interpreted as `True`) are:

Operator	Operation	Example
`Not`	$\neg$	`Not True` → `False`
`And`	$\wedge$	`True And False` → `False`
`Or`	$\vee$	`True Or False` → `True`
`Xor`	$\veebar$	`True Xor False` → `True`

11.4.4.1.3 Comparison Operators

These comparison operators are applicable to any data type:

Operator	Operation	Example
`=`	Equality check	`a = 4`
`<>`	Inequality check	`a <> 4`
`>`	Greater than	`a > b`
`>=`	Greater or equal than	`a >= b`
`<`	Smaller than	`a < -1`
`<=`	Smaller or equal than	`a <= c`
`Like`	Pattern match check	`a Like ""`

Note that `Like` does not use normal regular expressions, but rather follows its own rules. Patterns can be made up of the following:

Element	Description
`?`	Any single character.
`*`	Zero or more characters.
`#`	Any single digit (0-9).
`[` `$charlist` `]`	Any single character in `$charlist`.
`[` `!` `$charlist` `]`	Any single character not in `$charlist`.

Additionally, we can compare objects to check if two variables refer to the identical (`Is`) or different (`IsNot`) objects.

11.4.4.1.4 Miscellaneous

There's one more important operator to take note of: `&` concatenates two expressions and returns either a `String` (if both operands are `Strings`) or a `Variant` holding a `String`. If either operand is `Null` or `Empty`, it will be treated as an empty `String` (`""`). Importantly, if both operands are `Null`, the result will be `Null` as well.

11.4.5 Data Structures

11.4.5.1 Constants and Variables

VBA

```
1 Dim x as Integer: x = 5
2 Const y = 2
3 Dim z as Object
4 Set z = Application.ActiveWorkbook
```

Pseudocode

```
1 variable x = 5
2 constant y = 2
3 // does not exist in pseudocode
4 //
```

There are a few things to note above:

- Line 1:
  - `Dim` stands for **dim**ension, originally probably from defining the dimension (the space used) of a variable in bytes.
  - it's not really possible to declare a variable and assign a variable to it at the same time. As a workaround, we're using the statement delimiter character.
  - it is good style to declare variables explicitly even if VBA doesn't force us to do that. If we'd like to be told about undeclared variables, we can set `Option Explicit` at the beginning of our files.
- Line 2: the type declaration is implicit for constants: the type is derived from the value assigned.
- Line 4: when we assign an object reference, we have to use `Set`.

Any variables we want to declare on module level we can explicitly make `Private` (accessible only inside the module - that's the default) or declare them as `Public` to make them accessible from other modules. To do so, we replace the `Dim` keyword by either `Public` or `Private`. For constants, we add these keywords in front of `Const` to do the same.

11.4.5.2 Arrays

In our mapping to pseudocode syntax, we're going to assume there's a function `IntegerArray` we can use (it's included in the code snippets). Without it, we'd have to assign each array element individually if we want an array of type `Integer`. If we're happy with a `Variant` typed array, we can simply use the built-in function `Array`.

VBA

```
 1 Dim intArray() As Integer
 2 intArray = IntegerArray(2, 4, 9, 1, 7)
 3 Dim first As Integer
 4 first = intArray(0) ' => 2
 5 Dim length As Integer
 6 length = UBound(intArray) + 1 ' => 5
 7 intArray(1) = 5
 8 ' => [2, 5, 9, 1, 7]
 9 ReDim Preserve intArray(5)
10 intArray(5) = 18
11 ' => [2, 5, 9, 1, 7, 18]
```

Pseudocode

```
 1 //
 2 input          = [2, 4, 9, 1, 7]
 3
 4 variable first = input[0] // => 2
 5
 6 variable len   = input.length // => 5
 7 input[1]       = 5;
 8 // => [2, 5, 9, 1, 7]
 9
10 input[5]       = 18
11 // => [2, 5, 9, 1, 7, 18]
```

This looks quite a bit different from what we're used to, so let's look at some details:

- Line 1: very strangely, the type is declared as `Integer`, not as `Integer()`.
  - On the other hand, the return type of the function `IntegerArray` is in fact `Integer()`.
- Line 2: there is no way I know of to cast [5] an array of type `Variant()` to `Integer()`, so we cannot use the built-in `Array(...)` to create our array.
- Line 4: array items are accessed using regular parenthesis (`(...)`).
- Line 6: the generic way of calculating the array length is `UBound(...) - LBound(...) + 1` since arrays can have any start and end index - in `IntegerArray`, we only specify the upper bound, so the lower bound is implicitly `0`.
- Line 9: we need to explicitly resize our array - using `ReDim` - to add an item. Without the keyword `Preserve`, we would lose the content of the existing array. Note that the value in parenthesis is the upper bound (the last index) and not the array size!
  - We could set arbitrary lower and upper bounds by providing both values`($lower To $upper)` where `$lower` and `$upper` are the smallest and largest index, respectively.

We can also declare multi-dimensional arrays: `Dim foo(1 To 10, 1 To 10)` declares a 10 by 10 array with indexes from 1 to 10 in each dimension.

> Entire arrays can also be assigned to a variable of type `Variant`. This will put constraints neither on the dimensions nor the content of the array.

[5] https://en.wikipedia.org/wiki/Type_casting_%28computer_programming%29

11.4.5.3 Maps

We can use the `Collection` object type (you'll find many places recommending using the `Dictionary` type, but that's not available on all platforms). `Collection` only supports string keys, which most times is not a problem.

VBA

```vba
 1 Dim shapeCorners as New Collection
 2 shapeCorners.Add 3, "Triangle"
 3 shapeCorners.Add 4, "Rectangle"
 4 shapeCorners.Add 5, "Pentagon"
 5
 6 Debug.Print shapeCorners("Rectangle")
 7 Dim x As Integer
 8 x = shapeCorners.Count ' => 3
 9 shapeCorners.Add 6, "Hexagon"
10 ' => Triangle=3, Rectangle=4,
11 '    Pentagon=5, Hexagon=6
12 shapeCorners.remove "Rectangle"
13 ' => Triangle=3, Pentagon=5, Hexagon=6
```

Pseudocode

```
 1 constant shapeCorners = {
 2   "Triangle":  3,
 3   "Rectangle": 4,
 4   "Pentagon":  5,
 5 }
 6 print(shapeCorners["Rectangle"])
 7
 8 // does not exist in our pseudocode
 9 // does not exist in our pseudocode
10
11
12 // does not exist in our pseudocode
13
```

- Line 1: we use the keyword `New` to create a new instance of the class `Collection`.
- Lines 2-4: note that the value is passed first and the key second
- Line 6: this is the shorthand syntax for `shapeCorners.item("Rectangle")`. It works because `item` is the default method of class `Collection`.

> - If your keys are really integers, make sure to use `CStr(...)` to convert them to strings. If you don't, you will be doing a positional lookup, which (a) is not what you think you're doing and (b) is very slow.
> - Should you require more functionality, this library [a] may be useful.

11.4.6 Objects

We've already worked with objects without really realizing it: both `Debug` and `Collection` are globally available objects, which are generally available in VBA. Another object we've come across above is `Application`, which gives us access to properties of the application we're working on, e.g. Excel - where we can get the property `ActiveWorkbook`. In fact, the properties of `Application` are also globally available, so we can just write `ActiveWorkbook` instead of `Application.ActiveWorkbook`. The documentation [6] gives you an overview of what else is accessible from VBA.

We can navigate to related objects via `properties` such as when we're accessing the active worksheet via `Application.ActiveWorkbook` or we can call procedures on them such as `Application.Calculate` to calculate all formulas in open Excel worksheets.

11.4.6.1 With

If we want to access multiple properties of the same object or call multiple procedures on it, we end up writing a lot of code. To simplify this, VBA offers a `With ... End With` block, within which we don't have to repeat the object name each time.

[a] https://github.com/VBA-tools/VBA-Dictionary
[6] https://docs.microsoft.com/en-us/office/vba/api/excel.application%28object%29

VBA

```
1 With ActiveWorkbook.Worksheets("Sheet1")
2   .Range(.Cells(2, 1), "B5").Copy
3   .Range("D5").Select
4   .Paste
5 End With
```

- Line 1: We're specifying that any property or procedure call starting with `.` will actually be on `ActiveWorkbook.Worksheets("Sheet1")`.
- Line 2: copies the contents between (including) cell "A2" (second row, first column) to "B5" from "Sheet1" into the clipboard.
- Line 3: selects the cell "D5" in "Sheet1".
- Line 4: pastes the previously copied content to the range starting from "D5" (in "Sheet1").

Time to exercise!

11.4.7 Input and Output: Reading

As we've seen for printing, there isn't really a console. We can still capture user input, though: either by reading from a worksheet or by providing an explicit dialog which accepts input.

To read from a worksheet, we can simply refer to the `Range` we're interested in:

VBA: Read from Cell in Active Worksheet

```
Dim val as String: val = Range("D5")
```

To present the user with a dialog requiring input, we can use the following:

VBA: Input from Dialog Box

```
Dim val As String
val = InputBox(Prompt:="What's your favorite value?", Title:="Favorites", Default:="42")
```

Time to exercise!

11.4.8 Control Structures

11.4.8.1 Conditions

VBA

```vba
If x < 5 Then
  Debug.Print "smaller"
Else
  Debug.Print "greater or equal"
End If
```

Pseudocode

```
if x < 5 then
  print("smaller")
else
  print("greater or equal")
end if
```

VBA has an additional syntax element which allows expressing multiple alternatives: `ElseIf`. You can for example write the following:

VBA

```vba
If x < 0 Then
  Debug.Print "negative"
ElseIf x > 0 Then
  Debug.Print "positive"
Else
  Debug.Print "exactly zero"
End If
```

To simplify things if there is just one option (no `Else` or `Elseif`), we can write the condition on one line and omit `End If`:

VBA

```vba
If x > 0 Then Debug.Print "positive"
```

11.4.8.1.1 Conditionals

VBA doesn't have a conditional that's a shorthand for `If ... Elsse ... End If`, but it offers a function that usually achieves the same result: `IIf`.

VBA

```vba
x = IIf(y >= 0, y, -y)
```

So what's the difference? `IIf` evaluates both possible values instead of just the one returned. This usually only has as performance impact, but can change outcomes if there are functions which have side effects.

11.4.8.2 Case Matching

In VBA, case matching is done using `Select Case`.

VBA

```vba
Select Case objectType
  Case "plant":  ' water it
  Case "animal": ' feed it
  Case "gas":    ' inhale it
  Case "liquid": ' surf it
  Case "solid":  ' decorate it
  Case Else:     ' stare at it
End Select
```

Pseudocode

```
if objectType
  matches plant  then // water it
  matches animal then // feed it
  matches gas    then // inhale it
  matches liquid then // surf it
  matches solid  then // decorate it
else                  // stare at it
endif
```

The colon (`:`) is the statement delimiter we already know. We can replace it by a line break.

There are other options for matching than simply using equality:

- for integers, we we can use `Case $m To $n`, with `$m` and `$n` integer values
- it's possible to provide multiple options, e.g. `Case 5, 6`.
- we can use the explicit syntax `Case Is = "plant"`, which allows for other comparison operators as well (e.g. `Case Is > 5`).
  - "Not in a list" can be expressed as `Case Is <> $csv`, with `$csv` a comma separated list of values (e.g. `Case Is <> 1,4,5`)
  - Note: this `Is` has nothing to do with the one we saw used for object identity checks.

Finally, we can also use what I would call a "degenerated" version by giving the main expression to compare to as `True` and then doing tests in each `Case` branch. We can leverage that to match patterns using `Like` (which we can't do otherwise):

VBA

```
Select Case True
  Case x Like "[a-f]*": Debug.Print "starts with 'a' to 'f'
  Case Else:            : Debug.Print "starts with another character"
End Select
```

11.4.8.3 Loops

In VBA, there are the same basic types of loops as in many other languages:

11.4.8.3.1 "While" Loops

VBA

```
Dim i as Integer: i = 4
While i < 4

  Debug.Print i
  i = i + 1
Wend
```

Pseudocode

```
variable i = 4;
loop
  exit if i >= 4;
  print(i++);

end loop
```

We can write the same thing as follows:

VBA

```
Dim i as Integer: i = 4
Do While i < 4

  Debug.Print i
  i = i + 1
Loop
```

We can also reverse the logic in the exit condition:

VBA

```
Dim i as Integer: i = 5
Do Until x >= 4
  Debug.Print i
  i = i + 1

Loop
```

11.4.8.3.2 "Do While" loops

VBA

```
Dim i as Integer: i = 5
Do
  Debug.Print i
  i = i + 1
Loop While i < 4

```

Pseudocode

```
variable i = 5;
loop
  print(i++);

  exit if i >= 4;
end loop
```

11.4.8.3.3 “For”-loops

VBA

```
For i = 2 To 5


  Debug.Print i
Next
```

Pseudocode

```
variable i = 2;
loop
  exit if i > 5;
  print(i++);
end loop
```

There’s an additional cool feature: we can add `Step $step` - with `$step` an integer - to have an increment different from `1`. Note that `$step` can be negative.

VBA

```
For i = 1 To 10 Step 3
  Debug.Print i
Next
' => 1 4 7 10
```

11.4.8.3.4 “For Each” Loops

If we need to loop through an array, we can use the following:

VBA

```
For Each $item in $arr
  ' ...
Next
```

11.4.8.3.5 Exiting early

We can exit these loops by using `Exit Do` or `Exit For`. Exiting a `While` loop is not possible, but it can easily be translated to a `Do While` loop, which can be exited with `Exit Do`.

Time to exercise!