Introduction to Programming ... in C#

by Aslak Johansen

2026-05-21

Contents

List of Figures

List of Syntax Defintions

Preface

My background is from fairly classical (experimental) computer science, and has always been programming language tolerent. Today, I am teaching an introductionary programming course on a Danish software engineering education. That course has a focus on object-orientation. Previously, the it targeted Java, but we recently made the shift to C#.

C# (and Java for that matter) are languages that mix the fundamentals with a great amount of complexity. From what I see, students have a tendency to loose track of those fundamentals. So, why teach such langauges on the first course on programming? Well, only about half of a Danish software engineering education is computer science. The rest is project management, customer relations, requirement analysis and similar fields that are peripheral to programming. That means that some shortcuts have to be taken. And this is one.

Many C# books already exist, and quite a few of them claim to be aimed a novices. But as far as I can tell, they all bypass learning experiences that I feel are essential to informing the understanding of the student. This book is intended to fill that gap.

Chapter 1

Introduction

Before we can program we need a conceptual model of this computer thingy that we want to program. A useful way of looking at this is to see the computer as a general-purpose machine. The task of programming it then becomes the art of making that machine take a specific form that allows it to solve a concrete problem or class of problems. It turns out that there are common techniques, patterns and constructs that can help us out. Some are simple and others are very complex. Complexity, in and of itself, is a bad thing. It makes it hard for humans (such as programmers) to grasp what is going on. But sometimes no simple solution exists, and we may be forced to accept a complex one. It is up to the programmer to reason about this, and make informed decisions of *how* the software is constructed.

Depending on who you ask, the task of programming lands somewhere between a craft and an art. No matter where on the spectrum you see it, the decisions you make while programming should be rooted in theory. But why, you may ask? When programming something nontrivial, we are producing complex setups on top of a complex foundation. That foundation covers the hardware of the computer itself, the operating system running on top of the computer, other software running on the operating system, the programming language(s) at play, libraries of existing software, and … perhaps … other computers over a network. Understanding (the theory behind) that foundation is fundamental to constructing good software. Yet, for the vast majority of new students, practice is what allows them to connect with those fundamentals and begin to reason about programming.

Some languages are considered low-level. They are suited for working with concepts that relates to the computer itself. Your operating system, or at least its kernel, is written in such a language (or languages). Other language are considered high-level. They are suited for working with concepts that relates to domains outside of the computer. This could be a card game. In reality there is a spectrum between low and high level, and there is a myriad of languages that can be placed on this spectrum. Often, and especially for non-programmers, it is easier to spot the value generated close to the problem domains, and these areas then to lend themselves to high-level languages. These typically make programmers more efficient while being less efficient computationally. That is, they trade execution efficiency of development efficiency. As processing power is cheap and many tasks are tolerant in terms of response time, often computational efficiency is not considered critical or even important. Yet, there can be good reasons to care, namely (i) to improve user experience, (ii) to lower the cost of operations, and (iii) due to the ethics of how power consumption is driving climate change.

As this is an introductory course, we will mostly not care about efficiency. This textbook was

written primarily with a software education in mind that is mostly focused on how software is constructed for organizations by essentially gluing existing solutions together. I would call this a high-level software education. In such there is little time to learn about programming, so it is important that all the programming courses are directly applicable. For this reason we focus on a high-level language, namely C#. This choice, as any other, comes with downsides. So, let's take a look at the main ones:

- Once you have learned a language it is easy to go up in abstraction but hard to go down. When you go up in abstraction, there is somethings that you no longer have to worry about. When you go down in abstractions, something that you have come to rely on is taken away. By going the way of a high-level language in the beginning, makes it likely that you will stay there.
- Any high-level language will have high-level constructs, and why they may at times be easy to use, they tend to be hard to understand. It is simply far away form how the computer works. Unfortunately, that understanding is key to becoming a good programmer. A more ideal approach would be to start with a low-level language, then teach a medium-level language and finally end up with a high-level language. But this all takes time, and on a high-level software education that is unfortunately at a premium.

1.1 Intended Audience

This book will make no assumption of previous programming experience. It will, however, expect you to have some minimal understanding of your operating system of choice. You should, for instance, know how to install software, and preferably have an understanding of how your filesystem is structured. Section 2.2 covers what you need to know about your filesystem, but it won't make up for lack of practical experience. Finally, we expect you to have basic skills in looking up information on the internet and, critically, assessing its relevance and quality.

1.2 Tenets

1. The role of a course introducing programming is to create a technical foundation, not to please the industry. Ideally, students of such a course can – in time – bring positive change to the industry.
2. Practice should be rooted in theory.
3. The why is just as important as the how, if not more so.
4. Concepts are more important than concrete implementations.
5. Concrete implementation choices can often be highlighted by contrasting to other implementations.
6. Focus should be on concepts to root an understanding in rather than patterns to memorize and and apply.
7. Students should be able to look to educational material for good examples of how to write.

1.3 Reading Guide

This book is, as so many before it, intended to be read from beginning to end. The C# programming language has a lot of intricacies that make it virtually impossible to serialize the subject into a sequence of elements whose understanding only relies on the previous elements. That is a dependency graph all edges go backward in time[1].

This book is organized in topics which are grouped in themes. After this introduction you will find a background chapter with a summary of the theoretical that you are expected to have. Then comes a prereqs chapter that takes you though the process of installing the software that allows you to try out the examples and solve the exercises. Then follows four parts covering relevant topics and finally you will find a bibliography and an index.

Each part is made up of a number of topics. These topics are typically presented through a motivation, some generic theory, language-specific variations and finally exercises. While C# will be the natural centerpiece of the these variations, it will be contrasted to a number of other languages for context. The parts are introduced by a single example that gives a good impression of what will be covered.

1.3.1 Mindset

> *"It's not the Destination, It's the journey."*
>
> — *Ralph Waldo Emerson[3]*

People rarely decide to undertake the hardships of learning something because they want to go through the process of learning. They want have been through the process and know. It's the destination that pulls them in. As intelligent beings we subconsciously seek shortcuts, yet we are not particularly good at keeping the long-term goalpost present in mind. In order to absorb and understand the knowledge we need to be present along the way, observe the details and experience how they relate to each other. This applies to programming, perhaps more so than to most other fields.

This book is intended to help you in during the very first leg of your journey. As that is the first of many, focus will be on learning the basics rather than learning concrete tools and technologies that are used in the industry. At times these goals overlap, but more often than not they conflict. So, this book will not prepare you for the industry. That is a task you will have to deal with later on in your journey.

This leg of your journey can be seen as a journey in and of itself. It aims to bring you from some point to a destination. The conception that reading the book and finding the right answer to the exercises will bring you to that destination is a fallacy. There are many steps on this path, and how you tread them matter. Copying someone else's homework (be it a human or a generative AI) will help you *"solve"* the exercises, but doing so does not make you progress in your journey. These are empty calories and consistently going for that option is detrimental to your education. To benefit from from solving the exercises you need to go through the thought process of deriving a solution. You have to struggle a bit for your mindset and understanding to be adjusted. And for that reason, we purposefully don't take the straight path.

[1] This should make sense after reading section 2.1. That forward dependency should illustrate why they are bad.

That is also the case when it comes to the way you work with this book (and other resources, for that matter). Information will rarely be available in your preferred format. You need to build up skills for dealing with information in formats that you have no control over, and figure out how to best incorporate that in your workflow. For instance, if the information you need is in PDF format, then you can either print it, or spend a portion of your screen real estate displaying the file. In the latter case you should choose a decent PDF viewer (e.g., not something that came with your browser), and consider using the remainder of your screen for your coding. Operating two full-screen applications will impose the penalty of a large number of context switches on you. Conversely, when you are communicating to others, you will rarely be in a situation where your preferred means of communication fits that of your audience.

This is a significant part of the reason why this is not a website. Websites can be really good for many things. They are, for instance, good for quickly looking up definitions that it's not worth it to memorize. But that is a different problem. Learning to program requires you to interact with the matter, to tie individual parts of the material to personal experiences; be it the success of completing a small game in an exercise or being annoyed by not having everything "just work" and having to manually look though the index to find what you are looking for. Simply clicking through the material will not force you to involve yourself in it, and then your brain won't build the connections necessary for the constructed knowledge to persist over time. The effort will be lost.

So, we need your help in achieving this. You do so by keeping a focus on the following across your studies:

1. ***Curiosity*** The most important characteristic of a student of programming is that of curiosity; the desire to understand how things work and the willingness to invest time into exploring what makes them tick.

2. ***Pride*** Programming is a craft, and it needs to be honed. One needs to build up an opinion about what it means for code to be good. This is done by repeatedly attempt to improve the state of the codebase. Handing over a codebase to someone else should be a satisfying moment; one that makes you feel proud of what you have accomplished.

3. ***Thoroughness*** There is more to writing code than simply implementing functionality. How that functionality is achieved matter, and the quality characteristics involved are both many and intertwined through, e.g., through tradeoffs. When learning the trade, the value of implementing the functionality is negligible, and the details are much more important.

4. ***Communication*** Programming, like any other even remotely complex field, introduces concepts and notions. A technical vocabulary is needed to efficiently and unambiguously reference these. It is critical to acquire this vocabulary and exercise it in your daily interactions. Then trying to convey technical information, if you don't use the appropriate technical terms, the receiving party will more likely than not misunderstand you.

5. ***Effort*** Knowing how to best apply your time is key. Have pet projects, and choose them in such a way that they inspire your academic curiosity and/or are investments into your own ability to take in knowledge. Avoid squandering the time you put in trying to achieve arbitrary goals such as a perfect score.

> *"The first principle is that you must not fool yourself — and you are the easiest person to fool."*
>
> — *Richard Feynman[4]*

1.3.2 Index

At this point, you may have noticed some highlighted seemingly random words or phrases. This is by no accident. These are the words that appear in the index. If they, in addition to color, are underlined, then it will be a definition of a term. Entries of the index are grouped so that in order to look up a "directed edge", one first looks up "edge", and then "directed" among the related subentries. Page numbers with definitions are marked with bold.

1.4 Computer Literacy

Let's briefly take a look at how different people use computers. Some simply don't. Others do, but in reality all they use it for is as a window towards web services. In that respect, they are more users of web services that of computers. Others yet actually use their computer itself and have the fundamental understanding of it to do so. To construct software for the computer requires a deeper understanding though, and a culture. This could be summarized as:

- ***Level 0 "Computer Illiterate":*** Incapable of using a computer.
- ***Level 1 "Browser Literate":*** Capable of booting a computer, starting a browser and operating the internet through it. This implies a basic understanding of what a website is, internet security, accounts, search technique and copy paste.
- ***Level 2 "Desktop Literate":*** Capable of operating a general purpose operating system. That covers installing programs, managing local files, and understanding the file system hierarchy as well as the decoupling of file types and applications.
- ***Level 3 "Developer Literate":*** Capable of operating a general purpose operating system as a software developer. This includes having an understanding of environment variables, standard streams, the terminal, color representation, file format representation, the processor and the memory hierarchy, and programming language syntax.

Besides teaching a programming language, C#, this book also aim to raise the readers computer literacy to level 3. You are expected to be somewhere between levels 1 and 3. If you are at level 0 then this book is not for you.

1.5 Practical Experience

When learning programming, the need for practical experience is absolutely crucial. You may understand all the examples in the book and agree with all of the solutions to the exercises, but that does not make you capable of solving problems through code. The easiest way – by far – to build an understanding of the subject is through repeated use. Struggle is not something to avoid, it is a necessity.

That is not to say that you can learn everything you need to learn by simply writing code. The theoretical side needs to come from somewhere else. Otherwise, how would you know what to write? Videos may seem like a good source, but because they are so easily consumed, it requires

an extraordinary character to stay focused while consuming them and because the information is available at such a low effort, it usually turns out fleeting in nature: What goes in quickly leaves quickly as well. You may think that what you have just seen seems reasonable, but you have not truly thought it through. That requires effort and, perhaps more importantly, time. Books, on the other hand, forces you to stay active while reading. But it is not enough to read the text; you should frequently check that you understand what you read (e.g., at a paragraph or even sentence level). This checking forces you to reflect on what you have read, and that builds the structures in your brain that is what allows you to recollect the knowledge and to reason about it. Light struggle is particularly beneficial here. So, when you get annoyed that you have to turn a page for a figure (or, more likely, scroll for it) then you are actually training the neural network of your brain to remember the context of what you have just read.

> *"Smell is the most powerful trigger to the memory there is. A certain flower or a whiff of smoke can bring up experiences long forgotten. Books smell musty and rich. The knowledge gained from a computer is … it has no texture, no context. It's there and then it's gone. If it's to last, then the getting of knowledge should be tangible. It should be, um, smelly."*
>
> — *Rupert Giles*[13]

You will often be in doubt of whether something works in this way or in that way. We will provide you with three means to get out of such a situation. These are, in order of descending attractiveness:

1. Consider the consequences of the two options: If one of them lead to a ridiculous consequence, then the other option is likely the correct one.
2. Figure out what the pivot point is. Then write a program, that is as small as possible, where you from its execution can draw your conclusion.
3. Go look for other sources of information.

1.6 Generative AI

[1]

[6]

1.7 Societal Responsibilities

Be advised that his section gets fairly dark. If you are not somewhat informed about the state of the planet before reading this, you will not have a good day. There is no point in having our perception of reality deviate from the trajectory of the planet though. If we are to avoid a total collapse of the societies that we know today then we need to accept where we are headed and actively take steps to change direction of a complex of issues. A moonshot is necessary, and it will need effort, focus and resources like no other project in history.

The UN has found that we have moved from a water crisis (which is a temporary thing) into a state of global water bankruptcy[9]. This bankruptcy is defined as the combination of insolvency and irreversibility. Insolvency, because we are withdrawing water from the environment faster than it can be replenished and have crossed safe depletion limits. Irreversibility, because damage

both has been done and is being done to the natural mechanisms needed to restore the capacity of our water systems. Production of electronics and data center operations are major consumers of water. The more they consume, the less there is for the part of the human population that can afford it. As the price goes up and the availability down, the part of the population that is too poor for clean drinking water is growing. With explosion in data center construction for generative AI, many communities in the USA have begun realizing that they might soon turn out to be too poor to compete with a heavily subsidized generative AI industry over clean water.

Under the umbrella term of the Gaia Hypothesis, James Lovelock described the Earth as a living organism[8]. It is governed by a complex weave of control mechanisms including the ocean current systems, the winds, the rainforests, and many others. Around the industrial age we slowly began dismantling these control mechanisms. Whenever we eradicate some species that triggers changes in other species which in term causes a feedback loop to break, and suddenly the effects become significant. The large coral reefs can no longer handle the rising sea temperatures and are going to become a wonder of history. That is not just a loss of beauty, but a massive it to the oceans ability to support life. We are cutting down the rainforests an ever increasing rate. The rainforests plays a major role in regulating the climate, and the are sequesting enormous amounts of carbon. As we replace them with deserts, we loose these capacities and that results in less stable climate conditions, typically of higher temperature. This climate change is causing the Atlantic Meridional Overturning Circulation (AMOC) oceanic current system to weaken and it is currently approaching a potential collapse. This current is not particularly well understood, but there likely are tipping points that will cause an eventual collapse once passed. The AMOC is one of the great heat distributors of the planet. Without it, the tropics will we warmer and the arctics colder. In short, the bands of livable conditions would become narrower. As the temperature goes up, ice melts and that makes the planet darker which decreases the albedo. This in turn causes the planet to absorb more sunlight. The risk of a runaway scenario is real, but – again – not well understood. Basically, we won't know until it is too late and it might already be. We need to slow down this self-inflicted destruction of the planets mechanisms for sustaining human life. But the ball is already rolling. To stop it we have an urgent need to drastically reduce emissions, and software plays a pivotal role here. In part because a lot of the energy consumption is linked to software, in part because software development – due to the rise of generative AI – has a future set out for it that is extremely wasteful, and in part because software can be an enabler to reduce energy consumption in all kinds of other domains.

How people produce software has changed quite a bit through the course of the history of modern computing. In the beginning hardware was expensive, slow and had few resources (e.g., little memory). This placed a need to produce simple and efficient code on the software developers of that age. As photolithography improved, year over year, more transistors could be crammed onto the same piece of silicon at the same price[10]. The observation was that the number of transistors on an integrated circuit (i.e., a chip) doubled every two years. This later became known as Moore's Law, and it meant that programmers had ever increasing processing capabilities to exploit. Programmers got careless with resources, and the time it took to produce the software began to gain importance. When the Internet came, some software was split into a backend (running on a server on a centralized computer) and a frontend (running in the browser of the client). That frontend part was running on client resources and there was thus little incentive to make it efficient. As mobile devices (e.g., smartphones) were introduced this trend continued. Today, developer price has become a primary parameter of software development. With generative AI, production of software has become messy and power consuming, and so has data centers and networks. And the code being produced is incredibly inefficient compared to how it all started, but that also means that there are gains to be made.

[5]

Many futures have the potential to lie ahead of us. The most likely ones are truly dark, but we do have some agency in what will come to pass. Through our actions, we are collectively picking among these potential futures. It is up to our generations to make that decision in the pivotal moment in the story of humanity. I challenge you to make better choices than those who came before you.

1.8 Acknowledgments

Thanks to:

- Marcus Thomsen
- Evita Simaresi
- Sigurd Preibisch Behrens

Chapter 2

Background

Before we get started on installing the prereqs for this course, we need to cover some basic theory. This is not only relevant for installing the prereqs, but will pop up frequently throughout this course, and after it.

2.1 Graphs

A graph is the combination of a set of nodes and a set of edges between these nodes. You could see the nodes as *things* and the edges as *relationships between things*. But let's take a look at the concrete example of figure 2.1. Here, we have nine nodes – n_1 through n_9 – and eleven edges. The layout of where the individual nodes a placed doesn't mater. So, we could flip the positions of n_4 and n_7 whichout changing the graph itself. Although, the representation of the graph would obviously change.

Edges may be directional as in figure 2.2. This means that there is an edge from n_1 to n_2 but not the other way around. We say that the source node of this edge is n_1 and the destination node is n_2. Another way to look at it is that this edge belong to the set of outgoing edges of n_1 as well as the set of incoming edges of n_2. Most graphs in computer science consists of directed edges. If all edges in a graph are directed, then we say that the graph is directed or simply that it is a *digraph*.

Both nodes and edges can be associated with data. Edges, for instance, are often associated with weights. Such a graph is called a weighted graph. Figure 2.3 illustrates such an example. Directed graphs can be used to model road networks. Each node is then a point of interest (e.g., a city or a crossing), and edges represents roads. The weight of an edge is the distance of that road. So, we might say that there is a distance of 1.1 km from n_9 to n_8.

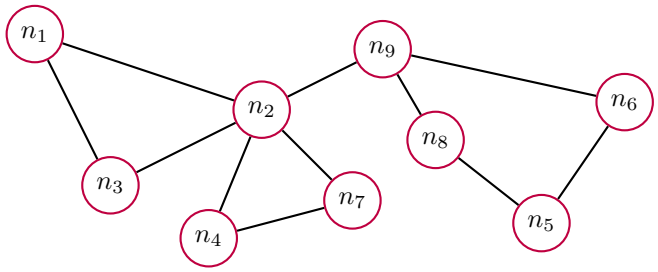


Figure 2.1: Example of a graph.

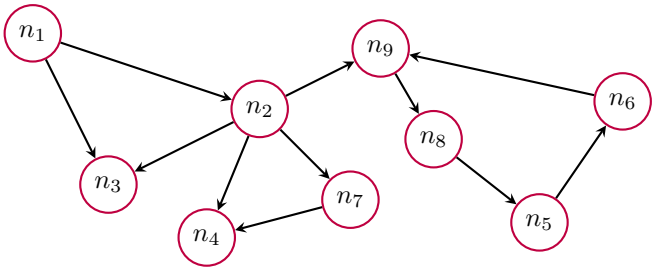


Figure 2.2: Example of a directed graph.

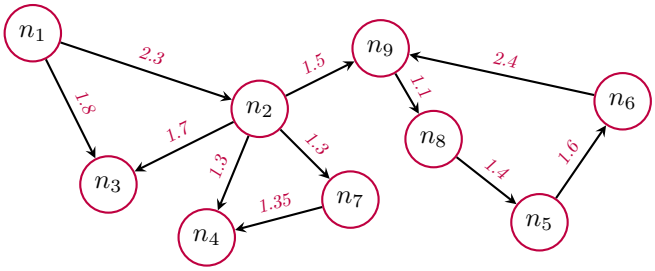


Figure 2.3: Example of a weighted graph.

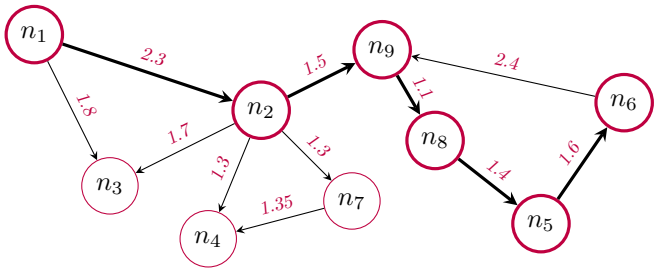


Figure 2.4: Example of a path in a graph.

A routefinding application – like Google Maps – has access to a a weighted graph of some area. When you ask it for a route from n_1 to n_6 it will try to find a sequence of edges where the source node of one edge is the same as the destination node of the previous edge. Such a sequence is called a path, and the resulting path is illustrated in figure 2.4.

Figure 2.5 illustrates a special kind of path called a cycle. This is a path where the destination node of the last edge is the same as the source node of the first edge. While they are often necessesary, their potential existence complicates the code of the programs that have to deal with them. For this reason we often see software designed to handle the class of graphs that have no cycles. These are called *Directed Acyclic Graphs*, or DAGs for short.

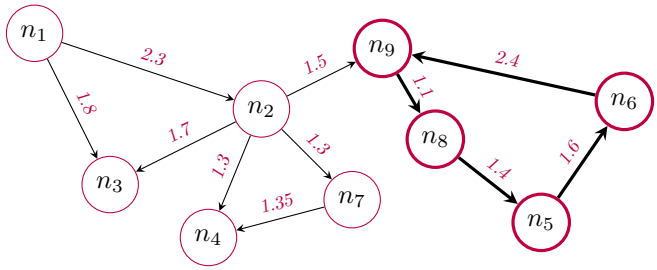


Figure 2.5: Example of a cycle in a graph.

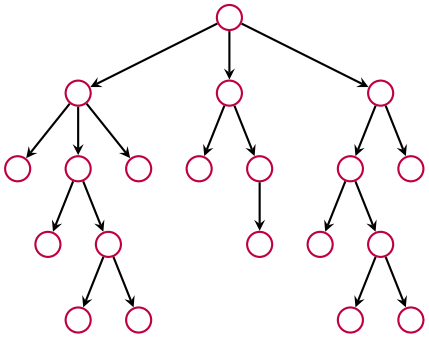


Figure 2.6: Example of a tree.

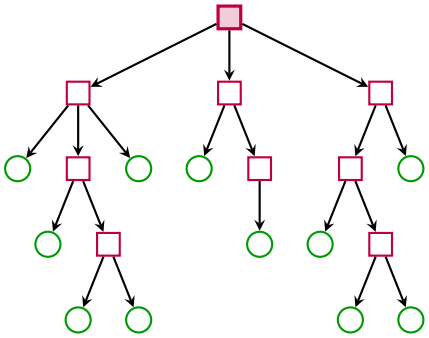


Figure 2.7: Node types of a tree.

2.1.1 Trees

A connected graph with n nodes has at least $n - 1$ edges. If it has exactly $n - 1$ edges then it belongs to the class of graphs called trees. Trees are traditionally drawn upside down, like in figure 2.6. Just as with other graphs, we have some liberty in how to draw the nodes and edges. Here, we have left out the identities of the individual nodes.

A special node of the tree is called the root node. It is drawn at the top of the illustration, and it has exactly one path the each of the other nodes of the graph. As illustrated in figure 2.7, each of the nodes in a tree are often categorized as either leaf nodes (with no outgoing edges) or branch nodes (typically with outgoing edges).

Trees are typically used for searching or representing hierarchical data. This could be a taxonomy tree, or the *nesting* of elements in a graphical user interface.

2.2 Filesystems

Most computers organize persisted data in files. That the data are persisted means that the data will survive a Power cycle of the computer (i.e., that it is turned off and on again). All data available to the operating system (includin programs and configuration) is stored in files in a filesystem.

Filesystems are *mostly* tree structures, organizing files in directories. A directory is a special form of file that can group other files. In tree terminology, they are branch nodes whereas ordinary files are leaf nodes. The visual metaphore that graphical user interfaces use to refer to directories

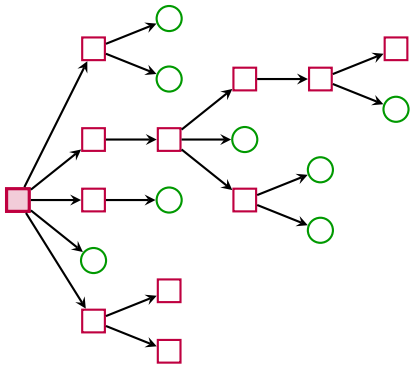


Figure 2.8: Example of a filesystem.

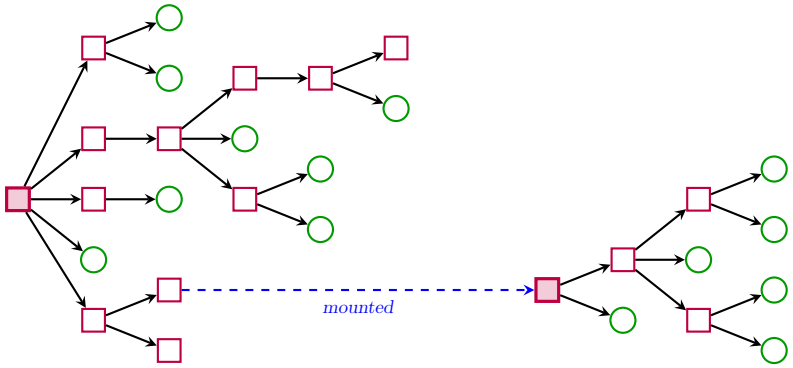


Figure 2.9: Example of a mounting a guest filesystem.

is the folder. But at the filesystem level – which is what we care about as programmers – it is a directory. Figure 2.8 shows what a typical filesystem could look like.

2.2.1 Mounting

Removable media, like a USB drive, are usually formatted to contain a filesystem. That is why we can place images and documents on it. In order to access that filesystem on your computer it must be mounted once attached. A mount operation essentially makes a specific directory – the mount point – a shorthand of the root of the mounted filesystem, thereby practically grafting the mounted filesystem to the root filesystem. The result of the mount operation is illustrated in figure 2.9. On DOS-based systems (such as the Windows family of operating systems), each known filesystem is assigned a letter, and a (hidden) root directory refers to each of these.

2.2.2 Links

Modern filesystems support links though, and that is a feature that makes them break their adherence to the tree definition. A link is a special kind of file that refers to an other file (regular or directory). Figure 2.10 shows a scenario where a link refers to a regular file. That means that we have two paths from the root node to this regular file; one that goes through the link and one that doesn't. That means that the filesystem is no longer a tree structure. As illustrated in figure 2.11, a link can also go backward. By pointing to a directory earlier in the path from the root node to the link node a cycle is created. This will complicate things if we need to scan the

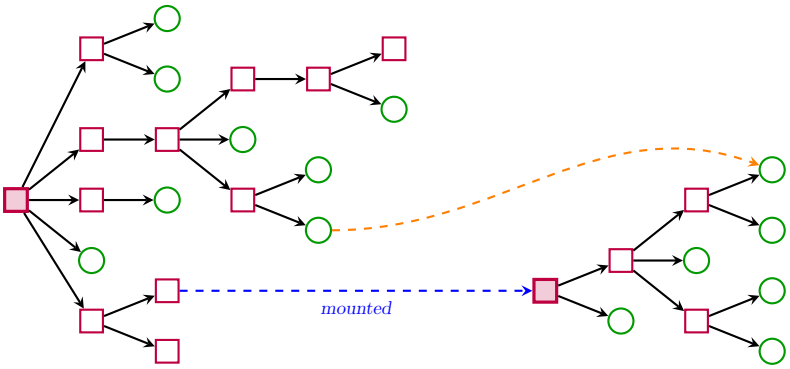


Figure 2.10: Example of a link in a filesystem.

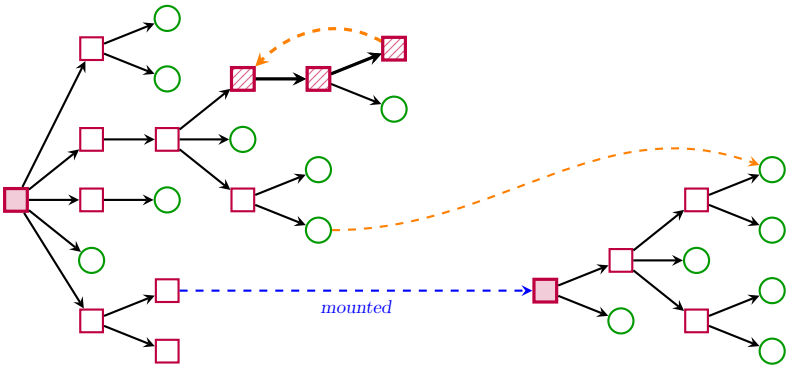


Figure 2.11: Example of how a link can create a cycle in a filesystem.

entire filesystem. Because of this, you often see commandline tools have options for following or not following links.

While modern versions of the Windows family of operating systems does have some support for links, it can be sketchy. Instead they support shortcuts through a special `LNK` file format. The content of a file following this format is a path to a file that the link points to. While this solution is a bit more flexible, it has to be interpreted by each application instead of the filesystem and is thus harder to work with and less performant.

2.2.3 Paths

When we want to work with a file, we need to be able to refer to it. That happens through a name, and that name can take two forms; either it is an *absolute* path or it is a *relative* path:

- ***Absolute Path:*** An absolute path (like the one from figure 2.12) is a path that starts in the filesystem root and goes to the node representing the file of interest. That is, it is the sequence of steps needed to be taken in order to step through the graph of the filesystem in order to go from the root to the file.
- ***Relative Path:*** A relative path (like the one from figure 2.13) is essentially the same as an absolute path except that the starting point is not the root of the filesystem, but the current working directory.

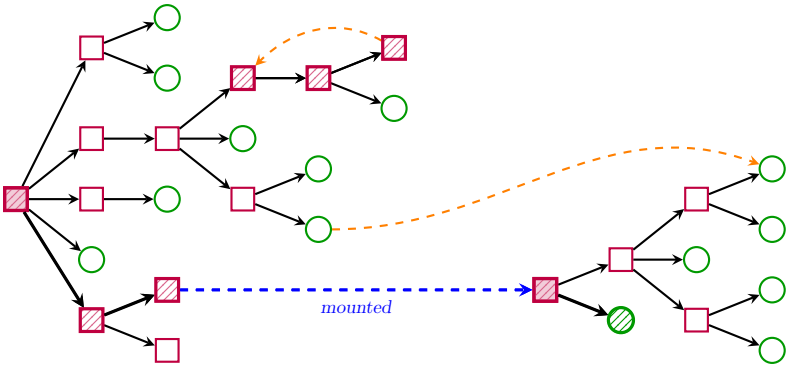


Figure 2.12: Example of an absolute path in a filesystem.

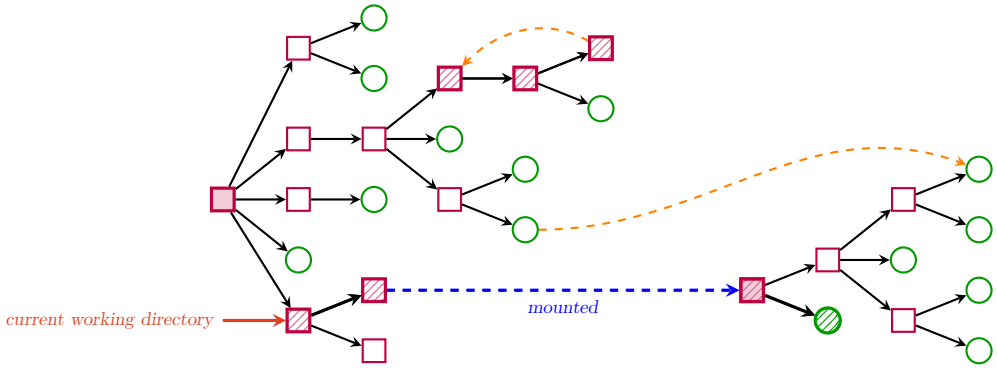


Figure 2.13: Example of an relative path in a filesystem.

> Every process is – so to say – standing in a directory, and this is called the current working directory. Current, because you can change it by transitioning either towards the root of the filesystem, or by following one of the outgoing edges of the node representing the current working directory.

So, a path is a sequence of names of steps (typically directories) in a filesystem. In practice, this is written in different ways depending on which operating system that needs to interpret it. On UNIX systems, a forward slash "/" is typically used as path separator. DOS-based systems – like the Windows family of operating systems – uses a backslash "\" instead.

2.3 Naming

Naming is a concept that is central to computer science. In this context, it has nothing to do with coming up with good names for things. Instead, it refers to the ability to reference things. In programming, we rely on a number of mechanisms for looking up references to things. Generally, they follow the model in figure 2.14. In this book, we cover several of these.

For now, we will leave you with an example: In figure 2.15, you will find an example of the structure of a typical software project. At the root of the project there are directories for the source code (`src`) documentation (`doc`) resources (`var`) and configurations (`etc`). Each of these contains files and/or subdirectories.

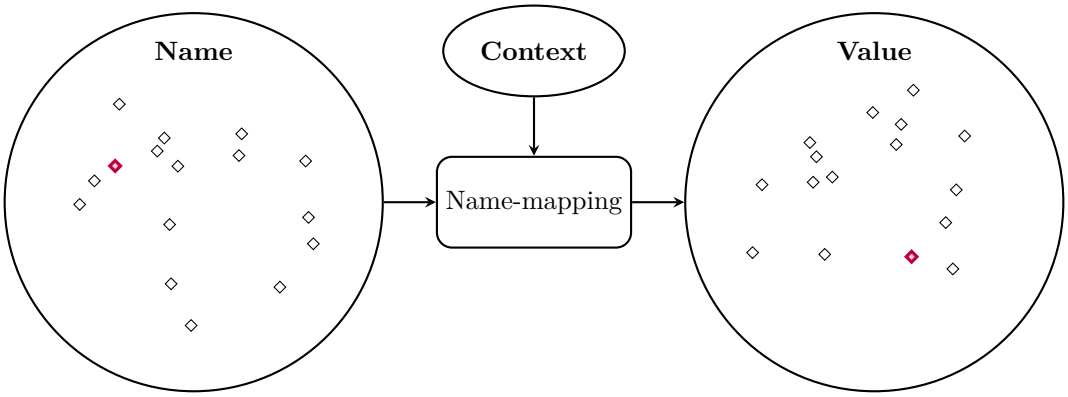


Figure 2.14: Name mapping principle. Given a context, each name may be mapped to a specific value.

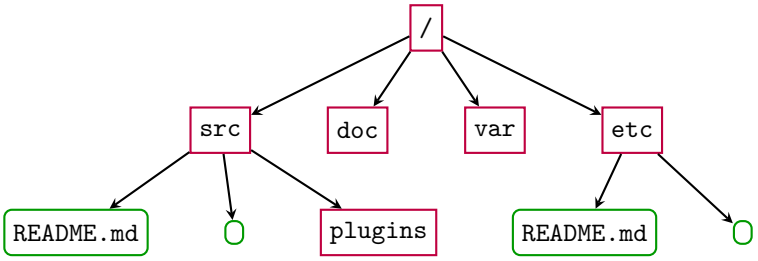


Figure 2.15: Example structure of the files in a software project. Note how parts of the directory structure have been collapsed.

Let's take a look at what happens when we look up a specific file. Starting from the root (`/`) of the project directory:

1. Given the context of `/` we look up `src`. We will call the result `/src`.
2. Given the context of `/src` we look up `README.md`. We will call the result `/src/README.md`. That is, the `README.md` file that exists in the `/src` directory.

You often see this form of repeated lookup. Because files are looked up through their encapsulating directories, two files of the same name can exists and be named. This is because they are looked up through different contexts (i.e., directories in this case). They are uniquely identified through a path. But the concept of naming extends beyond files. It has many uses inside program source code. Throughout this book you will encounter numerous examples of naming.

2.4 Processes

Seen from the perspective of the operating system, the stuff that runs on your computer is split into processes. Sometimes an application will consist of multiple processes, but most of the time there will be one process per application. But there are other processes. Some perform services under the hood, and others handle the interaction between the user and the operating system.

2.4.1 Kernel

Any general-purpose operating system will have a kernel at its core. The kernel is a gatekeeper of pretty much every central operation. It is involved whenever the computer receives input and whenever a process want to create an output. It is where processes are created, stopped and given time.

Processes are typically written as if nothing else is running on the machine. But in reality, lots of processes are running at any given time. In fact, while I am writing this text, I have 643 processes running on my laptop. Some of them are active, and others are waiting for some kind of input. The kernel is making sure that each of the active ones get time (on the processor) on a regular basis. As other processes need time as well, this also means that there are interruptions in the time given to each process. This in term means that the processes essentially experience gaps in time where they are frozen.

2.4.2 Desktop Environment

Most people interact with their computer through a desktop environment. Desktop environments are comprised of a number of processes responsible for drawing on the screen, arranging windows, starting applications, services for shared configuration and communication between applications.

Examples of desktop environments include Microsoft Windows, Apple OSX, Gnome and KDE. The latter two are popular choices on Linux, but there are many other options. Desktop environments are sometimes referred to as *desktop shells*.

2.4.3 File Manager

All successful desktop environments come with a file manager (or more). These are graphical programs that allows the user to browse their filesystem. Typically they display a directory at a time, and some also shows a tree structure of the directories. Here, directories are usually illustrated visually through a folder symbol and files are often previewed.

2.4.4 Terminal

One type of application that you will get familiar with is that of a terminal. In some ways, this is an extremely simple application. It opens a window that can act as the interface to a text-based program. It allows you to provide inputs to that program, and see the output that program prints to the *screen*. Sometimes a terminal is called a "console" or a "command prompt". The latter term is a bit unfortunate though as it also encompasses a *shell*.

2.4.5 Shell

The typical program to run inside a terminal is a shell. This is an interactive program that provides you with a prompt. Through the prompt you can issue commands. Some commands change the working directory, some adjust environment variables (i.e., a set of named values that affect the way programs execute), and some start other programs. Figure 2.16 illustrates how one can use the `cd` command to change the working directory.

Different shells will have different names for commands that perform similar action. In the book, we follow the convention of the bash shell (that is often default on Linux). The commands that we cover are called the same in zsh (that is default on OSX). Commonly used commands include:

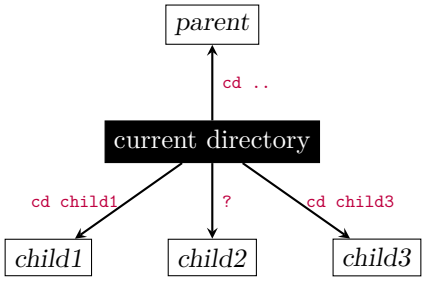


Figure 2.16: Navigating the filesystem.

- `ls` (*list*) This command lists the contents of a directory. A few variants:
  - `ls -l` Also display select metadata information for each file.
  - `ls ..` List contents for parent directory (any valid path can be given as parameter).
- `rm` (*remove*) Remove a subtree (unlink really) of the filesystem. Look it up before you delete anything of value.
- `mkdir` (*make directory*) Create a new directory:
  - `mkdir mydir` Creates a directory in the current working directory.
- `cp` (*copy*) copy a subtree of the filesystem (by default only a single element):
  - `cp source destination` Creates a copy of the source under the name of the destination.
  - `cp -r source destination` Same, but recursively.
- `mv` (*move*) move a subtree of the filesystem :
  - `cp source destination` Move the source to the destination.

Most commands will accept a `--help` and/or `-h` argument and react to it by printing out a help message instead of completing the operation it was designed to do. These arguments are also called flags.

2.4.6 Working with Files

Files are stored in a filesystem that is usually persisted on a disk. This works out very well when you reboot your computer and see that your data is still there. That would not have been the case had the files been stored in main memory. However, the computer can only work with data in main memory, so in order to do something with the contents of those files, they will have to be read into memory. And if you want the result of what you end up doing with them to survive a reboot, then you need to store it on the disk again.

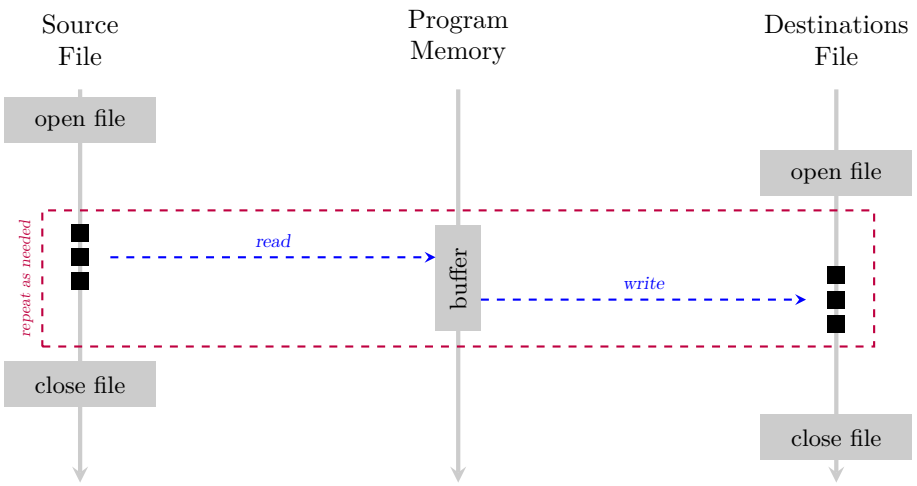


Figure 2.17: The process of copying a file.

2.4.6.1 Copying Files

Creating a copy of a file means to replicate the contents and the metadata of the file in some other directory of a filesystem. That could happen on the same disk or another disk, and that disk could be on the same or another computer.

The principal operation is illustrated in figure 2.17. First the source file is opened for reading and the destination file for writing. This gives the process a file descriptor for each through which it can ask the kernel to perform operations on the file. The process performing the copying allocates a buffer of a reasonable size. Then it reads data from the source file to the buffer. When this operation is completed it writes the contents of the buffer to the destination file. The buffer will likely be smaller than the file being copied (otherwise you would need to have more memory than the largest file), so the process may have to repeated a number of times with different offsets. When this process has been repeated enough times, the source and destination files are closed.

2.4.6.2 Editing Files

When we edit a file, we change the contents of that file. If we add something to the beginning, then the old contents needs to be shifted forward to make space for the new material. If we add something at the end, then some material can be easily added. If the modification happens anywhere in between then part of the original contents needs to be moved. Doing these operations on disk would be highly ineffective. So, in reality it happens in the computers memory and the saving to disk is initiated by either a human intervention or something like a timeout.

Figure 2.18 illustrates how a typical text editor will work with a file. First the file is opened. Then, the contents of the file is read and written to primary memory. The user may then manipulate the in-memory representation through some user interface, and saved the modified buffer (memory representation) to disk. Finally, the program will close the file.

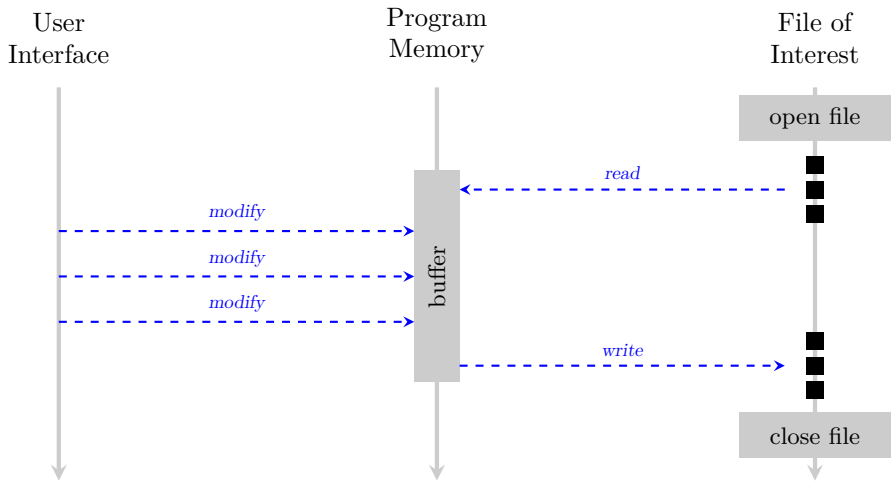


Figure 2.18: The process of editing a file.

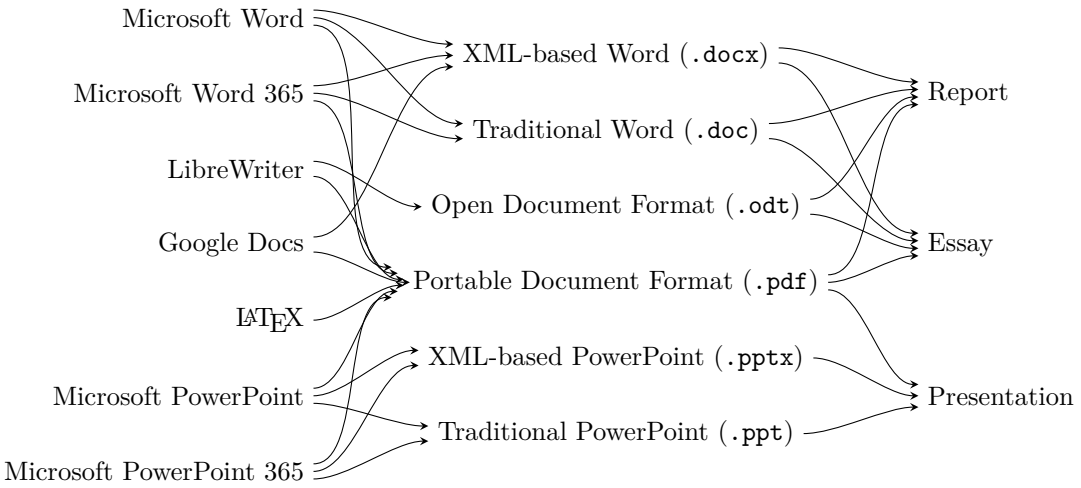


Figure 2.19: The relationship between programs, file formats and functions.

2.5 File Formats

Data is stored in files and they are organized in directories on in a filesystem. Files are organized in more or less standardized formats. Any program that understands a format can work on files that follow this format. Some formats can be used for different purposes. For instance, a PDF file can be used for presentations as well as reports, and many different programs can produce such files. Figure 2.19 lists combinations of programs, the files they can manipulate, and the different uses of these files. Be careful not to mix these three concepts (program, file format, purpose): A presentation is only a *powerpoint* if it is in a `ppt` or `pptx` file format, and if someone requests a PDF presentation then a powerpoint simply won't do!

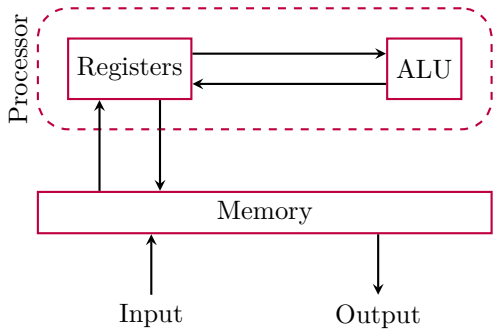


Figure 2.20: Model of computer.

2.6 The Machine

Figure 2.20 shows the model of a computer that we will rely on. Input (e.g, from keyboard) ends up in the computers main memory. The processor, as the name implies, *processes* instructions. Each processor architecture or type supports some instruction repertoire. In that repertoire you will find instructions for load values from main memory into registers. Registers are named values that the processor can instructions can access directly. Other instructions performs operations on registers (e.g., for adding two values). Such instructions are processed by something called an arithmetic logic unit (or ALU in daily speech). Yet other instructions can store the value of a register to main memory. We also say that we "write to memory". Storing values to certain portions of memory yields an output. For instance, some computers have a frame buffer in main memory that defines what ends up on your screen. Writing to that part of the memory will affect what you see on your screen.

There are many different processor architectures. Popular ones include AMD64, ARM and RISC-V. In this book, we will look into the integer side of the 32-bit RISC-V architecture (aka the RV32I instruction set). This is a simple instruction set and architecture, and the model of this is transferable to the other processor architectures for the purposes of this book.

2.6.1 Memory Model

The memory of a computer is laid out in a sequence of bytes. The first byte has address zero, the second one has address one, and so on. Most values take up more than a single byte. If a value takes up address four through seven then we refer to its position through its lowest address (i.e., four).

The memory is laid out as illustrated in figure 2.21. At the lowest memory segment we have reserved data. Among these, we have memory-mapped input and output. Every time we write to an output, it triggers some electrical signal to be generated to a peripheral to the processor. Every time we read from an input, a value is *read* from some peripheral to the processor. Then we have a text segment that contains our program code. This is the instructions that tells the processor what to do. The next segment contains static data that is available to the processor, and the remaining memory is used for dynamic program data. This is the *working area* of the processor.

As our program executes, it produces temporary data. This needs to be stored somewhere. In

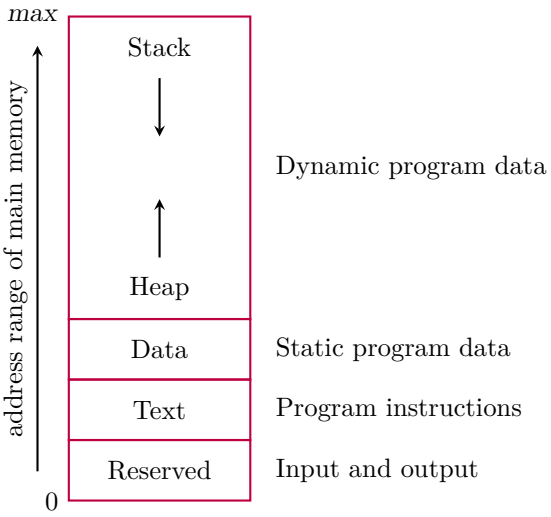


Figure 2.21: Memory model.

practice, it is stored on a heap that is placed at the lower addresses of the dynamic program data segment. Throughout the lifecycle of the program, this datastructure will grow to higher addresses when more space is needed, and shrink when the highest addresses are no longer needed. It is up to the program to to administer this. In C#, however, a so-called garbage collector will do this for you.

Another data structure grows from the other end of the dynamic program segment in the other direction. This way, we won't run out of memory until all the memory is in use. This datastructure is called a stack, and because it is so central to the operation of the computer it is referred to as *the* stack. The stack is used to store data when groups of instructions *farm out* part of their function to other groups of instructions. More about this in the next section.

2.6.2 Registers

RV32I has 32 registers (plus a something called a program counter that we will cover in the next section. The 32 registers are covered in figure 2.22. They are named `x0` though `x31` but also have aliases named after their purpose (the ABI name). The "Saver" column is covered in section 2.6.3.3. The `zero` register is always zero. It cannot be overwritten. We will not cover the remaining registers in detail. But for now, let's just say that they are 32 bit values that instructions can read from and write to.

2.6.3 Instructions

The instruction repertoire of the majority of instruction sets is fairly large, but few of them adds to the understanding of the fundamental operation of the processor. To keep things somewhat simple, we stick to covering a fairly minimal subset of the instructions.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired to zero	—
x1	ra	Return address	caller
x2	sp	Stack pointer	callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	caller
x6-7	t1-2	Temporaries	caller
x8	s0/fp	Saved register/frame pointer	callee
x9	s1	Saved register	callee
x10-11	a0-1	Function arguments/return values	caller
x12-17	a2-7	Function arguments	caller
x18-27	s2-11	Saved registers	callee
x28-31	t3-6	Temporaries	caller

Figure 2.22: Register overview.

2.6.3.1 Memory Access

The processor can only operate on registers, but has a very limited number of them. For 32 bit RISC-V, there are 32 of them. That is only enough for the smallest of programs, and nothing that can be written in C#. Instead, most program data resides in main memory. This means that we need a way of loading data from main memory into registers so that we can work on it, and then another way of storing the results back into main memory to make (register) space for other data. This way, we can continue to do useful work.

We have a number of instructions for loading and storing data. In this description, we only care about those that operates on *words*. A word is the type of integer that matches the size that the processor is optimized to work on. In our case, it is a 32 bit integer.

- `LW` ("load word") This loads a word from an address in main memory into a named register in the processor.
- `SW` ("store word") Stores the value of a named register in the processor to an address in main memory.

These addresses are the sum of the value of a register, and a 12-bit integer that is encoded in (read: part of) the instruction itself. That last part acts as a static offset to the value in the register.

2.6.3.2 Arithmetic Operations

The processor implements instructions for a number of arithmetic operations. The classical ones are addition, subtraction, multiplication and division. These make up a class of instructions that take two input values and produce a single output value. There are other operations that fit that pattern and have value for programmers. Boolean operations that you will get familiar with in section 5.4 are examples of these. The processor supports a number of these through instruction.

There is a generic variant of these instructions that maps each input and output to a register. That is, it takes inputs from two registers, and writes the output to a single register. These three

registers could potentially be the same. Often these operations have a fixed operand though. As registers are a scarce resource, specialized variants – that have the fixed value encoded directly in the instruction – are available for some of the operations. Such fixed values are then called immediate values. Because of their lower register footprint, the compiler (that turns the human readable source code into instructions) can do a better job.

As these instructions look very similar, we will only cover a few:

- `ADD` ("add") Add the values of two registers and store the resulting value in a third register.
- `ADDI` ("add immediate") Add an immediate value to the value of a register and store the resulting value in a second register.
- `MUL` ("multiply") Multiply the values of two registers and store the resulting value in a third register. No immediate variant exist for this operation.

In this book we will only cover integers to this depth. Similar instructions exists for floating point numbers that you may know as decimal numbers. The binary representation of the numbers is much more complicated, and this brings some intricacies. These are detailed in section 5.3 when we cover floating point numbers in C#.

2.6.3.3 Choice

Instructions are by default evaluated in the order they are laid out in memory. Special *branch* instructions can be used to deviate from this rule. Some are unconditional. That is, they *jump* to a specific memory address when evaluated. Others are conditional. That means that they only jump if a certain condition is met.

Unconditional branch instructions:

- `J` ("jump") Jump to a specific memory address.
- `JAL` ("jump and link") Jump to a specific memory address, and *link*.

But, what does it mean to link? Instructions are often arranged in groups. These are the equivalent of what you will come to know as functions. They operate on some registers and can use the `JAL` instruction to transfer execution to another group of instructions. That group of instructions can then solve a part of the problem, place the result in a specific register (or at a specific memory address), and then transfer execution back to the first group of instructions. We say that the first group calls the second. This makes the first group take the role of the caller and the second group that of the callee. To make sure that there is no conflict in register use (e.g., that the callee overrides a register that is in use by the caller) a convention is established for which role needs to save the contents of a register before overriding it (and restore it before returning execution). This can be seen in figure 2.22.

Conditional branch instructions (no link):

- `BNE` ("branch not equal") Jump to a specific memory address if two register have different values. Otherwise continue.
- `BLT` ("branch lest than") Jump to a specific memory address if one register has a value that is less than that of another one. Otherwise continue.

2.6.4 Big Picture

The components that of the machine that we have covered can be put together around the central processing unit (i.e., the CPU). An implementation of a primitive RISC-V processor is illustrated in figure 2.23. In reality, it is much more complicated, and RISC-V is a simple processor architecture. Let's briefly go through what is going on in this figure.

The diagram consists of various blocks connected by arrows. These blocks are complex. That is, they represent a logical unit but are physically constructed from smaller parts … with arrows between them. We call the arrows signals. They represent a value of one or more bits. One can think of them as how data flows through the processor. Concretely, this is accomplished through register transfer level (RTL) logic and a clock signal. But then things get really complicated, and we don't need it that badly.

Some of these signals exists to move data around between memory, registers and the implementation of the instructions. These are called data signals, and we say that these make up the data plane. This is a cross-cut of the processor through which "data flows". Other signals are control signals, and they make up the control plane. That is a different cross-cut that informs the blocks of *how* they should operate. For instance, the block that executes the instructions is called an ALU. It has a number of generic inputs and a control signal tells it which operation (e.g., an addition or a division) it is supposed to perform on these.

There are three ALUs on the diagram. They look a bit like an arrow pointing to the right and are labeled either "ALU" or "+". The latter version is a specialized version of the former. The specialized version takes two data inputs and produces a data output. This output is the result of adding the two inputs. The generic version has a control input which tells it which operation to perform. This ALU supports a large number of operations, and will perform the operation specified by the control input on the data inputs and output the resulting value on the data output. For arithmetic operations, this is straight forward to imagine: If the control signal specifies a multiplication operation and the data inputs a two and three, then the data output becomes six. Other operations are used to transfer the control flow from one place in instruction memory to another. This can be either conditional (e.g., using the BNE or BLT instruction) or unconditional (e.g., using the J or JAL instruction). If a branch operation needs to be executed, the ALU will indicate this using its control output signal.

If we follow that control signal from the ALU, we will find a shape labeled "AND" that on the input side looks like a rectangle and on the output side it looks like a circle. It is an AND gate. That is a piece of digital electronics that produces a signal that represents whether all of the inputs are true.

In the diagram, you will find three multiplexers (or mux for short). These are labeled "mux". They each have one outgoing edge, two incoming black edges, and one incoming *colored* edge. The mux is a selective forwarding mechanism, based on the value of the colored input, it forwards the value on one of the black inputs to its output. For this reason, the colored input is called a control signal.

2.6.4.1 Data Plane

The execution of an instruction starts at the program counter (or PC, for short). It is usually incremented in steps of four as this matches the size of a RISC-V 32-bit instruction. The program counter is a register that holds the address of the next instruction. This address is fed into the program memory block.

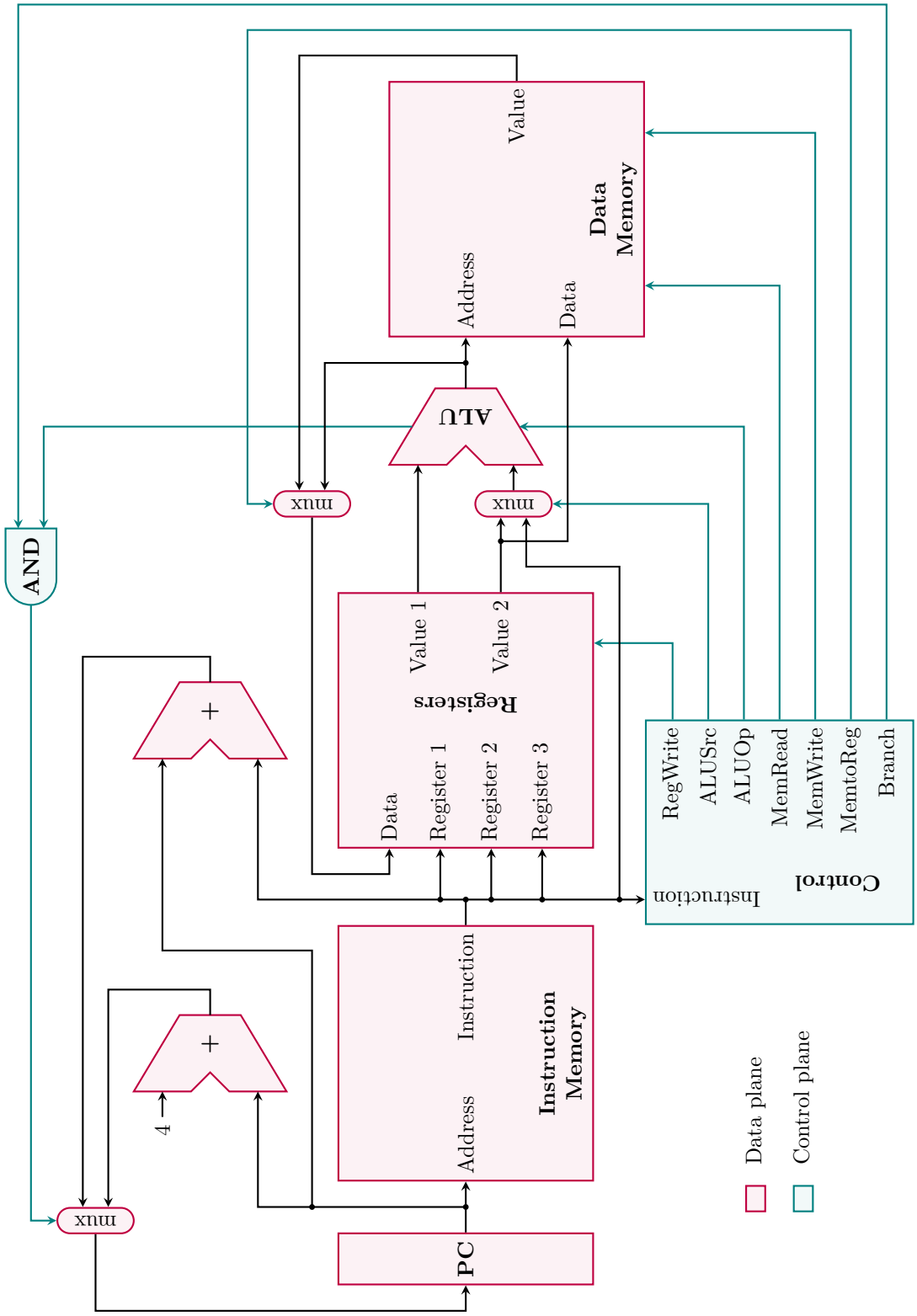


Figure 2.23: Primitive model of a RISC-V Processor.

The program memory block then fetches the instruction that the program counter points to. It then decodes it into it principal components. These are the code for the operation (aka the instruction identity), and the relevant register numbers and/or immediate values.

Depending on the concrete instruction, up to two of those registers are used as input to the operation. So the next step is to read the values of these registers from the register file. The register file is the set of registers physically present within the processor. These values are produced at the labels "Value 1" and "Value 2" in the diagram. If only one of them is needed, then the other will be undefined. That means that the architectural model of the processor does not dictate what any physical processor should do in this case. While this may seem like a problem, the control signals (that we will cover in section 2.6.4.2) will make sure it does cause any trouble.

Data flows through the *main* ALU. The first input comes from a register, but the second comes – depending on instruction – from either a register or an immediate value encoded directly in the instruction. The control model tells it which operation to perform. Many instructions will require the result of the operation to be written back to the register file, in which case the control module will make sure this happens.

Other instructions will need to store the value of some register in memory, or load a value from memory into a register. The data memory is responsible for those load and stope operations. The data involved in a store instruction will essentially bypass the main ALU by feeding "Value 2" for the register file into "Data" of the data memory. The address is calculated by the ALU though.

In a load operation, the "Value" output of the data memory block (which represents the data at the input "Address") is fed into the "Data" input of the register file. A mux on the way will have to be instructed by the control block to route the data signal appropriately.

2.6.4.2 Control Plane

The processor contains a clock generator. This is essentially a metronome for the machine. Every time it ticks, a new instruction is processed[1]. The clock signal is the original control signal.

At each clock cycle, the ALU receives two data inputs. An "ALUOp" control signal informs it of which operation to perform on these inputs. For branch instructions, the ALU has a control output that dictates whether the a jump should be initiated. For a unconditional jump instruction that control output will always indicate to initiate the jump. For a conditional jump (i.e., a branch) it will depend on the condition of the jump.

Let's find the program counter again. This is the register that holds the address of the current instruction. The execution of a non-branch operations is followed by the execution of the instruction stored at the following address in memory. Since all instructions are 4 bytes long (i.e., they take up 4 bytes of space), this next instruction resides at a memory address that is 4 higher. This, you can see implemented using a "+" ALU in the beginning of the diagram. However, a mux placed in this data pathso that this *normal* operation can be bypassed in case of a jump. In those cases, it is not 4 that is added to the address, but an immediate value that is encoded in the instruction. This can be seen in the other "+" ALU. Whether to jump is determined by a control signal that comes from the AND gate, and that states that we jump if and only if the control blocks "Branch" signal and the ALUs output signal agrees that a branch should be taken.

[1] While this is true in our simplified setup, real processors are much more complicated. This is, in part, how they achieve a high execution speed.

From the perspective of the data memory, there are three kinds of instructions, namely (i) those that have no relevance it it, (ii) those that read data, and (iii) those that write data. In the first case, there is no need to stress the memory and nothing should be done. In the second case, an output data signal should be produced that contains the read value. This is indicated by the "MemRead" control signal. In the third case, the contents of the data memory should be updated according to the blocks data inputs. Under no other circumstances is it acceptable to write to this memory. This is indicated by the "MemWrite" control signal.

Similarly to the data memory, the register file can be read from and written to. What to do depends on the instruction, and it is the job of the control block to decode the instruction and … well … *control* the register file. The register addressed on the data inputs "Register 1" and "Register 2" are always read and outputted to "Value 1" and "Value 2". The "RegWrite" control signal controls whether the "Data" input should be written to the register addressed by "Register 3". The "MemtoReg" control signal determines which data output is fed into the "Data" input of the register file: Should it be the output of the ALU (e.g., as in a plus instruction) or should it be the output of data memory (e.g., as in a load instruction)?

2.6.5 C# Link

This machine model is slightly deceptive in the sense that we – in this book – won't be programming the processor itself. A special program called a kernel is running on top of the processor. This program orchestrates a number of programs that make up the operating system. When we execute the C# programs that we make throughout this book, we do so through a special program called a virtual machine that runs on this operating system. All of these programs, except for the kernel, exists in the dynamic program data memory segment. The virtual machine translates the CIL instruction of your compiled C# code into instructions that can be executed on your physical processor.

2.6.6 Takeaways

You will find that many of the design choices of C#, and any other programming language for that matter, have their roots in this machine. There is a reason why these languages look like the do. At the end of the day, they have to be executed on a physical processor, and often there is really only one way of doing something on such a processor that doesn't incur a huge performance penalty.

You likely have found this section difficult to follow. That's okay. As you progress though this book, do yourself the favor of returning to this section. It should be easier to read by then.

2.7 Programming Languages

> *"It might seem easy enough, but computer language design is just like a stroll in the park. Jurassic Park, that is."*
>
> — *Larry Wall[12]*

In order to build a program, we describe *how* that program should work or *what* it should do. This is done according to the formalized rules of a programming language. Most such languages will have high-level abstractions (compared the the assembly instructions known to the machine). They are designed in such a way that a human can reason about them. Those descriptions are

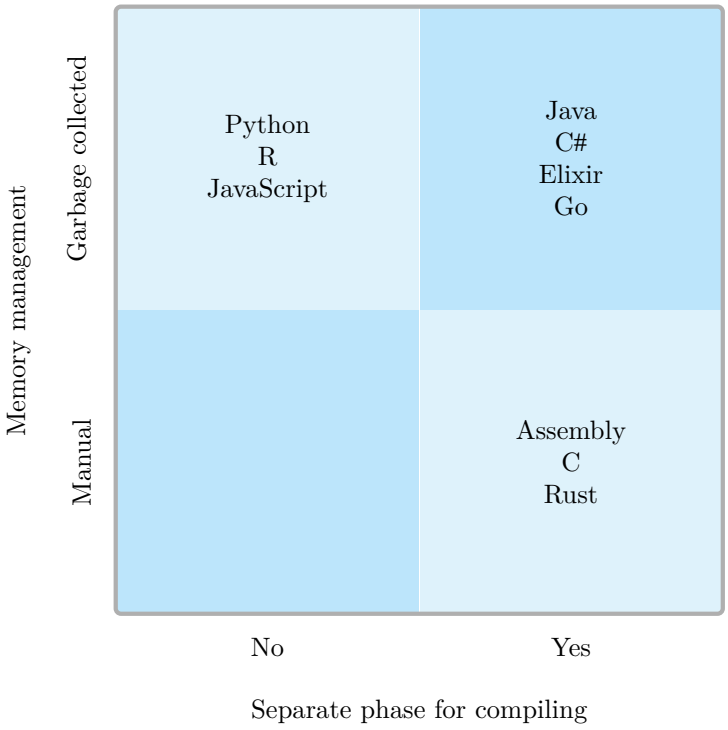


Figure 2.24: Categorization of programming languages.

stored in files and referred to as source code. Later on, that source code in converted into instructions and executed.

There are many programming languages to pick from, and they each have strong and weak points. When faced with a particular problem it is important to pick a language that have relevant strong points and acceptable weak points. Many properties can be used to classify these languages. Figure 2.24 focuses on two properties, namely whether there is a separate phase for compiling and whether a garbage collector is employed.

As programming languages are designed for humans to write, and the machine only understands assembly instructions, something has to compile the language into these instructions. Technically, this is always a separate step, but that can be more or less obvious to the user. Some compilers will give the option of compiling the code when you try to execute the code. So, the boundary is a bit fluid. Here, we are using whether the compiled code is stored on disk after compilation as the arbiter. Languages where this is not the case are often used for tying together workflows involving many processes and files. These are called scripting languages. C# is a compiled language and not a script language (even though Microsoft likes to talk about C# scripts).

Throughout the lifecycle of a process, it will need temporary chunks of memory to do its work. These chunks can be of different size and the need for this can come at varying frequency. We say that the kernel allocates memory to the process on demand. This ensures that the process has exclusive access to this piece of memory and thus that no other process can interfere with its business[2]. Once the process is done with this piece of memory it is deallocated (or freed) so

[2]In reality this can be be avoided through shared segments of memory, but that is a topic way too advanced

that it can be reclaimed by the kernel and put to use later on. This process of allocation and deallocation is manual in some languages and automatic in others. When automatic, a garbage collector component is compiled into the running code. It essentially scans all allocations and deallocate those that are no longer in use. Again, there are pros and cons with each solution. C# is a garbage collected language.

2.7.1 Parsing

The contents of a file is a sequence of bytes, i.e., units that each have one of 256 possible values. When we write text, these bytes are interpreted as characters and the programs we use to open such files choose to draw each character to the right of the previous one. At line breaks – represented by a specific sequence of characters – the horizontal position is moved all the way to the left and the vertical position is moved one line height down. Some programs break lines that are too wide for your screen. But that's about it. Such a text file does not contain any information about font or text size. If you want to store such information, you have to introduce a file format that encodes such information by special markup. But this is not relevant for files that contain code.

Files containing code are raw text files: A sequence of characters with line breaks. As programmers, we manipulate these files through a text editor, which most often – in addition to displaying these characters – chooses to color-code selected subsequences to make the text more readable for humans. However, these colors represent an *interpretation* that the tool uses to enrich the code that is actually stored. Modifying the file and saving it will not cause the interpretation to be stored, but only the underlying code. Similarly, there are some such editors that number the lines, highlight a current line, and mark matching parentheses.

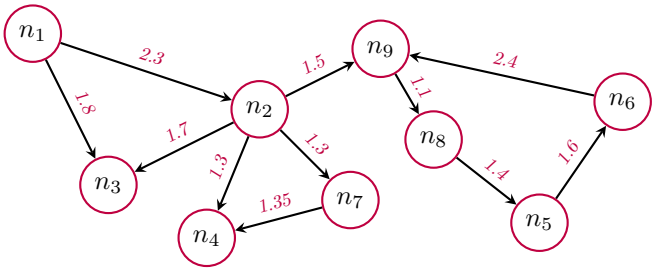
The text these files contain is called *program code* (or simply *code*), and we change it in much the same way as we change a written report. But when we ask the computer to execute our program code, it sees the file in a different way. The content is of course the same, but the *interpretation* is different: Based on some rules, the computer builds a tree structure of the code. This process is called parsing the source code.

2.8 Exercises

Exercise 2.1: Terms

Topic
Creativity

Consider the following graph:



1. What type of graph is this?

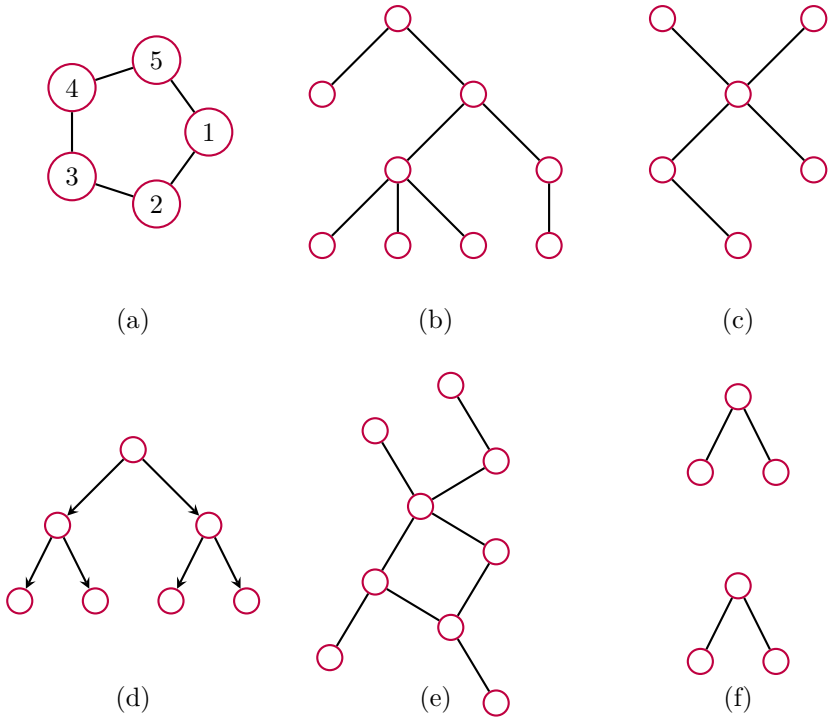
for this book.

2. How many nodes are there?
3. How many edges are there?
4. Is there a cycle in the graph, and if so, how long is it?
5. How far are there from n_8 to n_4?

Exercise 2.2: Tree Structures

Topic
Creativity

Which of the following are tree structure?



Exercise 2.3: Ancestry

Topic
Creativity

Draw a graph of the ancestry of your nearest family, or a fictional family from a movie, show or book.

Which graph-related terms do you choose to describe which relationships?

Exercise 2.4: Text Editor

Topic
Creativity

In this exercise, you will have to install a text editor on your computer. This is a program that edit the characters in a file. There are lots of options, and the choice is not particularly important.

The only thing that matters is that it is *not* an IDE. That is, you must not have access to features such as:

- Automatic code completion (e.g., IntelliSense).
- A debugger.
- Index of variables and other names.
- Built-in generative AI.
- Built-in suggestion engine.

If you already have experience with `vi` or `emacs`, you are all good. If not, then you should avoid them at this point in time as their learning curve is quite steep.

Here are some options (ask your instructor for more):

- ***Windows***:
  - Kate (without LSP)
- ***OSX***:
  - Kate (without LSP)
- ***Linux***:
  - Zed (install C#extension and turn most options off)
  - Gedit
  - Kate (without LSP)

So, you need to:

- Pick one of the above text editors.
- Figure out how it should be installed. This may require some googling.
- Install it.

Exercise 2.5: File Creation

1. Open a text editor and save a blank file anywhere.
2. Open a file manager, and locate the file. In this exercise, you should only use the file manager to locate the file; don't use it to open it.
3. Open a terminal, and change the *current directory* to match the location you found in the previous step.
4. List all files in this directory. How large is the file?
5. Go back to the text editor, write the text "Hello, World", and save the file again.
6. How large is the file now?

7. Explain the specific difference in size.

Exercise 2.6: Directory

Topic
Creativity

1. Open a terminal, and go to the directory of the previous exercise.
2. Using that terminal, create a subdirectory called “oop”.
3. Still using the terminal, move the file from the previous exercise into the “oop” directory.
4. Use a file namager to verify that you have accomplished the task.

Exercise 2.7: GitHub Account

Topic
Creativity

- Create an account on github[3].
- Create an “SSH key”[4].
- Register your SSH key with your github account[5].

Note: You are likely to use this account thoughout all of your studies. It will end as a portfolio for potential employers. Reconsider infantile account names.

Exercise 2.8: Git Installation

Topic
Creativity

In this exercise, you will install the GIT command line tool. The process is different for each of the major operating systems.

2.8.0.1 Linux

As a Linux user you should be able to figure this out by yourself. But on a Debian-based system (e.g., Ubuntu) the following command (when executed as a privileged user) will do the trick:

```
apt-get install git-all
```

2.8.0.2 OSX

Follow the instructions on github[6].

[3] https://github.com

[4] https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

[5] https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

[6] https://github.com/git-guides/install-git#install-git-from-an-installer

2.8.0.3 Windows

As we are using a Debian-based WSL in this book, you should open a WSL shell and follow the instructions for Linux.

EXERCISE 2.9: GIT SETUP

Topic
Creativity

Find the section called "First-Time Git Setup" in the Pro Git book[2]. Read it, and then execute the commands for setting up your identity (i.e., name and email), and editor[7].

Then run

```
git config --list --show-origin
```

This command lists those things that GIT knows about you, and what the sources for this information is. Find the name of the file containing`user.name`. Open – via the command line - this file. Which other information does it contain?

Note: If the printouts of the above command does *not* mention `user.name` then the first part of the exercise hasn't been correctly completed.

EXERCISE 2.10: REPOSITORY CREATION

Topic
Creativity

In the following instructions you will find a number of URLs. These are specific to the github user "aslakjohansen". Replace this part with your own user name.

As so many other websites, the interface of github is constantly changing. Because of this, you should expect to have to deviate from the following instructions:

- Go to the page for your account on GitHub: https://github.com/aslakjohansen/.
- In the top right corner you will find a drop-down menu marked with a "+". Click this and choose "New repository". Name the repository "progintro-exercises" and make a decision about who should be able to see your work on this repository: public or private. This visibility can be changed later on. The remaining default values are fine. Click "Create repository".
- You have now created a repository. Go to the page for it (https://github.com/aslakjohansen/progintro-exercises/). Click the green "<> Code" button, and choose "Local" followed by "SSH". Here, you copy the URL (git@github.com:aslakjohansen/progintro-exercises.git).
- Now, you have to decide where in your computers filesystem you want to store the local copy of this repository. This directory can be moved later on.
- Open a terminal and change directory to this location. Run the git command for cloning the repository: `git clone git@github.com:aslakjohansen/student-projects.git`.
- Verify that a `progintro-exercises` directory has been created.

[7] If in doubt, "nano" isn't the worst choice in the world.

Exercise 2.11: Markdown

Topic
Creativity

In the root of your local clone of the `progintro-exercises` repository, create a file called `README.md` using your text editor. Make sure the contents of the file looks like this:

```
# My Exercises

Useful links:
- [Githubs support for
↪  Markdown](https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax)
- [Is it Christmas?](https://isitchristmas.com)

## Formatting Examples:

1. *Italic* text.
2. **Bold** Text.
```

This is a markdown file. Markdown is an example of a markup. That means that it can give textual elements simple typographical properties.

What do you think this file will look like on githubs page?

Hint: Try to add the file, commit the change, and push the change to github.

Exercise 2.12: Commit Log

Topic
Creativity

List the commits of one of your repositories:

```
git log
```

The same can be seen on github. Where?

Which information is listed for each commit?

Exercise 2.13: Changes

Topic
Creativity

In your `progintro-exercises` repository, open `README.md`. In the top of this file, you will find a few links. Add a new link of your choice to this list. Save the change, commit the changes and push the new commit.

Verify that then push has made its way to the web-interface of github.

Exercise 2.14: Conflicts

Topic
Creativity

In this exercise, we raise the complexity a bit.

1. Make sure to check out the `progintro-exercises` repository in two locations of your filesystem. This is a way of simulating that multiple software developers are working on the same repository. Each have their own working copy. In the following text, we are going to refer to these as directory A and directory B.
2. Go to directory A, are replace the work “text” in each of the two last lines with the word “emphasis”. Commit the change, and push it.
   - What would be a good commit message?

3. Go to directory B, and translate the last two lines to a (human) language of your choice. For Danish, this would be:

```
1. *italics* tekst.
2. **bold** tekst.
```

4. Then save the file, commit the change and push the commit.
   - What would be a good commit message?
5. If you have followed these istructions, you should have gotten an error message. Destribe what has happened, and try to draw it as a graph of commits.
6. In order to solve the problem, we need to decide how to resolve the situation:
   a) Are we going to keep the version from B, and delete everything from A?
   b) Are we going to keep the version from A, and delete everything from B?
   c) Can we do a rewriting that honer both intentions (change of noun, and translation to different language)?
7. Argue your choice.
8. Make the necessary change, commit it, and push the commit.
9. Verify on github that things are now in order.
10. Check that your understanding of what has happened by interpreting githubs "network graph". You may have to search a bit for it.

Exercise 2.15: Repository Structure

Topic
Creativity

Let's imagine that you will be using the `progintro-exercises` repository to store the solutions to the exercises in this book. Under that premise, make a plan of how to organize your solutions within this filesystem.

Subtask 1

Skim through this book, and try to describe how the exercises are organized.

Subtask 2

Assume that all future exercises follow this same structure. How can you mirror this in your repository?

Chapter 3

Prereqs

Before we get started on the subject matter, we need to install a number of programs that the remainder of the book will be referring to. You will need to use these (or similar) in the exercises. This may not seem like a particularly glorious task, but it is important to know the system in front of you and tasks like these are common for a software developer.

3.1 Terminal and Shell

3.1.1 Installation

3.1.1.1 Windows

3.1.1.2 MacOS X

3.1.1.3 Debian Linux

3.1.2 Verifying Success

3.2 Text Editor

While any text editor would do, not all are created equal. In particular, not all are equally suited for beginners. Some features help you think about the code, and understand it. Such features include syntax highlighting (the highlighting of different parts of the code according to their role), line highlighting and paranthesis highlighting. Other features aim to maximize the rate of code production. Such features typically allows you to be very "productive" without having to think a whole lot about the code. This has a tendency to lead to absent-mindedness and that is especially problematic when learning how to program. When working with the subject, you need to be present in mind and think about code. Always think about what makes it tick! Features that detracts from this are undesirable. These include anything that provides you with suggestions on how to write or rewrite your code. Any kind of autocomplete functionality – be it rooted in generative AI or otherwise – diverts your thought process from solving the problem proactively. That is bad. Other editors are very keen to suggest minor rewrites according to some set of more or less well-intended guidelines. However, these guidelines were rarely intended with novice developers in mind. In fact, then often hide complexity from the developer that is critical for the development of the novice developer.

So, we are looking for a text editor that has the positive aids and none of the negative aids (or at least provide easily available ways of disabling them). Seasoned developers may suggest `vi` or

`emacs`, but they use their own shortcuts that have gone out of fashion. If you already use them, then there are no problems sticking to them. But everyone else should move on. Developers with deep roots in the industry will cling to Visual Studio (which is not portable) and have strong opinions on Visual Studio Code. Those opinions can go either way. Some companies are beginning to discard applicants who prefer Visual Studio Code as they have observed such developer to have a tendency to be way too reliant on the aids it provides. Rider is a popular option but recently they have gone all in on the problematic aids. This leaves a number of options. If you are on Linux, then you could use `gedit` or `kate`. If you are on Windows, then perhaps `notepad++` is a good option. However, the `zed`[1] editor is a good option, and it is available for all three major platforms. So, the following instructions will target `zed`. `zed` can be a bit finicky though, especially with regards to GPUs. So, if you are experiencing trouble, install one of the others.

3.2.1 Installation

3.2.1.1 Windows

3.2.1.2 MacOS X

3.2.1.3 Debian Linux

3.2.2 Verifying Success

3.3 GIT

3.3.1 Installation

3.3.1.1 Windows

3.3.1.2 MacOS X

3.3.1.3 Debian Linux

Git is available in the standard Debian repository. Install the client by running the following command as a privileged user:

```
apt install git
```

3.3.2 Verifying Success

At this point you should be able to run the `git` command without any parameters and get a description of how to use it.

[1] https://zed.dev

3.4 C# Development Environment

3.4.1 Installation

3.4.1.1 Windows

3.4.1.2 MacOS X

3.4.1.3 Debian Linux

3.4.2 Verifying Success

3.5 Livebook

3.5.1 Installation

3.5.1.1 Windows

3.5.1.2 MacOS X

3.5.1.3 Debian Linux

First, install the `asdf` version manager. To do so, go to `https://github.com/asdf-vm/asdf/releases` and download the latest version for your architecture. For version 0.16.7, that should be either `asdf-v0.16.7-linux-amd64.tar.gz` or `asdf-v0.16.7-linux-arm64.tar.gz`, depending on whether you have an Intel/AMD or an ARM processor. Let's assume that you downloaded the `asdf-v0.16.7-linux-amd64.tar.gz` file to `/Downloads`. Then do

```
cd ~/bin
tar -xzf ~/Downloads/asdf-v0.16.7-linux-amd64.tar.gz
```

Now that `asdf` is installed (and in your path), we need to add plugins for Erlang and Elixir:

```
asdf plugin add erlang https://github.com/asdf-vm/asdf-erlang.git
asdf plugin add elixir https://github.com/asdf-vm/asdf-elixir.git
```

Install Erlang using `asdf`:

```
apt install libssl-dev
apt install unixodbc-dev
apt install libwxgtk3.2-dev libwxgtk-webview3.2-dev
asdf install erlang 27.3.2
echo export ASDF_ERLANG_VERSION=27.3.2 >> ~/.bashrc
source ~/.bashrc
```

Install Elixir using `asdf`:

```
asdf install elixir 1.18.3-otp-27
export ASDF_ELIXIR_VERSION=1.18.3-otp-27 >> ~/.bashrc
source ~/.bashrc
```

This should allow you to run the Elixir REPL called `iex`. You can exit it by pressing Ctrl-C twice.

Now, let's install Livebook in your home directory[2]:

```
cd
git clone git@github.com:livebook-dev/livebook.git
cd livebook
mix deps.get --only prod
```

Finally, let's create a script that makes starting Livebook a breeze. For this you should come up with a password. In the instructions I use the password "`gazelle`"

```
cd ~/bin
touch start-livebook
echo \#\!/usr/bin/env bash >> start-livebook
echo export LIVEBOOK_PASSWORD=gazelle >> start-livebook
echo export MIX_ENV=prod >> start-livebook
echo cd ~/livebook >> start-livebook
echo mix phx.server >> start-livebook
chmod u+x start-livebook
```

You should now be able to start Livebook by running:

```
$ start-livebook
...
[Livebook] Application running at http://localhost:8080/
```

The first time you do this, it will take a while as the application needs to be compiled. When the last line appears, you can point a browser to the URL. If it connects and asks for a password, you have successfully installed Livebook.

3.5.2 Verifying Success

[2] You probably want to place it somewhere else according to your own organizational concept, but the instructions will assume this location.

Part I

Imperative Programming

Chapter 4

The First Program

> *"When the first caveman programmer chiseled the first program on the walls of the first cave computer, it was a program to paint the string 'Hello, world' in Antelope pictures. Roman programming textbooks began with the 'Salut, Mundi' program. I don't know what happens to people who break with this tradition, but I think it's safer not to find out."*
>
> — *The Linux Kernel Module Programming Guide[11]*

4.1 Phases

C# is an ahead-of-time compiled language. This means that program execution is a step separate from program compilation. Or, to skip the technical terms, there is a compilation phase whereby a special program known as a compiler produces a program binary by processing the program source code. This binary is transferable and (fairly) portable. This means that it can be moved to a different machine and executed without the presence of a compiler. But let's take a step back a take a look of all the involved steps.

4.1.1 Project Creation

First, let's create a directory that can host our project:

```
aslak@gaia:/tmp$ mkdir new_project
```

Then, we can go to that directory and initalize a new console project:

```
aslak@gaia:/tmp$ mkdir new_project
aslak@gaia:/tmp$ cd new_project
aslak@gaia:/tmp/new_project$ dotnet new console
The template "Console App" was created successfully.

Processing post-creation actions...
Restoring /tmp/new_project/new_project.csproj:
  Determining projects to restore...
  Restored /tmp/new_project/new_project.csproj (in 146 ms).
Restore succeeded.
```

So, what happened? Let's take a look at what happened in our newly created directory:

```
aslak@gaia:/tmp/new_project$ ls
new_project.csproj  obj  Program.cs
```

Three files were created, and one of them is a directory:

- `new_project.csproj`: This is a "project" file. It holds metadata that tells the `dotnet` command how to build the project. The build process is the process that converts our human-readable source code to a binary that can be executed on a machine.
- `obj`: For now, we don't care much about this directory.
- `Program.cs`: This is our program file. It is where we write our code. Later on we will add more files. From the perspective of the `dotnet` command, the name doesn't matter. However, as your project grows, humans will find is very confusing if certain standards are not met. We will cover these throughout the book.

4.1.2 Writing

Open the `Program.cs` in your text editor and make sure the contents is as follows: [extract file]

```
Console.WriteLine("Hello, World!");
```

4.1.3 Saving

Save the file. This persists to data that was present in your computers memory under the control of your text editor to disk. That way another program can easily acces it through a valid path to the file.

4.1.4 Compilation

One such program is a compiler, and that happens to be what we need to build the program binary that is needed for execution. That compiler can be called using the `dotnet` command:

```
aslak@gaia:/tmp/new_project$ dotnet build
  Determining projects to restore...
  All projects are up-to-date for restore.
  new_project -> /tmp/new_project/bin/Debug/net8.0/new_project.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:05.36
```

If we list the contents of the project directory we will see that a new `bin` directory has been created for storing the binary program:

```
aslak@gaia:/tmp/new_project$ ls
bin  new_project.csproj  obj  Program.cs
```

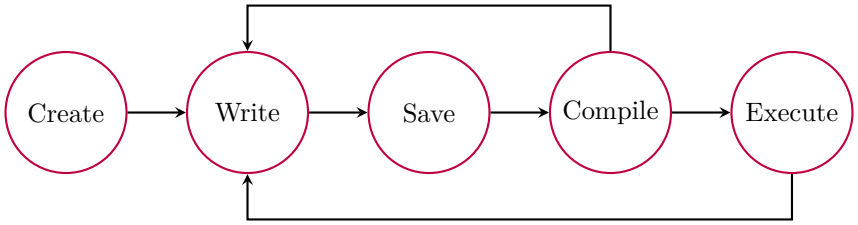


Figure 4.1: Typical developer cycle.

4.1.5 Execution

While we say that the compiler has produced an executable binary for us, it is not a binary that our hardware is capable of running. We need a virtual machine for that. This is essentially a special program that simulates a different architecture. The binary we have contains Intermediate Language (IL) bytecode, and we need a Common Language Runtime (CLR) virtual machine to execute it. Luckely, once again, the `dotnet` command gives us access to such:

```
aslak@gaia:/tmp/new_project$ dotnet run
Hello, World!
```

4.1.6 Big Picture

A typical development cycle looks like figure 4.1: First a project is created, then the developer writes some code that is then compiled and executed. When writing the text editor may provide hints of errors that the developer then takes into account. The compiler may give provide feedback in the form of errors or warnings that need to be addressed. When executing, one might discover bugs that needs fixing. All of this brings us back to the writing phase.

On top of that, developers usually work on a small portion of the intended functionality. When that functionality is deemed correct – as there are no indicators of problems in our *Write*, *Compile* and *Execute* phases – the developer takes on an other portion and starts in the *Write* phase.

4.2 C#

The program from section 4.1.2 looks trivial on the surface, but there happens to be quite a bit more to it that deserves some attention. So, let's give in to it! After all, there are fundamental details at play here that is going to inform our understanding of the C# programming language.

We will start by looking into how C# code is compiled into an executable binary. This happens in four distinct steps as illustrated in figure 4.2. The following subsections will cover the role of each of these phases, and give a very short introduction to the mechanisms behind them.

4.2.1 Lexing

The first thing a compiler will do when it sees some code, is to hand it to its lexer that then chop it up into a sequence of tokens. This is the only job of the lexer. Each token represents a small part of the code. These parts are atomic units with respect to the following phases of

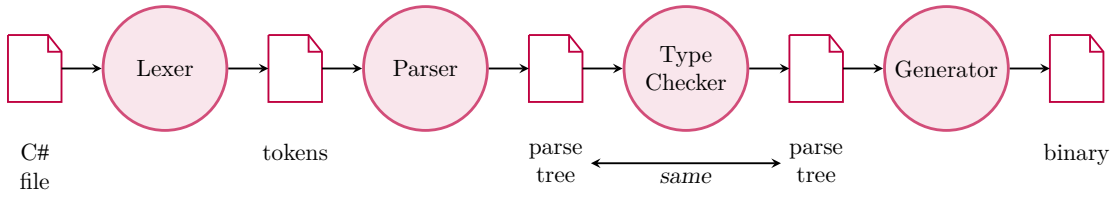


Figure 4.2: Compilation phases of a C# program.

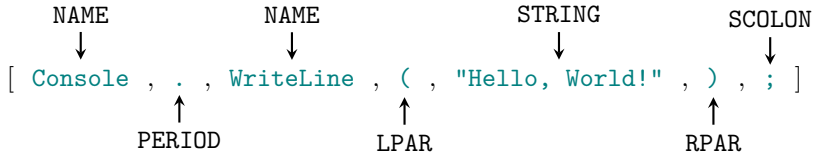


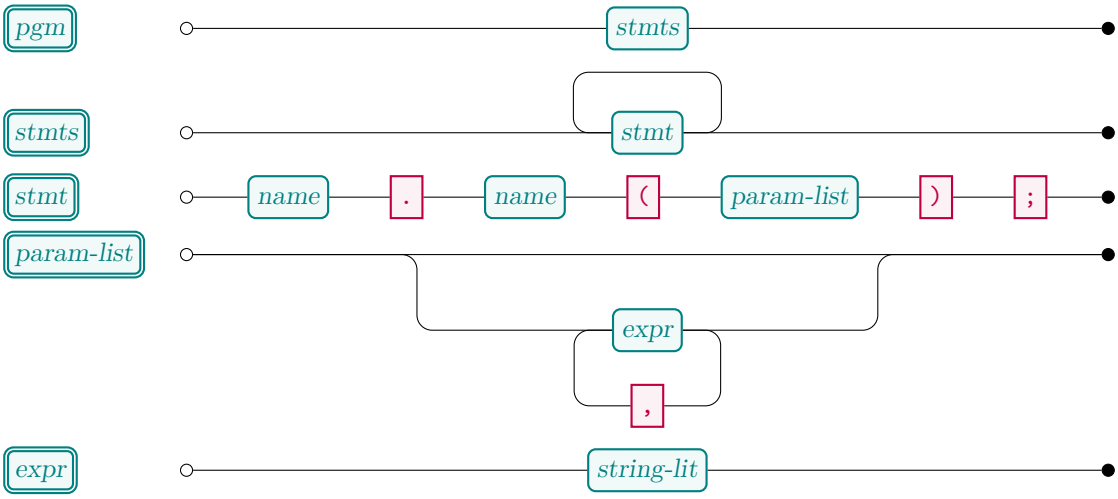
Figure 4.3: Token sequence of the hello-world program produced by the lexer.

the compilation. By this, we mean that there is no reason for any of the following phases to deal with the code at a higher granularity than this. This may seem very abstract at this point, and that's okay. Let's just say that the output of the lexer is as illustrated in figure 4.3. In this figure, each token is annotated by type. Some are simply named after the character they represent. These include `RPAR` for right parenthesis and `SCOLON` for semicolon. Others represent a sequence of characters.

`NAME`, for instance, is our first encounter with the concept of naming. We will dig more into this as we progress through this book. But for now, let's just say that it is a name which can be used to reference *something*. Such names are defined by programmers and referenced by programmers having a shared context. In C# a `NAME` has to start with either an underscore character or any alphabetic character, and the remaining characters have to be underscores, alphabetic characters or decimal digits.

Another example is `STRING` (also called a string literal). This is a sequence of arbitrary characters surrounded by quotes. This is a gross simplification. But for now, we don't have to worry about a single detail. Those remaining worries can wait until section 21.3.

Naturally, we would like to be able to write whatever text we like in such a string, and that include quotation marks. But if we simply place a quotation mark in there, instead of the comma between "`hello`" and "`world`", then the lexer would see this quotation mark as the one terminating the string. And so, a convention has been introduced to solve this problem whereby the backslash character will *escape* the normal meaning of the following character. We say that the backslash character is an escape character. And thus, "`"The \" character is used to wrap strings"`" is a valid string. This, of course, puts us in a similarly awkward position when we want to have a backslash character in our string. But here the same solution applies: We simply escape the normal meaning of the backslash character by writing two backslash characters in a row.



Syntax 4.1: Concepts involved in the basic hello-world program

4.2.2 Grammars

When you read this book, we rely on a shared understanding of the syntax and semantics of the English language. Throughout the book we extend our shared vocabulary with terminology relevant for this particular domain, just like any other textbook. Every language intended for communication between humans has such syntactic rules and a vocabulary. Likewise, every programming language has a grammar. It defines what is syntactically valid code.

When a written English message is used to carry information between humans, the receiving party has the ability to reason about flaws in the *coding* to that message. When I miss a comma, you will still get the message. A typo is not the end of the world. If I use a word you don't know, then you will look it up, or walk away with perhaps 90% of the meaning of the offending paragraph. As humans, we are quite robust in terms interpreting text which deviates from the expected format. This is not the case for compilers. When a compiler receives code that does not fit the grammar of the programming language it expects it will outright reject it, and by doing so force you to correct it.

The reason for this is quite simple: While one could make the compiler attempt to guess your intention, this would represent an ambiguity that can have unforeseen consequences. For instance, it could easily change the result of a computation. In section 6.2.3, we will cover a good example of this. The kind of mess-ups that we see from generative AI today are simply not acceptable in most production systems. As generative AI dumps the price of low-quality code (e.g., through vibe coding), we may start to see a shift in peoples mindset and a lowering of expectations. But for now and the foreseeable future, the compiler being dumb should very much be considered a feature.

But let's get back to the hello world example. The specific grammatical rules relevant for understanding how a compiler sees the code are illustrated in syntax 4.1 as a railroad diagram. Later on, you might encounter the same information in a textual form called Backus-Naur Form (or simply BNF). While that format is more relevant for programmatic use, in this book we stick to the visually more appealing railroad diagram.

This grammar defines five grammatical concepts, namely pgm, stmt, param-list, expr and name. Each of these definitions define the valid sequences of tokens from an empty circle on the left to a filled circle on the right. We read these by following the tracks. The program itself (or pgm) thus has to contain a sequence of at least one statement (or stmt). This, due to the loop around stmt, as defined in the intermediary stmts.

The next rule unfolds what a statement is. A statement must consist of a name followed by a period followed by another name followed by a left parenthesis followed by a parameter list (or param-list) followed by a right parenthesis followed by a semicolon. The next rule states that this param-list is either empty, or a sequence of expressions (expr) with commas in between. The expr has to be a string literal (string-lit).

By now, you should have noticed the two different types of nodes in the diagram. One type, the terminals, represents a single token like a comma or a left parenthesis. The other type, the nonterminals, represents another concept that can be unfolded or expanded. So, why those names in particular? This has to do with wether the node in question *terminates* the unfolding of concept nodes: While a program is a complex thing that can take many forms and needs further detailing to be fully described, a comma is simply a comma.

4.2.3 Parsing

That sequence of tokens is fed into the parser. Informed by the grammar of the language, the parser finds a unique application of the rules that match the sequence of tokens. The result is a tree structure called a parse tree. Each unfolding of a rule results in child nodes being added to the tree. The hello world program results in the parse tree of figure 4.4. The parse tree contains only those elements whose values could vary from rule application to rule application. So, each comma or parenthesis is left out. Should the parser fail to produce a parse tree from its input tokens, then it will exit with a parse error.

4.2.4 Type Checker

When the final program is executing, the parts of that program that comes from an expression will evaluate to some value. The parts that comes from a statement does not. As we will see in the next chapter, any value has a type. Many of the grammatical rules that deal with expression have restrictions on the types of those expressions. For instance, you can divide a number by another number, but not by a string (such as `"Hello, World!"`).

The parse tree is fed into yet another component of the compiler called a type checker. The type checker has access to a suite of type rules. It's job is to verify that none of them are violated. Any violation is a type error.

4.2.5 Compilation

The binary output of the compilation maintains much of the structure of the parse tree. In fact, the structure of the parse tree defines *how* the binary will be evaluated, when executed by the processor. This processor is, in the case of C# a logical one rather than a physical one; it is implemented in code as a virtual machine and essentially runs as a program on top of your operating system. That means that the binary can be executed on any architecture as long as a matching virtual machine is available for it. Such virtual machines have lots of intricacies, but we will leave them out of this book.

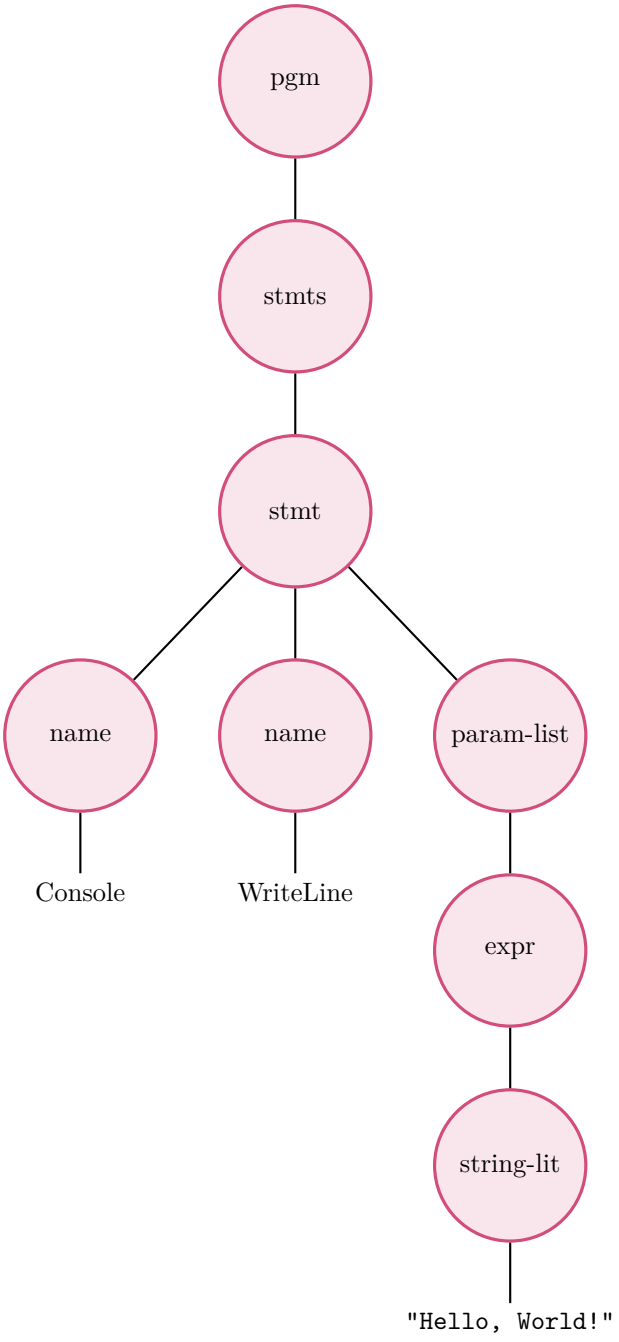


Figure 4.4: Parse tree of hello-world program.

4.2.6 Importance

While it is not critical to understand these intricacies in your day-to-day practice as a programmer, they do a great job at helping you reason about what you experience, and that saves you a lot of both effort and frustration. The takeaway should be that the compiler goes through a number of phases. Each of these have a distinct function and if you provide the compiler with invalid input (i.e., your source code), one of them will fail.

As we move through this book, and broaden our understanding of the C# language, we will extend on this grammar. The pgm definition will receive one extension and both stmt and expr will receive many. We settle for these definitions being *mostly* correct as the correct rules are at least an order of magnitude more complicated, and this complexity is disruptive for understanding of the language at this level.

4.3 Python

In Python, the equivalent program looks like this: [extract file]

```
#!/bin/env python3
print("Hello, World!")
```

In order to execute this script, we first need to make the file executable:

```
aslak@gaia:/tmp/python_hello$ chmod u+x hello.py
```

That first line of the Python source code is called a shebang after the first two characters. When executing, this line uses the `/bin/env` program to look up an appropriate interpreter for `python3` and then feed the remaining lines to it. This is known as a dispatch mechanism. The development cycle is very similar to C# except that there is neither an initial project creation step or a separate compilation step. Running the program is done like so:

```
aslak@gaia:/tmp/python_hello$ ./hello.py
Hello, World!
```

The Python interpreter has an interactive mode that we can invoke directly:

```
aslak@gaia:/tmp/python_hello$ /bin/env python3
Python 3.13.3 (main, Apr 10 2025, 21:38:51) [GCC 14.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>>
```

On the first line we invoke the interpreter. Then, it – exactly as our shell – presents us with a prompt. This prompt allows us to issue python commands. The interactive mode is typically used for rapid experimentation. Such an interactive mode is commonly known as a REPL

(request-eval-print-loop). That is, a program that requests a line of input from the user, evaluates that line, prints out the result and loops back to requesting a new line.

Another way to write python code is though a notebook environment. Such an environment is usually provided through a browser by a system such as Jupyter. In such an environment, code is split up into cells and these cells are laid out in a sequence. Any cell can build on the code that is introduced in previous cells and the code in these cells have easily accessible means for producing visual outputs. The workflow is essentially the same as for C# except that notebooks are usually saved automatically.

4.4 C

The minimal hello world program in C looks (more or less) like this: [extract file]

```c
#include <stdio.h>

int main (int argc, char* argv[]) {
  printf("Hello, World!\n");
}
```

This version immediately looks a lot more complicated! So, why is that? Well, let's try to disect it …

4.4.1 Explanation

Most general-purpose languages (like C#, C, Python and Elixir) come with a sizeable library of functionality. While all of this is available to you as a programmer there are downsides to having it all adirectly available all the time. To simplify grossly, it is a situation similar to finding a specific needle in a stack of slighty different needles. The first line tells C – or rather the C preprocessor – to include the definitions present in the `stdio.h` file. This pulls in, amongst others, the `printf` function, that can print stuff to the screen for us.

Wrapping use of that `printf` function is the declaration of `main`. This is the main entry point of the program. That means that this is where the execution of the program starts. It informs the program of how many options have been passed to it (through `argc`) and what those options are (through `argv`). This is kernels primary means of informing a program of what inputs it should operate on. The `int` before `main` indicates that the program evaluates to an integer value. It is convention that a zero indicates success and a non-zero value functions an an error code. As this program doesn't explicitly return such a number, it will default to zero.

It is important to point out that C# supports a very similar program structure. After all, this is how the kernel passes options to a program. Any language that supports such options will have something similar. It is typically in a `main` function. But sometimes it is implicit or has to be actively imported. That is not likely to mean much to you at this point, and that is okay.

4.4.2 Workflow

In order to compile the code to an executable binary we need to invoke a compiler. Let's use the `clang` compiler and explicitly name the file that should hold the resulting binary:

```
aslak@gaia:/tmp/c_hello$ clang hello.c -o hello
```

One key difference, compared to C#, is that C compiles to a native binary. That is a binary that follows a format that is directly compatible with the processor architecture of your computer. We notice this when executing the code. With C#we used `dotnet` to essentially simulate a virtual machine that was capable of interpreting the intermediate language binary. This manifested as a command that began with `dotnet`.

The `dotnet` program that is then invoked is another example of a native binary. So, we can invoke the binary that we have built from our C code in exactly the same way. However, unlike `dotnet`, our binary is not in the path. That means that we need an explicit path to it. That is accomplished by prefacing it with `./`, like so:

```
aslak@gaia:/tmp/c_hello$ ./hello
Hello, World!
```

4.5 Elixir

4.5.1 Interpretation

Elixir code can be executed through an interpreter (like Python), it can be worked on through a notebook system (like Python) and it can be executed on a virtual machine (like C#). This virtual machine is called BEAM. Unlike that for C#, it is a distributed virtual machine. The hello world program looks slightly different depending on whether a separate compilation phase is used or the code is interpreted. For the script version, the it looks like this: [extract file]

```
#!/bin/env elixir
IO.puts("Hello, World!")
```

As with the Python script (in section 4.3) it uses a shebang mechanism. For that mechanism to work it needs to be executable. However, while the interpreter for the script is called `elixir` this is not the case for the REPL. The interpreter for the REPL is called `iex`:

```
aslak@gaia:/tmp/elixir_hello$ iex
Erlang/OTP 27 [erts-15.2.1] [source] [64-bit] [smp:16:16] [ds:16:16:10]
↪  [async-threads:1] [jit:ns]

Interactive Elixir (1.18.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> IO.puts("Hello, World!")
Hello, World!
:ok
iex(2)>
```

The line that makes the printout appear is still the same. Here, you see it appear after a prompt. The `iex` prompt is, unlike the Python one numbered. Also, everything in Elixir evaluated to a value, and the REPL prints out this value after having evaluated a line. In this case `IO.puts("Hello, World!")`, after having performed its function of printing *"Hello, World!"* to the screen, evaluates to `:ok`.

4.5.2 Compilation

Like C#, Elixir has a notion of a project. A new project can be started using the `mix` command like so:

```
aslak@gaia:/tmp/elixir$ mix new hello
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/hello.ex
* creating test
* creating test/test_helper.exs
* creating test/hello_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd hello
    mix test

Run "mix help" for more commands
```

Let's `cd` to the newly created directory and make sure that `lib/hello.ex` contains the following code: [extract file]

```
defmodule Hello do
  def hello do
    IO.puts("Hello, World!")
  end
end
```

We can compile the project using the following `mix` command:

```
aslak@gaia:/tmp/elixir/hello$ mix compile
Compiling 1 file (.ex)
Generated hello app
```

The BEAM virtual machine is in some ways more complex than the CLR. One such way is that it can run multiple applications at the same time. In order to do so, we need to define an application in our project and tell it to invoke our `Hello.hello` function. This, however, is not an Elixir book, and that is a detail that does not help us progress.

You should note that the project does compile. One reason for that is that we can use the compiled code through our REPL. We do this by passing the `-S mix` paramter along to our `iex` command:

Figure 4.5: Screenshot of the main page of Livebook.

Figure 4.6: Screenshot of the configuration page of Livebook.

Figure 4.7: Screenshot of a notebook in Livebook.

```
aslak@gaia:/tmp/elixir/hello$ iex -S mix
Erlang/OTP 27 [erts-15.2.1] [source] [64-bit] [smp:16:16] [ds:16:16:10]
↪  [async-threads:1] [jit:ns]

Interactive Elixir (1.18.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Hello.hello()
Hello, World
:ok
```

4.5.3 Notebook

Like Python, Elixir code can run in notebook form. Unlike Python, we will actually be using this later on. So, let's see what it looks like in Livebook, the Elixir application for editing and executing notebooks.

Once started, the Livebook program exposes a webserver that hosts a webapp. Point a browser at the URL of this webserver and you will be greated by the webapp. Depending on the specific version you should see something like the screenshot of figure 4.5. At the top righthand corner there is a blue button labeled "+ New notebook". Next to it you will find a button labeled "Open" that you can upse to open a previously saved notebook. Click the first button to create a new notebook.

Livebook has a few settings that we would like to change in order to improve our experience. To do so, open up Livebook, go to Settings and scroll down to the "Code Editer" segment. Make sure that:

- ***Automatically close brackets*** is disabled.
- ***Render ligatures*** is enabled.

4.5.4 Clustering

First, we need to start a named node and provide it with a cookie (read: secret). There are a number of ways to accomplish this, but the easiest one is to start a new REPL session using the `iex` command with a few parameters:

```
aslak@gaia:/tmp/elixir/hello$ iex --name test@127.0.0.1 --cookie mycookie
Erlang/OTP 27 [erts-15.2.5] [source] [64-bit] [smp:16:16] [ds:16:16:10]
↪  [async-threads:1] [jit:ns]

Interactive Elixir (1.18.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(test@127.0.0.1)1>
```

Then, create a new notebook in Livebook and add four Elixir code cells to it. In the first cell we make the node hosting the livebook connect to the node of the REPL (this only has to be accomplished once):

```
node = :"test@127.0.0.1"
cookie = :mycookie

Node.set_cookie(node, cookie)
Node.connect(node)
Node.alive?()
```

This should evaluate to `true`. Next, we can run some code on the node we just connected to. In this case, we wait for a message of a particular format to be received and send back an appropriate response:

```
pid = Node.spawn_link node, fn ->
  receive do
    {:ping, org_pid} -> send org_pid, {:pong, "Hello, World"}
  end
end
```

The third cell contains code for sending a message containing the process ID of the current process to the process that we just started on the other node. The code for this is:

```
send pid, {:ping, self()}
```

At this point we can use the fourth cell to wait for a message to arrive:

```
receive do
  message -> message
end
```

Note: This is a very simple example of how code can communicate across a clustered virtual machine. While this may seem overwhelming to you now, it can both get much more complicated and much more beneficial.

4.6 Summary

The covered languages support for execution models is summarized in figure 4.8. C and C# only support a single model each, while Elixir supports 5 (or 3 with variations). Does that mean that Elixir is a superior language? No, certainly not. But it does mean that you have more options as to how you execute Elixir code. Sometimes support for specific execution models matter, but often it does not.

It is also important to point out that the design of a programming language is riddled with tradeoffs, and these are often resolved in very different ways. That results in some set of qualities, of which the supported set of execution models are found. As a developer, it will be your job to decide which combination of qualities are needed to maximize value for a given project, and based on that choose a language.

C	C#	Python	Elixir	**Execution model**
×				Compile and execute directly on hardware
	×		×	Compile and execute in VM
		×	×	Script
		×	×	REPL
		×	×	Notebook
			×	Script attached to running VM
			×	REPL attached to running VM
			×	Notebook attached to running VM

Figure 4.8: Summary of execution models of various programming languages.

4.7 Exercises

Exercise 4.1: Framework

Topic
Creativity

Install the .NET 8.0 framework on your laptop by following these instructions: https://dotnet.microsoft.com/en-us/download.

Exercise 4.2: Framework Version

Topic
Creativity

Create a new C# project called `project` and inspect the resulting `project.csproj` file. Which version of the .NET framework should `dotnet` use to compile the project?

Exercise 4.3: Hello World

Topic
Creativity

Create a C# project called `hello`, implement in it a program that writes out "`Hello, World`", compile it and execute it.

Chapter 5

Primitive Types

5.1 Numbers

All values that we may want to work with in a computer are, one way or the other, numbers. This fact may not always be obvious, but it will be quite a while before you encounter such examples. From public school and highschool, we know these classes of numbers:

- ***The natural numbers*** ($\mathbb{N}$) These are the non-negative numbers that can be written without a decimal point[1].
- ***The integers*** ($\mathbb{Z}$) These are all the natural numbers (according to the first definition) along with the negative versions of the natural numbers (according to the second definition). That definition juggling is to make sure that we don't have a negative zero.
- ***The real numbers*** ($\mathbb{R}$) These are all the numbers that can be placed on a line between $-\infty$ and ∞. We often refer to these as floating point numbers.
- ***The rational numbers*** ($\mathbb{Q}$) These are then numbers that can be written as the fraction m/n where $m \in \mathbb{Z}$ and $m \in \mathbb{N}$.
- ***The irrational numbers*** ($\mathbb{P}$) These are numbers that belong to $\mathbb{R}$ but not $\mathbb{Q}$. This is where π and e belong.

Computers typically works with other classes of numbers. In the following sections, we will explore the most commonly used ones. Common to these is that they are all represented in binary numbers. That is, using the base-2 digits. We are used to working with base-10 where numbers are digits between zero and nine. In base-2, the numbers are made up of digits between zero and one (i.e., `0` or `1`). Such a digit is called a bit, and a sequence of eight bits is referred to as a byte. The plural forms of the singular bit and byte is bits and bytes. Bits are, by the way, the SI unit of information, in the same way as meter is the SI unit of distance. Accordingly, we measure information in bits.

[1] There are two definitions of natural numbers. The other definition is that they are the positive numbers that can be written without a decimal point. That is to say, without zero. As this definition is generally less applicable in computers, we stick to the first definition.

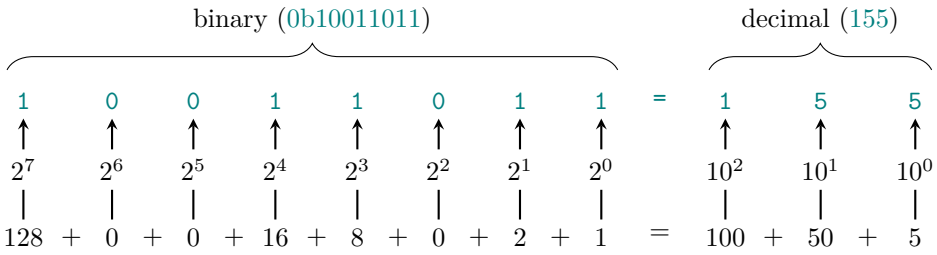


Figure 5.1: Representation of numbers in decimal and binary.

5.2 Integers

We recognize integers as numbers without a decimal point. We use them for counting, and variants thereof. A human may have two or three siblings. But if I were to tell you that I know someone who has 2.3 siblings then you would begin to question my faculties.

5.2.1 Operations

All general purpose languages support the four basic arithmetic operators for addition, subtraction, multiplication and division of integers. The tree first of these will guarantee an integral result. For instance, if you add two integers, you will get an integer. The *size* of this integer may be affected though. But more on that in section 5.2.2.

That rule, about the result of an operation resulting in an integral value does not hold for the division operation though. Mathematically, the result *may* be integral, but generally speaking there is no guarantee, and more often than not it is simply not the case. Most programming languages supports two types of division; namely integer division and floating-point division. The result of a floating-point division is a floating-point number as introduced in section 5.3. The result of an integer division is an integer.

This operation is tightly coupled to both the remainder and the modulo operations. The difference between these is how they handle negative numbers.

5.2.2 Representation

We all know the decimal numeral system. That is, the way we represent numbers using base-10 positioning of digits. In the west, we write the least significant digit to the right. There are ten potential values for each digit and a 10x order of magnitude between two successive digits. The same goes for binary (or base-2). We write the least significant digit to the right, we have two potential values for each digit (zero and one), and the difference between two neighboring digits is a factor of two. Binary values are often prefixed by "0b". Figure 5.1 illustrates how to interpret a binary number.

Normally, when we write out a number, we don't care about how much space it takes up. We have a need to represent the number and it takes up the space it takes up. Space is not an issue. But in a computer, that number need to be worked on by the machine (see section 2.6). The ALU has to be designed for parameters of a specific number of bits, and the instructions for loading and storing register values have to be designed for a specific number of bits. While the hardware support doesn't necessarily put hard restrictions on what we can do, it does dictate

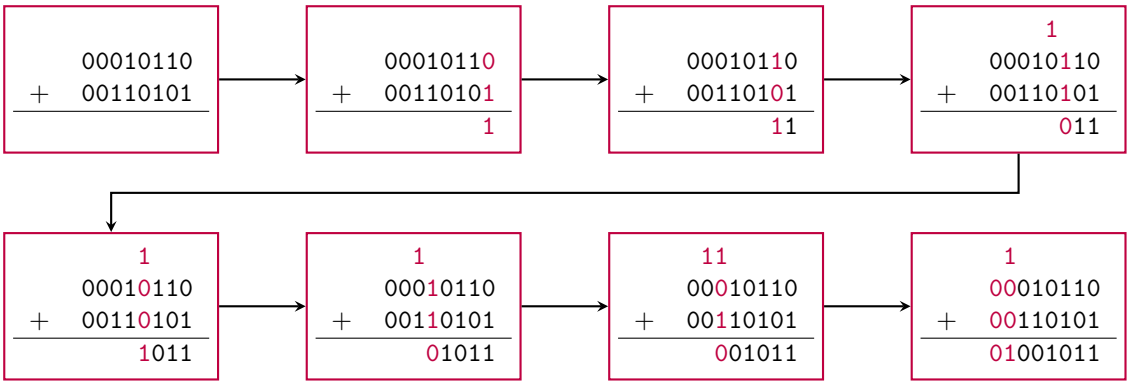


Figure 5.2: Addition of two 8-bit binary numbers.

which sizes of integers will be efficient for us to work with. These are usually calculated in bytes and measured in powers of two. Common examples are 1, 2, 4, and 8 bytes, or the corresponding 8, 16, 32 and 64 bits.

Often, you will also come across hexadecimal representation, which is still the same only base-16. But here we run into a problem: While it is easy how to represent the ten lowest valued digits in a 16 digit system (i.e., 0 through 9), what do you the with the remaining six? How about eleven? Obviously, we could call it 11. But the consequence of that would be that we would either have to use two characters for each digit (e.g., 06 for 6), or rely on some other formatting trick (e.g., {1,7,12,14,3}) to be able to unambiguously decode a number. Both solutions are cumbersome and take up space. Instead, we usually say that eleven is "a", twelve is "b" and so on. Then a digit is still a single character, and this allows us to write a number compactly. Hex values, as these are called, are often prefixed by "0x". An example of such a value is 0x04ff2b.

The binary notation is convenient when we think about how the numbers are represented in the machine. Operations are performed in pretty much the same way that you are used to from decimal numbers. Figure 5.2 illustrates how to add two binary numbers. If you squint your eyes, you should recognize something familiar. The main difference is that when we are doing calculations on a piece of paper, we can pretty much keep on adding digits. In a computer we have a fixed number of digits. And when we run out of digits, we get a wrong result. For instance, the highest number that we can represent consist of all one digits. If we were to add the number one to this number, then the carry travels all the way through the digits in ascending order thereby clearing all bits. This is called an overflow. The series of numbers that can be represented essentially loops around so that the largest integer that can be represented (the so-called MAXINT) comes just before zero.

5.2.3 C#

C# supports the signed and unsigned integer types listed in figure 5.3. Let's say that you need to be able to represent any positive number below 100.000. That number sits somewhere in between the limits of 16 bits and 32 bits. This means that we need at least 32 bits, and if we use more than 32 bits to represent it then we will be wasting memory. Depending on our architecture, there might be processing speed to trade for this "wasted" memory. But let's just stick to the 32 bits for now. As we don't need the negative values we will go for an `int`.

Size	*Unsigned Type*	*Signed Type*	*Comment*
8 bits	`byte`	`sbyte`	
16 bits	`ushort`	`short`	
32 bits	`uint`	`int`	
64 bits	`ulong`	`long`	
1 word	`nuint`	`nint`	Do not use

Figure 5.3: Integer types available in C#.

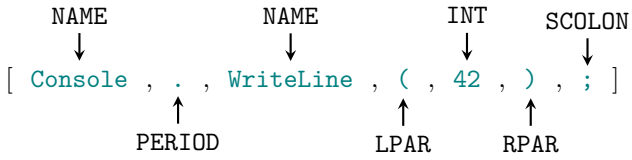


Figure 5.4: Token sequence of a program the prints out an integer.

That last line on the table is a little bit different though. You may have noticed that it is measured in words instead of bits or bytes. On most processor architectures, a word will be defined as either 32 bits or 64 bits. See it as the most efficient integer size. The `n` in `nuint` and `nint` stands for *native*. A side-effect of using these types is that the effective MAXINT is machine dependent, and that can cause portability issues if you are not careful. For now, just stay away from these two types.

The naming convention of the integer types is a bit unfortunate from a consistency perspective. The `u` prefix is used to denote an unsigned variant, except for the `byte` case where the `s` prefix is used to denote an signed variant. In `nuint` the `u` still denotes something that is unsigned but it is not at the beginning of the name. So, from a human side, parsing these names if a relatively complex task. With a bit of experience though, you won't notice.

5.2.3.1 Expressions

The following line prints out "42" to the screen:

```
Console.WriteLine(42);
```

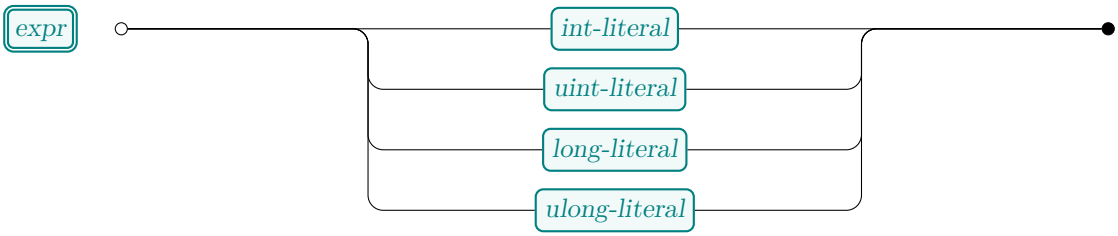
Then the lexer is fed this program, it produces the token sequence of figure 5.4. The `INT` token represents a concrete `int` value that was embedded in the input source code. Such values embedded in source code are called literals. Literal forms exist for most of the primitive types.

It is a bit more complicated than that though: Literals are typed. That means that the compiler sees it as having a type in addition to having a value. So, in the previous example, that `42` part of the code represented an `int` with a value of 42. We know this because of the literal specifier rules of C#. For integers, the rules are listed in figure 5.5.

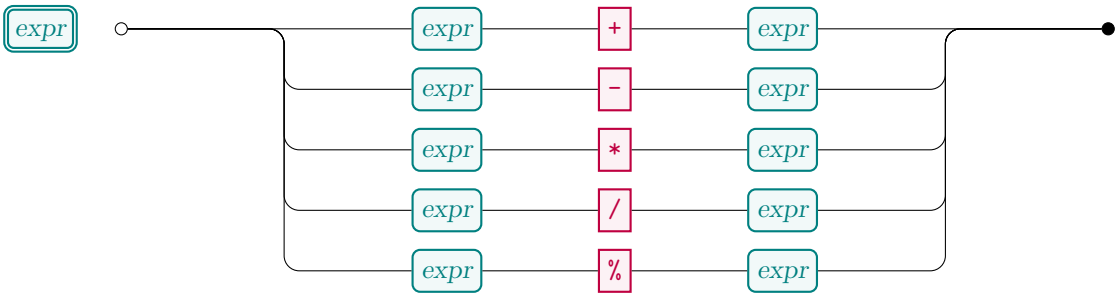
Literals in general, and integer literals in particular, are expressions. That means that we can extend our set of syntax rules with the set from syntax 5.1. This is the reason why the line of code from the beginning of this section is syntactically valid.

Suffix	Example	Type
	`1`	Delayed typing
`U`	`1U`	Explicit `uint`
`L`	`1L`	Explicit `long`
`UL`	`1UL`	Explicit `ulong`

Figure 5.5: Literal specifiers for integer values.



Syntax 5.1: Expressions of integer literals



Syntax 5.2: Expressions of arithmetic operators

5.2.3.2 Operations

5.2.4 High-Level Languages

In Elixir, there is only one integer type and that has an arbitrary size. As long as an integral value fits in memory, an Elixir integer can hold it. The value is not in having integers that each take up 90% of your primary memory. It is very rare that we have a need for integers beyond 128 bits or even 64 bits. But when we work with fixed size integers, we always have to be aware of that hard limit. In Elixir, we don't have to: If we somehow end up with a number needing 7000 bits, then that is what gets allocated. We don't have to worry.

Elixir code, still runs on the same hardware though. This hardware does not have instructions capable of operating on 7000 bit integers. So, instead of an integer operation being a single instruction, it is an algorithm in itself. This is significantly slower. Elixir is designed for network intensive tasks, and these are even slower. So, for the tasks where one would choose Elixir, it basically doesn't matter. And that is why the designers of that language has made that choice.

5.3 Floating Point Numbers

We use integers to express a number of whole units. The number itself does not tell us what that unit is. That depends on what the integer represents. What it is used for. Decimal numbers, on the other hand, are use to express numbers are a granularity that is higher than one unit. That is, they can express fractional units. A recipe for an omelet might state that you need 1.5 eggs per person. We can't represent those 1.5 using an integer type. For that, we need a decimal (or floating-point) type.

5.3.1 Operations

Most of the operations listed under integers are also applicable to floats. This includes all the comparisons, and all of the arithmetic operations don't rely on integer values. That is, addition, subtraction, multiplication and division. The modulo (remainder, really) operator is the odd one out. While it would make sense to talk about a remainder when dividing one float by another, this is not how the operation is used in practice, and we normally only have hardware support for it as the result of an integer division.

5.3.2 Representation

Modern processors support operations om floating point numbers that follow the IEEE-754 standard. These are commonly referred to as IEEE (pronounced I-tripple-E) floats. This standard covers the representation of the numbers and definitions of how certain operations work on them. It is important to realize that many numbers cannot be represented exactly using IEEE floats, and to accept that this often causes calculated results to be a bit off.

The basic idea behind the IEEE representation is that a floating-point number is an integer that has been shifted. That is, it can be expressed as $m \cdot 10^e$. Here, m is referred to as the mantissa, and e as the exponent. -1.2 can then be expressed using a mantissa of -12 and an exponent of -1 because $-12 \cdot 10^{-1}$.

Reality is, of course, a bit more complicated. Just as our integers come in multiple sizes, so do floats. We will be focusing on two versions, namely single-precision that takes up 32 bits each and double precision that takes up 64 bits each. Unlike integers, they do have an inner structure. This structure is illustrated in figure 5.6 and consists of:

- ***Sign:*** A single bit is dedicated to the sign of the number. This means that all floats are signed. As -1.2 is a negative number, the sign bit becomes a "`1`".
- ***Normalized Mantissa:*** We have already shown that we can represent -1.2 as $-12 \cdot 10^{-1}$, but we can also represent it as $-120 \cdot 10^{-2}$ and a number of other mantissa/exponent combinations. If we multiply the mantissa by ten, then we need to subtract one from the exponent. One representation is the normalized one. In reality both mantissa end exponent are coded in binary. The normalized representation of the mantissa is such that the number is shifted all the way to the beginning of the sequence. That way, any non-zero mantissa will have a one digit at the first position. This turns out to be a waste of a single bit, and because of that the sequence is shifted one more position. At the end, this won't affect the zero case. This is how the normalized mantissa is encoded. To find the normalized mantissa of -1.2, we first remove the sign to get 1.2. This is $1 + 0.2$ and that is (if we remove the leading one) approximately "`00110011001100110011010`" in single-precision binary.

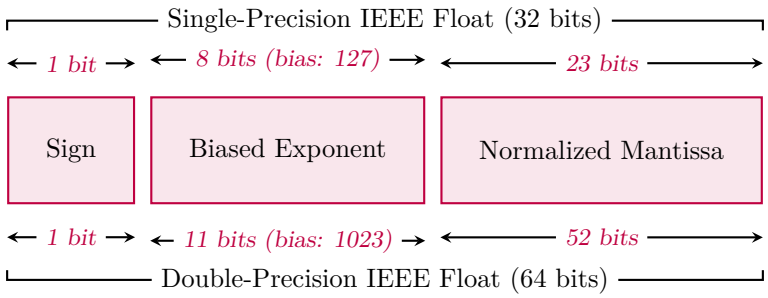


Figure 5.6: IEEE representation of floating-point values.

- ***Biased Exponent:*** If the mantissa is zero, then so is the exponent. Otherwise, the exponent represent how much the mantissa needs to be shifted after adding the previously removed initial "`1`". That is, where to place the comma. In practice, we take the binary logarithm to the positive variant of our number (we have already covered the sign, remember?). We then floor the resulting floating-point number and add a bias. This is our biased exponent. The bias is added to enable us to shift the number in the opposite direction and thus allow for numbers below one. A specific bias thus tells us how the representable range is placed. In our example, this is $\lfloor \log_2(1.2) \rfloor + 127 = 127$, and that is "`01111111`" in 8-bit binary which is needed for the single precision variant.

The IEEE float format leaves space for a number of special values:

1. ***Zeroes:*** Any float with a normalized mantissa of zero and a biased exponent of zero is a zero value. The only thing left is the sign bit, and accordingly we have a positive and a negative zero. These two values are equal by definition.
2. ***Infinities:*** Any float with a normalized mantissa of zero and a biased exponent of all ones represent infinity (∞). The sign bit is used to differentiate between ∞ and $-\infty$.
3. ***NaN:*** Some combinations of operations and operands are not allowed (e.g., the multiplying any infinity by zero). Such operations yields the NaN (or Not a Number) value. Any float where the normalized mantissa is nonzero and the biased exponent is all ones is a NaN value. There are a few more details to this, but we have no need to get into them here.

Adding two floating-point numbers involves finding a common exponent. This may result in reduced resolution in one of the operands, and that causes rounding errors in the result. Because of this, we may find that $a + b = a$ even at times when $b \neq 0$. This usually happens when the magnitude of a and b are very different. Something similar happens with other operations on floating-point numbers, and the compounding error can end up being not only significant but large as well. This is referred to as numerical instability. In scientific computing, care is taken to rephrase important calculations in such ways that the order of operations minimize the introduced error.

So, IEEE floats consists of a sign, an exponent and a mantissa. If either the mantissa or the exponent isn't large enough, then you need a larger float size. Most modern processors have hardware support for IEEE single and double precision floats, and this is typically mapped to two types in programming languages: 32 bit floats and 64 bit floats. We can implement support

Suffix	Example	Type
	`1.0`	Implicit `double`
`F`	`1.0F`	Explicit `float`
`D`	`1.0D`	Explicit `double`

Figure 5.7: Literal specifiers for float values.

for arbitrary sized mantissas and exponents ourselves, but that will be without hardware support in the processor and thus very slow.

5.3.2.1 Alternatives

One alternative is to express a floating-point number as a fraction of two integers. Operations on such numbers are slower than their equivalents on IEEE floats, but it allows us to to avoid the situation where common operations introduce errors.

5.3.3 C#

C# supports both single-precision and double-precision floating-point numbers. The `float` type is used for 32 bit single-precision floats, and the `double` is used for 64 bit double-precision floats. As with integers, C# has literal specifier rules for floats. These are listed in figure 5.7.

5.4 Truth Values

So far, we have covered numbers. Integers of various sizes and signedness, and decimal numbers of different range and precision. A 32 bit unsigned integer is capable of representing any value between 0 and $2^{32} - 1$ (both included). Similarly, a 1 bit integer would be able to represent exactly two values: 0 and 1. Sets of integers go, that is not particularly useful. But, if we look past the individual numerical values, being able to represent that something has one of two states (e.g., on/off or open/closed) certainly is.

One particular pair of states is especially useful for us: Truth values. It allows is to represent whether something is true or false. In reality we map 0 to false and 1 to true. That is, we have the value mapping: $\{0 \mapsto \text{false}, 1 \mapsto \text{true}\}$. As the concept of truth values was introduced by George Boole, we have come to refer to these values as booleans or, in short, *bools*.

5.4.1 Operations on Numerical Values

Any valid comparison between two numerical values will yield a truth value. For instance, if we ask *"is 42 equal to 56?"* then the resulting value is `false` but if we ask *"is 1 less than or equal to 2?"* then the resulting value is `true`. Comparing values is one of the primary ways of creating boolean values.

5.4.2 Operations on Truth Values

Booleans are, however, not just something that can be used to represent the outcome of some comparison. In fact, there is a whole algebra built around this type. We will cover the following operations:

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$\overline{a}$
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

Figure 5.8: Truth table for the boolean operations.

- ***NOT Operator*** Whether a value is not true (aka whether it is false). It is written as $\overline{a}$. A door is open if it is not closed: $open = \overline{closed}$.
- ***AND Operator*** Whether two values are both true. It is written as $a \wedge b$. You are free to go if you have a green light and the road is clear: $free\_to\_go = green\_light \wedge clear\_road$.
- ***OR Operator*** Whether at least one of two values is true. It is written as $a \vee b$. A plant will die if kept dry or in darkness: $will\_die = dry \vee darkness$.
- ***XOR Operator (pronounced X-OR)*** Whether exactly one of two values is true. It is written as $a \oplus b$. A swinging door will open if it is either being pushed or being pulled: $will\_open = pushed \oplus pulled$.

It is common to demonstrate operations on boolean values using a so-called truth table such as the one in figure 5.8. The leftmost columns represent the input parameters. Any additional columns represent operations on these. Each column is labeled, and there is one row for each possible combination of the inputs. Truth tables can be used to document the behavior of an operator or verify that two operations are equal under all input.

The XOR operator is not considered one of the *basic* boolean operators, and it is also – by some margin – the least used one. It can be hard to differentiate between OR and XOR. For instance, be might say that a game is over if you win or if you die. Should you choose to define this using OR or XOR? The only difference between these operators is what happens if both are true, and that never happens in a game: You don't both win and die. So, in this case it doesn't really matter. When it doesn't matter, most programmers will chose OR. Probably because we consider it linguistically simpler. In other scenarios, the behavior when both inputs are true does matter, and then that becomes the arbiter.

But, let's look at a real-world example. Imagine that we are building a game. In this game the player progresses through a sequence of levels. In order to progress to the next level, the player must at the end of the current level (i) be alive, and (ii) have obtained a certain number of points. After each level the number of required points is raised. Whether the player should progress can then be determined as $alive \wedge P \geq P_{required}$ or (depending on what we track) $\overline{dead} \wedge P \geq P_{required}$.

As we know from elementary algebra, there are rules for transforming certain constructs. For instance, we know that $a(b+c) = ab+ac$. Similar rules exist for boolean algebra. The two most important of these rules are known as De Morgan's Laws (after Augustus De Morgan). They state that:

$$\overline{A \vee B} = \overline{A} \wedge \overline{B}$$
$$\overline{A \wedge B} = \overline{A} \vee \overline{B}$$

Or, in plain English, we have that:

a `==` b	True if a is the same as b
a `!=` b	True if a is different from b
a `<` b	True if a is less than b
a `<=` b	True if a is less than or equal to b
a `>` b	True if a is greater than b
a `>=` b	True if a is greater than or equal to b

Figure 5.9: Comparison operators.

1. The negation of A or B is the same as (the negation of A) and (the negation of B).
2. The negation of A and B is the same as (the negation of A) or (the negation of B).

5.4.3 Representation

As mentioned, a boolean is a type that holds one of two values. That is a bit of information. However, in order to perform operations on a boolean, we need to store it in a register and common processors don't have these. Instead, booleans are placed in integer registers. Depending on the language and/or compiler may store the same booleans in main memory as bytes, integers (of register compatible size) or – if there is a number of them – packed into one byte per 8 booleans.

5.4.4 C#

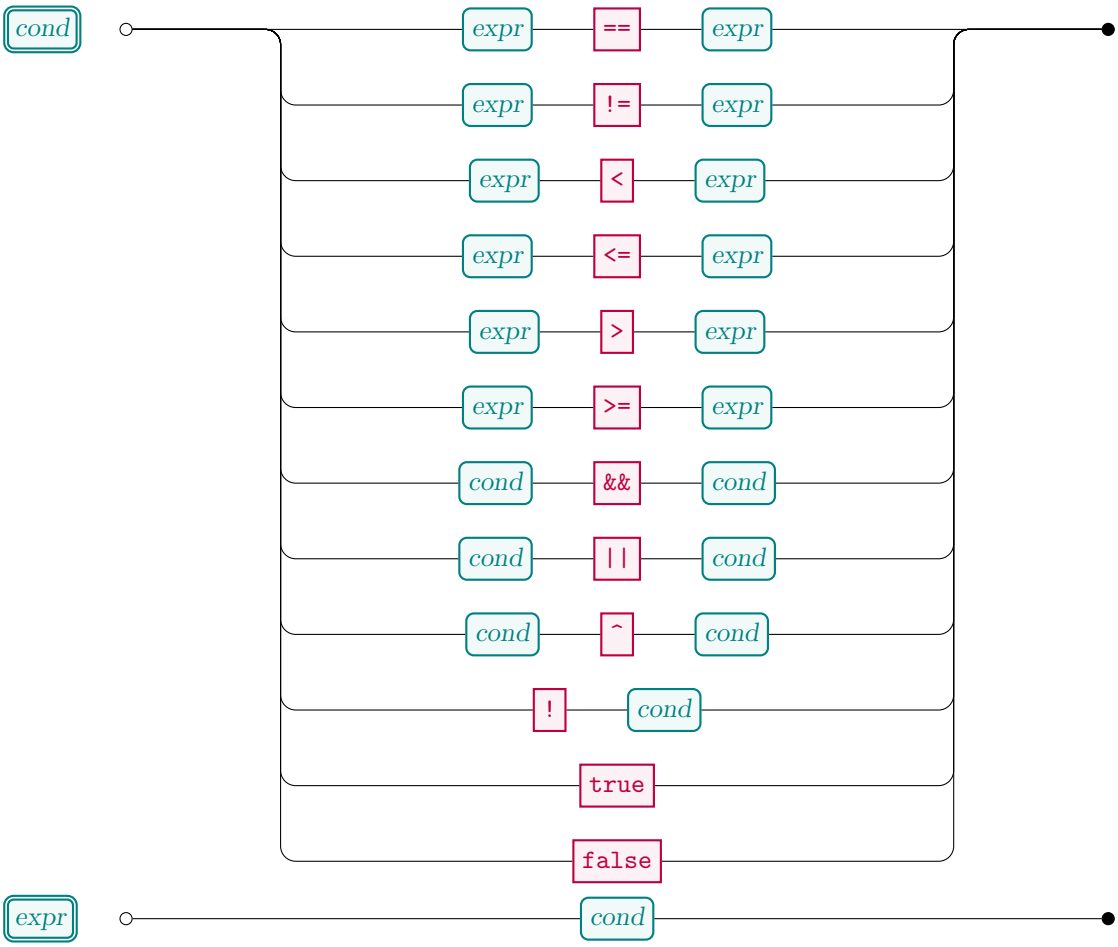
Comparison operations yield boolean values. The full set of comparison operators supported by C# are listed in figure 5.9.

It would be very tedious to use the symbols covered for AND, OR and NOT. C#, like many other languages, use a different notation:

- ***AND Operator*** Two ampersands in a row is an infix AND operator.
  Example: `unlocked && open` should be read as "unlocked and open".
- ***OR Operator*** Two vertical bars in a row is an infix OR operator.
  Example: `working || waiting` should be read as "working or waiting".
- ***XOR Operator*** A hat character is an infix XOR operator.
  Example: `sunken ^ afloat` should be read as "sunken xor afloat" (as in, it cannot be both).
- ***NOT Operator*** An exclamation mark negates whatever condition comes after it.
  Example: `!ready` should be read as "not ready".

In order to support booleans, we need to introduce a new syntactical condition concept called cond. A cond is an expr that ealuates to a boolean value. A cond can take the form of a comparison, a boolean operator or a boolean constant. This is covered in syntax 5.3.

```
bool progress = alive && (points > required_points);
```



Syntax 5.3: Expressions of boolean operators

5.4.4.1 Lazy Evaluation

5.4.5 High-Level Languages

There are historical reasons for C# to use special character combinations for logical operations. From the perspective of C#, this decision was taken to mimic C that, when C# was introduced, had played a defining role in the minds of programmers. Accordingly, using `&&` for a logical AND operation was the natural choice for most programmers of this era. But why did C pick it then? The choices of how to represent these operators were made in such a way that they had no potential conflicts with names for variables (and other things). That would make it simpler to write the parser for the compiler.

Tooling for writing compilers have evolved to a point where the need to simplify to parsing step of the compiler in this way isn't really there. Modern language designers thus don't operate under this restriction. Many modern languages (which usually operates at a higher level of abstraction) have this begun using natural English keywords. They use `and`, `or`, `xor` and `not` instead of `&&`, `||`, `^` and `!`. While this makes it easier for a novice programmer, it also takes up more characters,

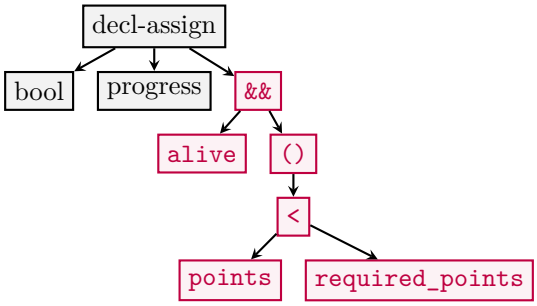


Figure 5.10: Type checking of the statement `bool progress = alive && (points > required_points)`.

and that makes lines longer and thus harder to grasp. And so, people have disagreements about which is the right design choice. Typically, high-level languages (like Elixir and Python) use the natural English keywords, and low-level languages (like C and Rust) uses the special character combinations. In this case, C# fits in with the low-level languages.

5.4.6 Low-Level Languages

C is a simple low-level language. That means that it mirrors the fundamental properties of the underlying hardware and adds some highly convenient abstractions. These abstractions are chosen is such a way that they essentially can be delivered without a performance overhead.

Machine code does not have a boolean type. Instead register values that are represented using all zeroes in binary are *false* and every other value is *true*. That means – in terms of integers – that zero is *false* and non-zero is *true*.

A consequence of this is that C doesn't have a notion of a boolean. Instead, integers are used: A boolean is an interpretation of an integer, and can thus be used in an integer expression. Typically, this will make absolutely no difference. Proponents of C# will point out that booleans and integers are fundamentally different notions and should thus be treated differently. Proponents of C will point out that this allows them to write code such as this:

5.5 Type Checking

Every value has a type, and so does every expression. More specifically, every expression over a value (or two) evaluates to a value that has a type. As an initial step of the compilation, the tree structure of each complex expression is handed over to the type checker so that it can perform type checking on that part of the program. For each operand, it compares input types, operand and output type to a set of rules for how operands translates input types to an output type. If no matching rule is found, then it produces a type error and the compilation fails.

5.5.1 Calculations on Types

5.6 Type Conversion

When every expression is typed and those types determine the operands available to us, we may find ourselves in a situation where we don't have access to (the version of) the operand that we

Syntax 5.4: Casting of an expression to a different type.

need. And so, we need to convert the type. Since the type is linked to the representation, that is really what we need to change. For a primitive type, this process is called type casting.

Most, if not all, of these casts costs a bit of execution time. The cause of this is that the representation has to change in order to guarantee that the rules of the typed operands (e.g., overflow) are observed. It is not a lot of execution time though. In fact, it is so little that we usually don't care about it at all.

5.6.1 C#

Syntax 5.4 introduces the practical mechanics of an explicit cast in C#: We need an expression of one type and a different type to cast it to. If the cast is invalid (i.e., not between compatible types) then the type checker catch it and reject the code with a cast error. An implicit cast does exactly the same, but the destination type is automatically inferred, and (ii) it is only allowed when the type conversion is guaranteed to be lossless. As it is done automatically, no syntax is needed.

But what is behind that *lossless* term? Consider two integer types, namely `int` (32 bits) and `long` (64 bits). If we have an expression that evaluates to a value of type `int`, then that value falls within the $0..2^{32}-1$ range. Note that the corresponding range of a `long` is $0..2^{64}-1$ and thus strictly broader. That is, every single value of an `int` can be converted to a `long` and back again without without us having to worry about loss of data. We say that we can roundtrip it. The same is not true if we go in the opposite direction: While some `long` values can be converted to `int` values and back again successfully, this is not generally the case. The value 2^{32}, for instance, will overflow into the value 0 in `int` form and it will stay 0 when converted back to `long` form. Similarly, any floating point value will naturally loose the decimals if converted to an integer format.

An explicit cast is necessary if we risk loosing data. By performing an explicit cast, we tell the compiler (or really the typechecker part of it) that we take on the responsibility of the risk: Trust me, bro! And so, care should be taken.

5.7 Local Variables

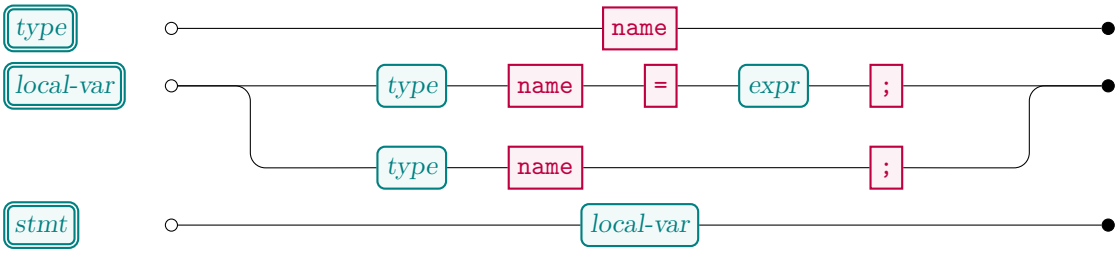
5.7.1 State Transformation

5.7.2 C#

5.7.3 Functional Programming

5.8 Naming Conventions

When communicating with another human, we use names to refer to things. We can do this because we have a shared vocabulary. It doesn't matter what specific names we use, as long as we agree on their meaning. The same goes for when we write source code. Here, we name



Syntax 5.5: Local variables.

Standard	First Word	Separator	Remaining Words	Example
Snake Case	Lowercase	Undescore	Lowercase	`best_score`
Camel Case	Lowercase	None	First letter uppercase	`bestScore`
Pascal Case	First letter uppercase	None	First letter uppercase	`BestScore`

Figure 5.11: Overview of select naming conventions.

various parts of our code so that other programmers (as well as older versions of ourselves) will "get" what their intended function is. As long as all parts of the code is written with a consistent understanding of what these names mean, the compiler won't care. But the specific names are key to us humans.

We still haven't covered much of the C# language, and thus haven't encountered a whole lot of things that need to be named. We have explicitly covered a few different kinds of primitive types and local variables. As the name suggests, there are other kinds of variables that are *not* local. We have briefly touched on `Console.WriteLine`. As we will cover in chapter 12, `Console` is something called a class, and `WriteLine` is a static method. So, we have already touched on a number of classes of elements that we need to be able to name, and we will keep on introducing such classes. One could call this a classification scheme.

Programming languages often comes with naming conventions. A naming convention dictates a mapping from each such class to a specific naming scheme. That is, it defines rules for *how* to name elements belonging to these classes. By choosing different naming schemes for different classes, a naming convention can make it likely that a human – based on the observed naming scheme – can guess which naming class it belongs to. This acts as a hint to the programmer. Software projects often adopt the naming convention of the programming languages they use, but it is not uncommon – for better or worse – to see deviations.

A name could theoretically be anything, but more often than not, we en up wanting to use a sequence of words. An yet, we are not allowed to use space characters in our names.

5.8.1 C#

The C# naming convention states that local variables should be named using camel case.

5.9 Parsing

When we are writing code, what we are really doing is to produce a sequence of characters. We store those in a file, and then ask the C# compiler to interpret them. Given that C# is a language that we share with the compiler, we need a certain level of understanding of how it goes about interpreting it. Otherwise, how can we hope to "communicate" correctly with it?

In this chapter, we have introduced the notion of an expression. Expressions are – in part – made up of operators. Let's take a look at how expressions are parsed. That is, how does a sequence of characters, at a position in our code where the compiler expects an expression, get translated into a parse tree?

5.9.1 Operator Precedence

Basic math tells us that $2+3\cdot 4 = 2+(3\cdot 4) = 14$ as opposed to $(2+3)\cdot 4 = 24$. This is because we have agreed on the rule that the multiplication operation binds tighter than the addition addition operation. Similar rules exist for expressions in programming languages. Just as we learned that $2+3\cdot 4 = 2+(3\cdot 4)$, we ned to learn how tightly each of the operators (we make use of while programming) bind.

Most programming languages define a hierarchy of operator classes. Operator classes at the top level of the hierarchy bind the strongest, and operators and the bottom of level binds the weakest. Within a given level, all operators are treated as binding equally strong. Parsing of expressions involving two such operators is dictated on the associativity of the operator class. More on this in the next section.

5.9.2 Operator Associativity

We are not done yet though. Assuming that we have some operator – let's call it $\odot$ – and keeping in mind that the compiler needs to convert any expression needs to be converted to a sequence of operations for the processor, how should $a \odot b \odot c$ be interpreted? By adding parentheses, we can remove this ambiguity. But, should $a \odot b \odot c$ be equal to $(a \odot b) \odot c$ or to $a \odot (b \odot c)$?

Given what we have covered so far, that question may not seem to matter. Most of the operators that we have covered exhibit the associative property. That is, the order of evaluation does not affect the result. This changes, however, when composing complex expressions that modifies shared state (e.g., when pre-incrementing variables). From section 8 onward, we will cover operators that obviously does not have the associative property.

But let's get back to the question of where to place the parentheses. Some operators are left-associative, some are right-associative and some are non-associative. A left-associative operator is grouped from the left. This means that $a \odot b \odot c = (a \odot b) \odot c$, and that the syntax tree of a sequence of such operations skews heavily to the left-hand side. A right-associative operator is grouped from the right. This means that $a \odot b \odot c = a \odot (b \odot c)$, and that the syntax tree of a sequence of such operations skews heavily to the right-hand side. A non-associative operator is not allowed to appear in such a sequence.

5.9.2.1 Overruling

There are good intentions behind the parse rules; those that cover precedence and associativity. They are defined the way they are defined for a reason. In most situations they represent "what you want" by allowing code to be written as briefly and succinctly as possible. There is, however,

Syntax 5.6: Expressions of parentheses

Category	Operator
Primary:	`a[i]`, `x.y`, `x++`, `x--`, `new`
Unary:	`-x`, `!x`, `++x`, `--x`, `(T)x`
Multiplicative:	`x*y`, `x/y` `x%y`
Additive:	`x+y`, `x-y`
Relational testing:	`x<y`, `x<=y`, `x>y`, `x>=y`
Equality testing:	`x==y`, `x!=y`
Logical XOR:	`x^y`
Logical AND:	`x&&y`
Logical OR:	`x||y`
Conditional:	`c ? x : y`
Assignment:	`x=y`, `x+=y`, `x-=y`, `x*=y`, `x/=y`, `x%=y`

Figure 5.12: Operator precedence levels.

no one way to cater for all scenarios: You will find yourself in situations where the compilers interpretation is suboptimal for what you intend to express.

When this happens, we use parentheses to force a specific interpretation. Novice programmers are often seen sprinkling parantheses around their code to avoid learning the underlying rules. This becomes a source of confusion for more experienced programmers and – for that reason – it is frowned upon.

5.9.3 C#

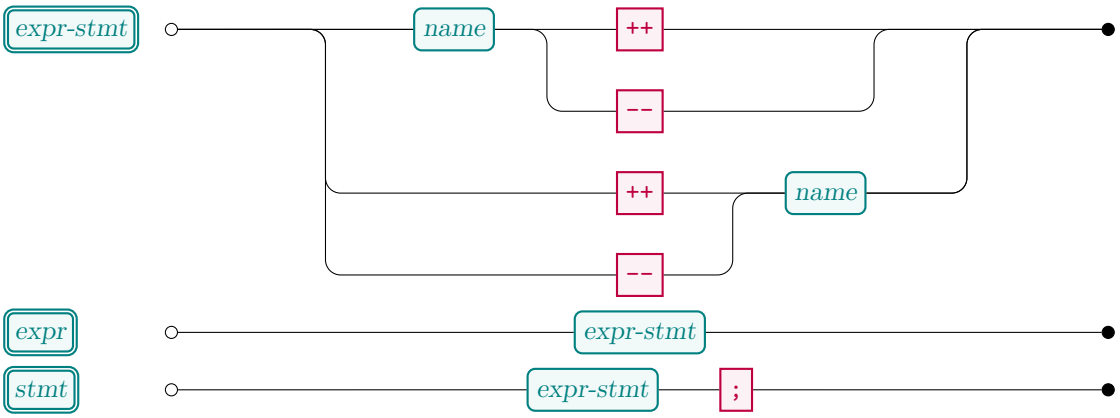
> **Caution:** C# has an `&` operator in addition to the `&&` operator that we have covered here. The difference is that the `&` operator does not short circuit. Normally, in the C family of languages, the `&` operator has a different meaning. C# is really only pretending to be part of this group of languages, and in this case it stumbles. The similarity is mostly at syntax level.

5.10 Statements vs Expressions

A language can be said to be defined through three layers:

1. ***Syntax*** Describing how the structure of valid code *looks*.
2. ***Semantics*** Describing what code *means*, and by extension how it should be interpreted.
3. ***Libraries*** Contain reusable code fragments, written using the same syntax and semantics.

In this section we use syntax and semantics to describe pre- and post incrementation and decrementation.



Syntax 5.7: Expression statements.

We have covered various forms of expressions and statements through the expr and stmt rules. Certain expressions can be used as statements. We could call them expression statements. While expression statements are evaluated to a value, they also have side-effects, like modifying some state in memory (e.g., a variable). The necessary additions to the language can be found in syntax 5.7.

Take the following C# code as an example:

```
int i = 42 + j++ * i;
```

In order to compile this to bytecode that the CLR (the .NET virtual machine) can execute, it must first parse it to a syntax tree. This is done according to syntax rules presented in this chapter and results in the parse free of figure 5.13.

Each node contains annotations of semantic classes, and not part of the tree structure itself. The top node represents the root of the tree. It then branches off into *child nodes* and ends up in *leaf nodes* (those that have no children). This is the parsed statement produced by the parsing of the code. It matches a pattern in the semantic class stmt that is composed of three class elements. An *edge* connects the root statement to each of these. The first is a type, the next is a name and the last is an expr. That last expr takes the form of a multiplication. Try to continue this description of the remainder of the parse tree yourself.

5.10.1 Semantics

The semantics of a language defines what the expressions, statements, operators and the like *mean*. For instance, the operator `+` adds two values. The expression `a + 1` evaluates the `+` operator on the value referenced by the variable `a` and the constant `1`. Knowing these rules, the parse tree should be evaluated from the leaves towards the root.

5.10.1.1 Pre Incrementation

Pre incrementation is an example of more complex semantics. It essentially does two operations:

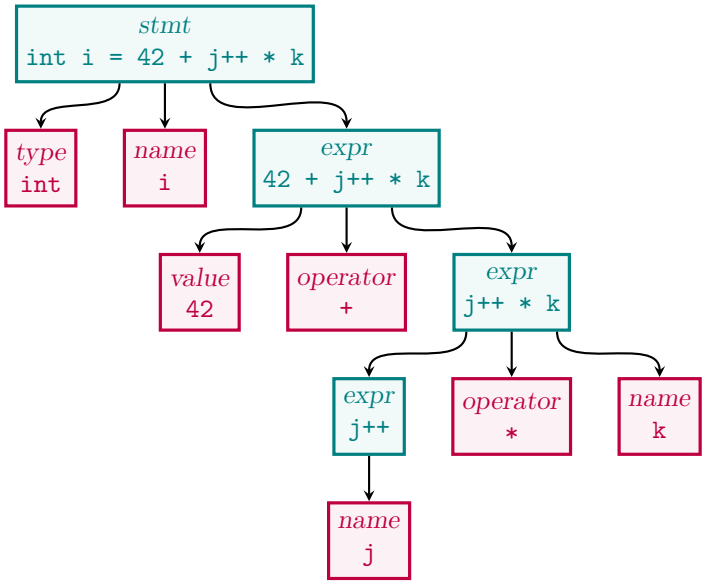


Figure 5.13: Parse tree of `int i = 42 + j++ * i`.

- First, it *increments* (that is, it increases a value) the value of a variable by one.
- Next, the expression statement takes the value of the that variable.

Lets look at a concrete example:

```
int i = 1;
int j = ++i;
```

The first line declares a variable `i` and assigns the value `1` to it. In the second line, `++i` is first evaluated. In doing so, `i` is incremented to the value 2 and then this value is used as the *result* of that evaluation. That is, the evaluation of `++i`. Still in the second line, `j` is then assigned this value. After those two lines, we can describe the state as $\{\mathrm{i} \mapsto 2, \mathrm{j} \mapsto 2\}$.

5.10.1.2 Pre Decrementation

This is exactly the same as pre incrementation, only with a different operator. The corresponding example is:

```
int i = 1;
int j = --i;
```

Again, we assign the value `1` to `i`, and again we perform an operation on `i` before we assign the value of `i` to `j`. Accordingly, we end up with the state $\{\mathrm{i} \mapsto 0, \mathrm{j} \mapsto 0\}$.

5.10.1.3 Post Incrementation

Post incrementation changes the order of the two operations involved, so that:

- First, the expression statement takes the original value of the variable.
- Next, it *increments* the value of the variable by one.

The example is:

```
int i = 1;
int j = i++;
```

Once again, we start by assigning a value `1` to `i`. Then we extract the value of `i` (that is the `1` we have just assigned). Next, we increment `i` by one and marks `i++` as having the previously extracted value. This is then assigned to `j`, and we are left with the state $\{\mathrm{i} \mapsto 2, \mathrm{j} \mapsto 1\}$.

5.10.1.4 Post Decrementation

If you have any doubt regarding the outcome of the following code, then I suggest that you figure it out experimentally by writing a program:

```
int i = 1;
int j = i--;
```

5.10.1.5 Summary

These expression statements can be thought of has having two components:

- ***A Side-Effect*** This is an operation (plus or minus one) on a variable.
- ***A Value*** This is taken from the variable.

When the value is extracted from the variable is determined by whether the `++` or `--` is located before or after the variable in the code.

5.11 Resolving Uncertainty

C# is a complex language. That means that is takes a long time to master, even with effort. We need to study the underlying mechanisms to understand the language, and to (hopefully) become good programmers in the end. These mechanisms are not always simple and well documented. You will find yourself in situations where you are unsure of how the language works.

Such situations has at least one of a number of negative consequences. If you are not aware of the disconnect between the language and your understanding of the language, then you may end up using sub-optimal programming constructs or you may introduce a bug.

Awareness of the disconnect enables you to handle the situation constructively. See it as an opportunity to treat the situation with the respect (and time) it deserves: Read up on the relevant theory, or start poking it experimentally. Construct a piece of software with the single purpose of revealing *how* it behaves. This, with the intention of expanding your mental model of the language. Don't be afraid to get your hands dirty!

5.12 Exercises

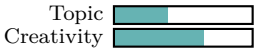
Exercise 5.1: Temperature

Which types are suitable for representing a temperature?

Exercise 5.2: Month

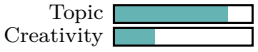
Which types are suitable for representing a month?

Exercise 5.3: Limits of Integers



1. Which integral types are available to us in C#?
2. Pick one of them.
3. Write a program that experimentally explores what happens when we add one to the highest value this type can represent.
4. Describe what you observe.

Exercise 5.4: Casting



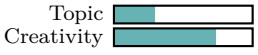
In order to convert between `int` and `long` values we have to perform a cast. But, when do we have to do this explicitly, and when can it be implicit?

- How are `int` and `long` different?
- Write a program that (i) declares `i` as an `int` variable, (ii) assigns it a value, (iii) declares `l` as a `long` variable, (iv) assigns the value of `i` to `l`, and (v) assigns the value of `l` to `i`.
- Determine experimentally when it is necessary to have explicit casts.
- Do the same thing with a `float` variable `f` and a `double` variable `d`.
- Experiment once again in order to find out where it is necessary to have explicit casts.
- Do the results of these experiments depend on the value you initially assigned to `i`/`f`?

Exercise 5.5: Definition of Expression

What is an expression?

Exercise 5.6: Assignment



Is an assignment (of a value to a variable) an expression in itself? How do you test a hypothesis on this?

Exercise 5.7: Expression vs Statement

What is the difference between an expression and a statement?

Exercise 5.8: Areas of Circles

Write a program that calculates and prints out the area ($\pi \cdot r^2$) of three circles with radiuses of 1, 3 and 5.

Exercise 5.9: Summed Circumference of Circles

Write a program that calculates and prints out the circumference ($2 \cdot \pi \cdot r$) of three circles with radiuses of 1, 3 and 5, and finally prints out the sum of these.

Exercise 5.10: Celcius to Fahrenheit

Write a program in which:

1. A Celcius temperature is assigned to a variable.
2. This temperature is converted to degree Fahrenheit and stored in another variable.
   - Formula: $T_F = 32 + \frac{9}{5}T_C$
3. Print out the conversion in a reasonable way.

Exercise 5.11: Epoch

Write a program in which:

1. Some number of seconds since a specific point in time (e.g., January 1st, 1970) is stored in a variable. Given an agreement of this starting point (January 1st, 1970), such a number can be used to represent a timestamp.
2. Convert this number to an even number of years (lets assume that there are 365 days in every year) and how many days the timestamp is into this year.
   - This way, we can convert the timestamp into something that makes more sense for humans: A combination of a year and a day. The day will then have a values between 0 and 364 (both included).
   - Ideally we would add months as well so that we have a year-month-day date. Months, however, don't have the same lenght, and we don't want to include that complexity in this exercise.
   - Added together, the two numbers (year and day) should be within 24 hours of our timestamp.
3. Print out the resulting year and day.

Verify that the program works correctly.

Exercise 5.12: Incrementing a Month

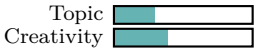
Write a program in which:

1. An integer variable is declared.
2. Assign a value to this variable that represents a month (i.e., 2 is February).
3. Print out the value of the variable.
4. Increase the value of this variable by 0.5.
5. Print out the value of the variable.
6. Increase the value of this variable by 0.5.
7. Print out the value of the variable.

Make the necessary adjustments to make the program compile by casting to `int`, run it, and describe what you observe.

Explain the printouts you observe.

Exercise 5.13: Value vs Variable



What is the relationship between a value and a variable?

Exercise 5.14: Daily Differences

Write a program that has 7 hardcoded daily temperatures. It should calculate and print out the temperature of all consecutive days (i.e., Tuesday-Monday, Wednesday-Tuesday …Sunday-Saturday).

The daily temperatures could be:

- Monday: 21.5
- Tuesday: 23.7
- Wednesday: 19.6
- Thursday: 22.5
- Friday: 25.3
- Saturday: 21.7
- Sunday: 18.9

Then, the result would be:

```
2.1999999999999993
-4.099999999999998
2.8999999999999986
2.8000000000000007
-3.6000000000000014
-2.8000000000000007
```

Why is the result wrong?

Exercise 5.15: Average Age

Topic
Creativity

Consider the following program:

```
int ada_lovelace      = 36; // https://en.wikipedia.org/wiki/Ada_Lovelace
int dennis_ritchie    = 70; // https://en.wikipedia.org/wiki/Dennis_Ritchie
int grace_hopper      = 85; // https://en.wikipedia.org/wiki/Grace_Hopper
int hedy_lamarr       = 85; // https://en.wikipedia.org/wiki/Hedy_Lamarr
int edsger_dijkstra   = 72; // https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
int douglas_engelbart = 88; // https://en.wikipedia.org/wiki/Douglas_Engelbart

float male_avg   = (float)(dennis_ritchie + edsger_dijkstra + douglas_engelbart) / 3;
float female_avg = (float)(ada_lovelace + grace_hopper + hedy_lamarr) / 3;
float avg = (male_avg + female_avg) / 2;
float diff = male_avg - female_avg;

Console.Write("Average lifespan of a male computer scientist: ");
Console.WriteLine(male_avg);
Console.Write("Average lifespan of a female computer scientist: ");
Console.WriteLine(female_avg);
Console.Write("Average lifespan of a computer scientist: ");
Console.WriteLine(avg);
Console.Write("Males live this much longer than females: ");
Console.WriteLine(diff);
```

Execute the program and explain what happens?

Write an english text (or one in any human language for that matter) where you, through technical terms describe what happens. Make sure that this text is detailed enough for another programmer to reconstruct the original code (or equivalent).

Exercise 5.16: Printf

Topic
Creativity

Figure out experimentally how the following code works. Do so by trying out different variations of the string in the last line:

```
int i = 42;
long l = 56;
float f = 3.14159F;
double d = 3.14159 * 10;
Console.WriteLine("i = {0} \nl = {1,4} \nf = {2} \nd = {3,6:0.00}", i, l, f, d);
```

In most programming languages, this function would have been called `printf`. But Microsoft have for some reason decided to make C# stand out in this regard.

Exercise 5.17: Definition of Boolean

What is a `boolean`?

Exercise 5.18: Creation of a Boolean

Which operators can be applied to *expressions* in order to create a `boolean` value?

Exercise 5.19: Decision of Purchase

Consider the following code:

```
double price = 599.95;
double budget = 1000.0;
bool requiredReading = true;
bool shouldBuy = price < budget && requiredReading;
```

Explain the last line, and focus on:

- In which order is what calculated?
- Which values (named and otherwise) are each operator evaluated on?
- What are the types of these values?
- What does the variable `shouldBuy` represent?

Exercise 5.20: Dice

Write a program in which:

1. the result of a roll of a dice is stored in a variable named `dice`.
   - Which type should the variable be declared as?
   - Chose a value and initialize the variable to it.
2. Declare a boolean variable and assign it a value that represent whether the value in `dice` is even and greater than 3.
3. Print out the value of this boolean variable.

Exercise 5.21: Manual Type Inference

Assume that the following line of code compiles cleanly (no warnings or errors):

```
// this line contains a declaration of a
int b = a + 1;
```

What can you tell about the type that `a` is declared as?

More specifically:

- Does this mean that `a` is an integer?
- Would the code compile if `a` is a non-numeric datatype?
- Would the code compile if `a` is a numeric datatype other than `int`? Which ones?

Chapter 6

Flow Control

So far, we have worked with a strict sequence of statements. All of these statements are evaluated, in order. This is a quality that we will often come to rely on. But it is not expressive enough. We need to introduce choice and repetition to our vocabulary, and our toolbox. Figure 6.1 illustrates the set of basic flow abstractions that we will cover in this chapter. It is up to the individual programmer to decide how to combine these in order to implement desired behavior.

6.1 Choices in Expressions

The perhaps simplest expression of a choice in C# is that of a conditional expression. In practice, this is implemented by the ternary operator. The syntax definition for that ternary operator can be found in syntax 6.1. Here, one condition decides which of two expressions to evaluate. These two expressions must be of types that satisfy the type of the full ternary expression.

The ternary operator is used to express conditional expressions. That is, expressions whose evaluation takes one of two paths depending on the evaluation of a condition. For instance, given a boolean variable called `leap_year` that defines whether we are dealing with a leap year,

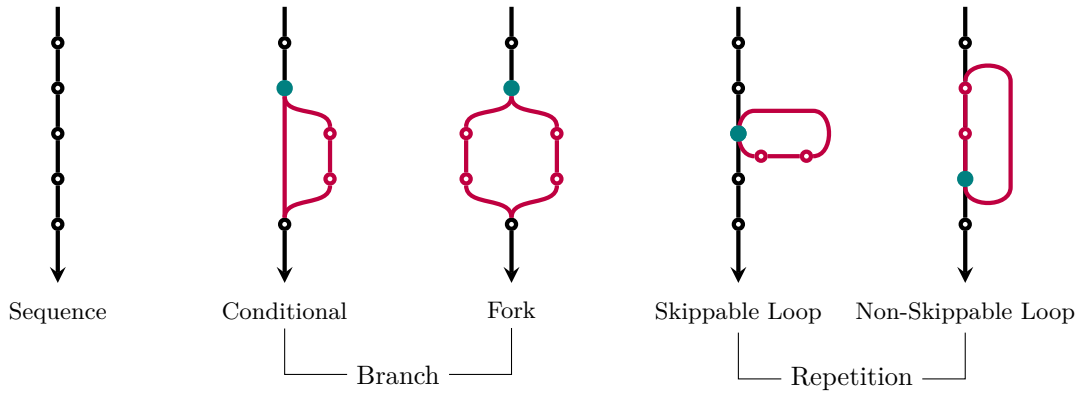


Figure 6.1: Basic statement flow abstractions. The execution path can be controlled by making a choice at runtime. The point of the choice is marked in teal and the optional dynamic "routes" are marked in purple.

Syntax 6.1: Ternary operators in expressions

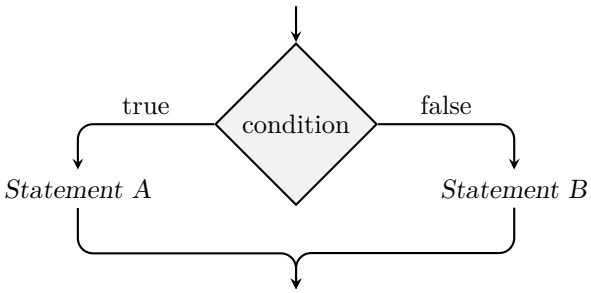


Figure 6.2: Flowchart of a generic choice.

the following expression defines the length of the month of February:

```
int length = 28 + (leap_year ? 1 : 0);
```

6.2 Choices in Logic

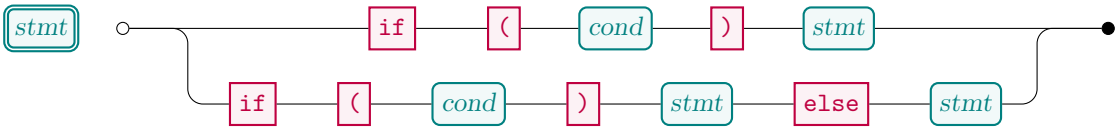
While not particularly common, choices in expressions can be a very convenient way of expressing how certain values are defined in a conditional way. We would like the same ability in the context of a statement instead of an expression, or in other words, we would like conditional statements.

Branching is a term that encompasses any way of choosing one statement to evaluate from a number of statements. In this section, we will cover the most common means of achieving this. First, we cover the simplest options, and then we gradually add more complex constructs by build on that understanding.

6.2.1 `if` and `else`

As illustrated in figure 6.2, the decision on which branch to take is made based on the evaluation of a condition. That condition evaluates – by definition – to either `true` or `false`. As there is two possible outcomes, this represents two possible futures; a fork in the road. You are either in the situation where the condition evaluates to `true` (and you go down one road), or you are in the situation where the condition evaluates to `false` (and you go down a different road). Eventually, these roads will merge. This happens after the evaluate of a single statement.

A common pattern emerges in which the else statement is empty: "if this condition is met, then do this". Many programming languages include a construct in which the false statement is removed. Instead, it simply bypasses the true statement if the condition evaluates to `false`.



Syntax 6.2: Statements for branching

6.2.1.1 C#

6.2.1.2 Functional Programming

6.2.1.3 Reverse-Order Conditioning

The Perl programming language has a feature that allows programmers to write branch statements in *reverse order*. They are not forced to do so. It is just an option that is a available to them should they decide that this is the best way of expressing themselves. In this way, it is a bit like expressing a sentence in passive voice. Given a definition of `$var`, the following is valid Perl code:

```
print "Variable is positive!\n" if $var > 0;
```

6.2.2 Blocks

Technically, branching on a single statement is all we need. But this would not be particularly convenient. Imagine us having to do two things if a condition holds true: We would have to check a condition and if true do the first thing. Then we would would have to check the same condition again, and if true do the second thing. This comes with three significant downsides, namely:

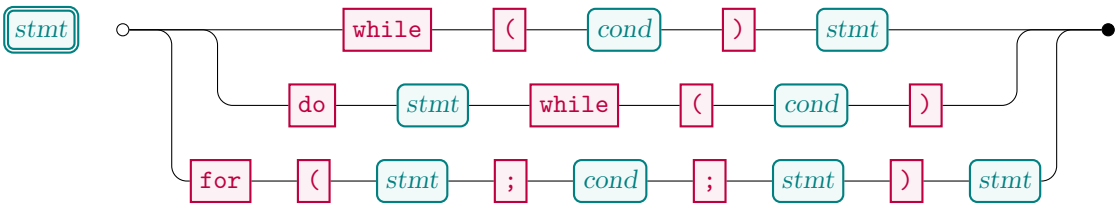
1. The condition needs to be evaluated twice. This takes some amount of time and could involve complex side effects.
2. The code would quickly get complex to understand.
3. It would get significantly more convoluted if the first thing modifies a variable that takes part in the condition.

Most imperative programming languages have a notion of a block. This is a sequence of statements that take the place of a single statement. So, by employing a block the one statement at a true branch of an `if` statement could become a sequence of statements. And likewise for every syntactic occurrence of a statement (e.g., the false branch of an `if` statement).

6.2.2.1 C#

In C#, we use blocks all the time. As illustrated in figure 6.3, is simply a sequence of statements wrapped in a pair of curly braces. By convention, we usually indent those statements one level.

Syntax 6.3: Block statements



Syntax 6.4: Statements for repetition

6.2.2.2 Functional Programming

6.2.3 Dangling Else

6.2.4 `unless`

The most important mechanics for braching are the `if` and `else` constructs. They are, however, not the only ones. But not all are available in every language. The `unless` construct is a bit obscure. Few languages support it, and a few more allows you to define it. For instance, in C you can define it like this:

```
#define unless(cond) if(!(cond))
```

This shows exactlt what it does: An `unless` statement takes a condition, and it does exactly the same as an `if` statement with the negated condition. As C# does not have a `unless` construct, we won't dig further into it.

6.2.5 Chaining Branches

6.2.5.1 C#

6.2.6 `switch` and `case`

6.2.6.1 C#

6.2.6.2 Functional Programming

6.3 Repetition

6.3.1 `while`

6.3.2 `do-while`

6.3.3 Converting Between `while` and `do-while`

6.3.3.1 Problem of Repetition

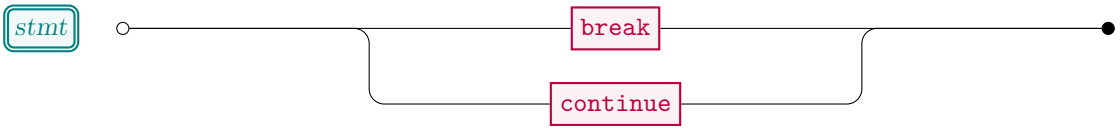
So, now we have two ways of doing (more or less) the same thing. That begs the question: Which option should we pick? More importantly, why should one option be chosen over the other? And which qualities should we use for *scoring* each of the two options?

Two qualities stick out: Readability and maintainability. Readability essentially means how easy the code is to read, and maintainability boils down to how easy the code is to maintain. We can employ a number of (more or less well-defined) metrics for each. In this particular situation, they both happen to point in the same direction. This stems from one of the options having a duplicate element. For instance, the way we went from a do-while loop to a while loop was to place a copy of the loop body before the loop. From a readability perspective this creates mode lines of code, and it burdens us with the need to both recognize that those two pieces of code are identical and that they are part of the same loop. The latter part is explicit in the do-while option. From a maintainability perspective, duplicated code is always risky as the developer is burdened with the need to manually keep the two copies in sync. Any failure to do so will result in a bug. In summary, unless you have a good reason not to, you should simply use the most compact version.

6.3.4 `for`

It didn't take long for a common pattern of while loop usage to emerge. In this pattern, a loop variable is declared before the loop, at the end of the loop this variable is adjusted and the condition check whether this variable has passed some value. This is, for instance, how you would loop through all integers between two values.

If generalized, a template can be synthesized from this pattern. We ne an initialization statement, a condition, an update statement, and a body. The generalization comes from us using the generic construct of a statement for initialization and updates. Note that by choosing to build on statements, we also gain the option of having the initialization and update statements be expressed without any shared variables. By generalizing the template we ensure that it is as applicable as possible. A new kind of loop – the `for` loop – is designed specifically to accommodate this template.



Syntax 6.5: Control structures for repetition

6.3.5 C#

6.3.6 Control Structures

6.4 Exercises

Exercise 6.1: Epoch

Topic
Creativity

Write a program in which:

1. you use your work from exercise 5.11 as a starting point.
2. you declare a variable which value represents a number of seconds since newyear.
3. Based on tthe value of this variable, you calculate which month and day it corresponds to. Assume that all months are 30 days long.
4. If the day happens to be the day some culture of your choice has designated as Christmas, then print out "It's Christmas!".

Exercise 6.2: Epoch Diff

Topic
Creativity

Extend your solution for exercise 6.1 to – if it isn't Christmas – to print out how long time it is until Christmas.

Exercise 6.3: Christmas Sale

Topic
Creativity

Write a program in which:

1. A variable is declared and initialized to the value 21816000. This value represents a number of seconds since newyear (assuming that all months are 30 days long).
2. Some other variable represents a price and has a value of 599.95.
3. If it is Christmas, there is a 30% rebate. Find a reasonable way of determining when it is Christmas.
4. Calculate the current price (including any rebate) and store this in a third variable.
5. Print out the value of this variable.

Be sure to test the logic you have written be assigning the first variable different values. Which values would be relevant to test?

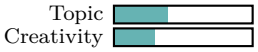
Exercise 6.4: Length of Month

Write a program in which:

1. A number of a month is stored in a variable. This variable acts as input to your program.
   - Which name would be appropriate for this variable?
   - Which type would be appropriate for this variable?
2. Write some code that determines how many days there is in the input month (lets assume no leap year).
3. Prints out the result.

Use a calendar to verify that your implementation is correct.

Exercise 6.5: Holidays



A common educational calendar will have the following holidays:

- ***Autumn Holiday*** October
- ***Christmas Holiday*** December
- ***Spring Holiday*** April
- ***Summer Holiday*** July + August

Write a program in which:

1. A month is given in the form of an integer via a variable.
2. Depending on the value of this variable, either print out the name of the holiday (if there is a holiday in the month), or simply "Hard work" (if there isn't).

Exercise 6.6: Celcius to Fahrenheit

Write a program in which:

1. When executed, the program should write out a table of matching Celcius and Fahrenheit values.
   - Formula: $T_F = 32 + \frac{9}{5}T_C$
2. There should be one matching pair per line.
3. The list should start at -5°C and end at 40°C.

4. The list should have one line for ach 0.5°C.

Exercise 6.7: Celcius to Fahrenheit in Reverse

Topic
Creativity

Rewrite the program from exercise 6.6 to print out the lines in the reverse order so that the lines go from 40°C to -5°C.

Exercise 6.8: Celcius to Fahrenheit Alternatives

Topic
Creativity

Make two other variants of the program from exercise 6.6; one for each to the remaining loop types. Rewrite your solution to use this loop type.

Exercise 6.9: Areas of Circles

Topic
Creativity

Write a program that calculates and prints out the area ($\pi \cdot r^2$) of three circles that have the radiuses 1, 3 and 5.

Notice: This is a repetition of exercise 5.8, but this time you have a larger toolbox at your disposal.

Exercise 6.10: Primes

Topic
Creativity

Write a program that calculates all prime numbers below 1,000,000 and prints out the largest.

Hints (unless you want more of a challenge):

- A positive integer is a prime number if, and only if it is not divisible by integers other than 1.
- Use a divide-and-conquer strategy[1] whereby you subdivide the task into these subtasks:
  - Iterate through all positive integers below 1.000.000. Don't we have a construct that can do this?
  - Determine whether a given positive number is a prime.
  - Print out an integer number (if it is prime).
- In order to determine whether a given positive integer is prime, you can, once again, divide an conquer:
  1. Declare a `boolean` variable named `is_prime` and initialize it to `True`.
  2. Iterate through all integers from (and including) 2 until (but not including) 1,000,000.
  3. Check whether the prime candidate is divisible by each of these numbers. If that is the case, you should assign the `False` value to `is_prime`. But how do you determine one number is divisible by another?
     - You try it, of course!

[1] https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm

   - If a number is divisible by another, then there won't be a remainder if you do an integer division and the modulo operation (via the `%` operator) will evaluate to zero.
   - As an alternative, one could exploit that the integer division will result in a rounding error so that $(a/b) \cdot b \neq a$.

4. At this point, `is_prime` represent the truth value of whether the prime candidate is an actual prime.

Chapter 7

Basic Datastructures

Keeping track of a single value is trivial. But once you start having a handful of values to juggle, you begin to feel a burden. Yet, by following good practices for naming your variables, it shouldn't be a big deal. However, as the number of values that we need to deal with grow, that solution model starts to break down. Imagine a program that we are writing a program that has to do some sort of analysis on the historical daily temperatures some calendar year. That is 365 (or 366) values right there, and that isn't even a remotely large number of values. Even with a good naming convention, keeping track of that many variables will be a mess. But it doesn't stop there: Some values are related and we often have to treat them that way. In this chapter we look into how we can structure data in ways that allows us to reason about much larger number of values through code. This is a form of scalability.

7.1 Complex Types

Primitive types cannot meaningfully be subdivided further. Yes, an `int` can be split into bits, but why would I care what the value of the 7th bit is. It gets a bit more murky with floats. Here, I might want to look into the sign or the magnitude (i.e., the exponent). But all the floating-point instructions of our processor operate at the unit of a full floating-point value (i.e., sign, mantissa and exponent). So, atomic is an apt qualifier for such types. After all, real atoms also consists of smaller elements. In C#, such types are called primitive.

Complex types are complexes of data. They contain (or at least may contain) multiple elements of data that can in turn be atomic or complex. Those complex types can be used to arrange and expose these elements in different ways. This chapter will cover the most basic variations. Because these types structure data, we call them datastructures.

7.2 Reference Types

From the compilers point of view, atomic types are easy to work with. They fit into a register, and can be read from and written to memory in a single operation. Choosing one of two values is trivial. A complex type, on the other hand, does not fit into a single register, and so the compiler will have to decide how large a part of the data needs to be loaded from memory into registers, and how much needs to be written back after we are done with it. Later on, you will learn that multiple things can go on at the same time within a single program. When this happens, the

implementation of this needs to ensure that a consistent view of the data is presented to all of these ongoing things.

A reference is an address (in memory) to some datastructure, and such a reference can be used as a value. References are atomic values and do fit into a single register. Variables of reference types thus hold the reference (i.e., the address) and not the value (i.e., the datastructure). Two reference type variables can refer to the same data. All it takes is for them to hold the same address. Nothing changes with respect to the data being referenced: It still doesn't fit into a single register. But we now have the option of copying references, and that operation is both trivial and fast.

Another solution is simply to accept that datastructures cannot be represented in a single register, and deal with the complications that arise from that. This is essentially what you have done so far when you had two variables and needed to make a copy of that combination. Luckily, those complications can be handled by the compiler. So, while making datastructures reference types is the natural choice, they can also be made value types. C# offers mostly reference type datastructures, and these are the ones we will be focusing on.

So, types where a variable is mapped to hold values in register(s) are called value types, and types where variable is mapped to hold a reference to an address in memory is called a reference type. Copying a reference type copies the reference, and any modification to whatever it refers to is visible through all reference copies to these data.

7.3 Structured Data

Let's start with an example. We need to create a program that helps us navigate a house. For this we need to model the concept of a door. A door, in the context of the this problem, has a height of type `int`, a width of type `int` and `bool` for wether it is open. We would like to be able to represent multiple doors and to refer to them individually. This example of a door is an example of structured data, or – as they are often called – structs. To generalize, structs are groupings of typed and individually named values that are stored contiguously in memory.

Looking at figure 7.1, we can see how a concrete instance of a door is laid out in memory. The whole structure is placed at address 273141, and thus takes up space from this address onwards. As the struct contains two `int` values (each of 4B) and one `bool` (of 1B), the total size of the struct is 9B. Accordingly, the last address occupied by the struct is $273141 + 9 - 1 = 273149$. The compiler does this for us, but is also keeps track of the offsets of the individual names. For instance, when we in code refers to the `height` attribute in our code, the compiler knows that this field is stored at an offset of 1B in memory (i.e., 273142), and it will produce code that does the lookup on our behalf. So, to look up the `height` field, all we need is a reference to the struct.

7.3.1 C#

Note: C# has a construct that it calls a struct, and that definition deviates a bit from what we have covered here. Pedagogically it doesn't fit nicely into a clean introduction to the language, and it is neither central to the language or the object-oriented paradigm. Because of this, we have decided skip the C# variant of the concept. Just know that when people talk about structs in C#then they mean something slightly different. But why call them structs then? Well, that is what everyone else calls them.

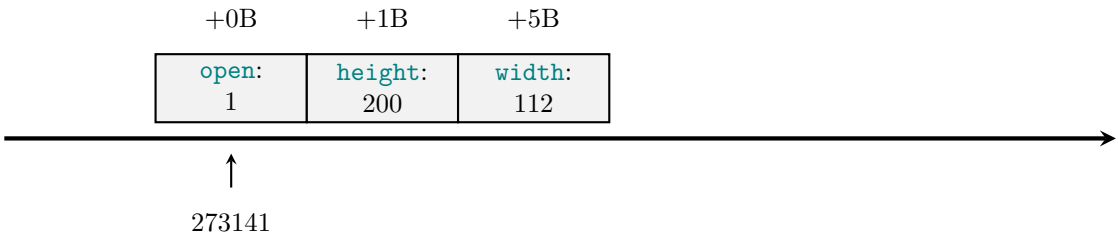
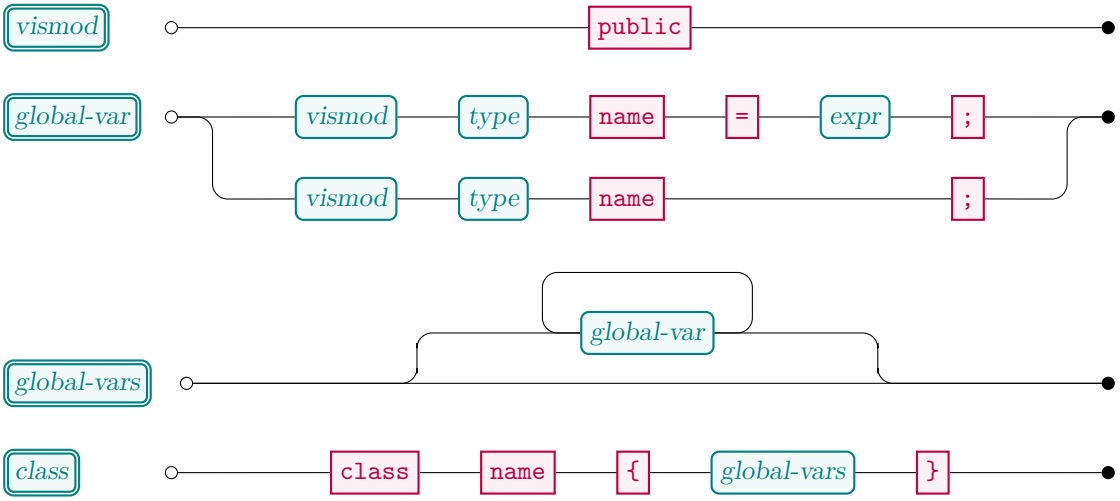


Figure 7.1: The positioning and layout of a `Door` struct in memory.



Syntax 7.1: Structured data

7.3.2 Colors

Light is made up of photons, and each photon has a wavelength. The wavelength of the photons determines which color our eyes will observe. A wavelength of 380nm is violet and one of 740nm is red. In between these you have all the colors of the rainbow, in the order of the rainbow. Coincidence? Just outside of the visible spectrum, you will find infra-red and ultra-violet. Our eyes are, however, constantly bombarded by photons, so in reality we always observe a distribution across this spectrum. Our eyes, though only has three (or four) tunings of receptors. These, roughly corresponds to red, green and blue. So, your eyes (and brains) need to do a lot of interpretation.

In a computer, colors are typically represented using three values; one for red intensity, one for green intensity, and one for blue intensity. That somehow matches the model of our eyes … We say that these are the *components* of the color. There is a number of ways to express such an intensity. It seem natural to use a number between 0 and 100, or perhaps a number between 0.0 and 1.0. However, due to the way processors work on integers, these boundaries are not particularly special. Typically, a byte is use to represent each component of the color. That means that we have the ability to express $2^8 = 256$ different intensities of each component, or a

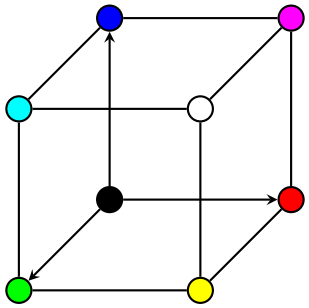


Figure 7.2: RGB color cube.

total of $256^3 = 16777216$ different colors. The value zero is used to represent complete darkness. Full intensity is then the value 255.

This constitutes a color space whereby all colors fit within a color cube spanned by the red, green and blue color vectors. Hence, it is named the RGB color space. Figure 7.2 depicts this space. It is a direct fit for how GPUs represent images. Other color spaces exist though, and there are good reasons for their existence. Other popular color spaces include HSV (which is shaped like a cylinder) and XYZ (which is shaped like an tongue). Formulas exists for converting between these. Each of these color spaces represents practical simplifications of what light really is.

7.3.2.1 C#

7.4 Sequences in Data

Structured data are laid out in sequence, and every element can be named through a symbolic field name. For many purposes, this is great. It does, however, have two consequences that will at times work against us. Those are (i) that we need to have explicitly defined a name for each field at compiletime, and (ii) that doing something with every element requires us to keep track of the order of those names through other means.

Addressing individual elements through symbolic names certainly has its advantages. But this is not always the case. Imagine a company that stores doors. One day they may have 80 doors in stock. We certainly don't want to create symbolic names for each of these. Then we would have to update our code every time a door is shipped or a new shipment of doors have arrived. What we really want is for us to be able to express our code is such away that it automatically adapts as the stock fluctuates. To achieve this quality, we need an indexed datastructure. That is, a datastructure that contains elements that can be named through an integer value (that typically represents its position in that structure). The ability to do so allows us to loop over all elements of the structure and that is – as we will see – and extremely useful ability.

7.4.1 Arrays

An array is a simple datastructure which contains zero or more elements of the same size. These elements are laid out contiguously in memory as depicted in figure 7.3. Given a pointer to an array (i.e., the address of the beginning of the array allocation) p, an index within that array i and a an element size s, we can find the address of that element by:

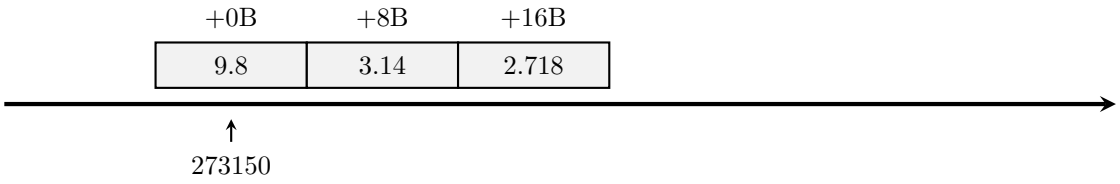


Figure 7.3: The positioning of an array in memory.

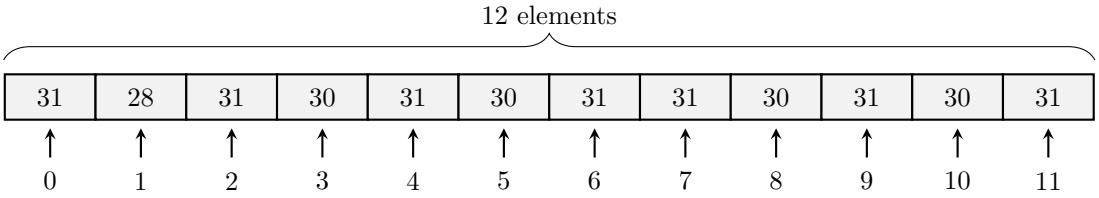


Figure 7.4: The indexing of an array.

$$\text{index}(p, i, s) = p + i \cdot s$$

7.4.1.1 Indexing

7.4.1.2 Arrays of Arrays

We can store values of any type in an array, as long as all elements have the same size and – depending on language – type. This includes array types. We can thus have arrays of arrays of some type. Here, we differentiate between the outer array and the inner arrays. The rules of an array state that all elements must have a common type. However, arrays are reference types and the type does thus not include the array length. Because of this, the inner arrays do not necessarily have to have the same length.

7.4.1.3 Multidimensional Arrays

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} i \bmod width \\ \lfloor i/width \rfloor \end{pmatrix} \tag{7.1}$$

$$i = y \cdot width + x \tag{7.2}$$

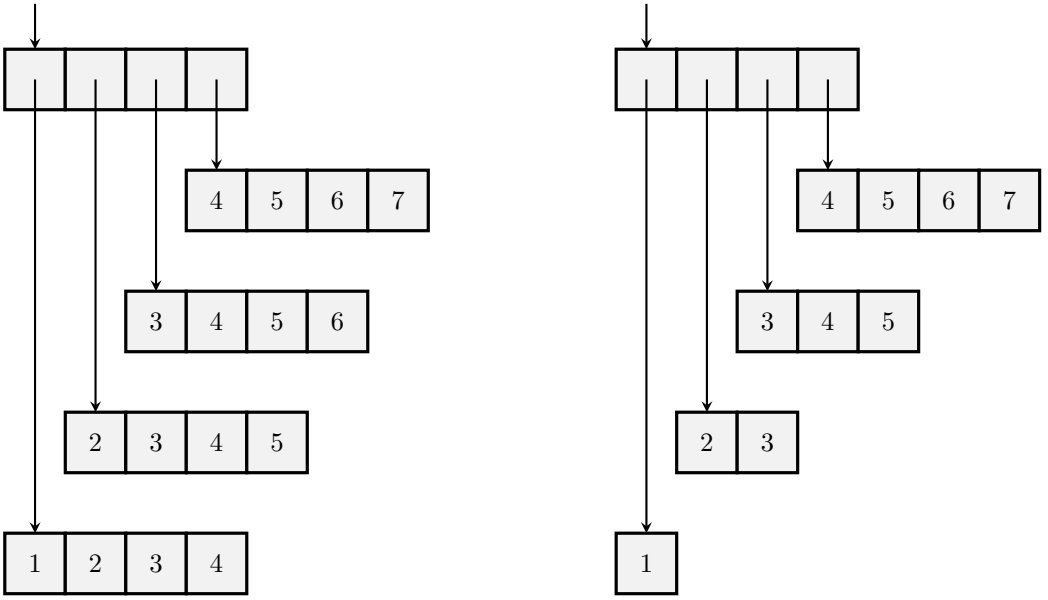
7.4.1.4 C#

7.4.1.5 Nesting vs Multiple Dimensions

7.4.2 Linked Lists

7.4.2.1 Functional Languages

Most functional languages make heavy use of pattern matching. This makes the structure of linked lists become very apparent. Typically, they provide a number of patterns to match linked



(a) Nesting into square. (b) Nesting into triangle.

Figure 7.5: Nesting arrays in arrays.

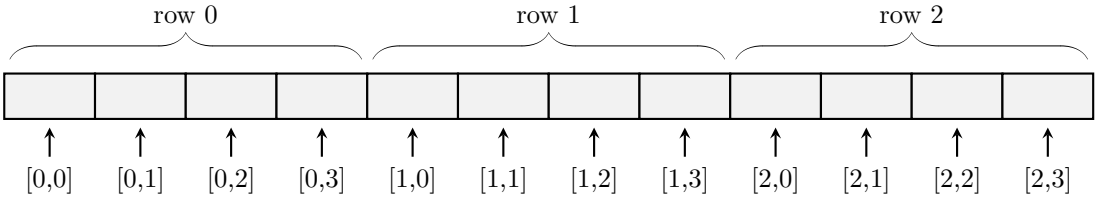


Figure 7.6: The memory layout of a 2d array.

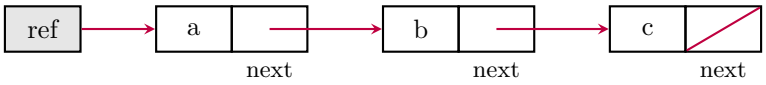
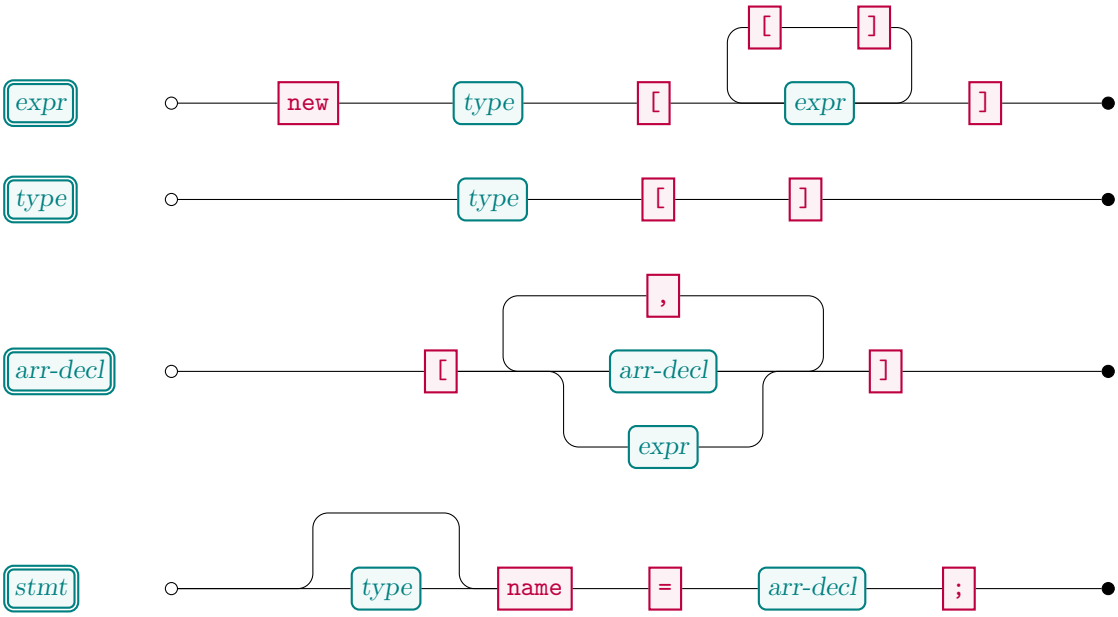
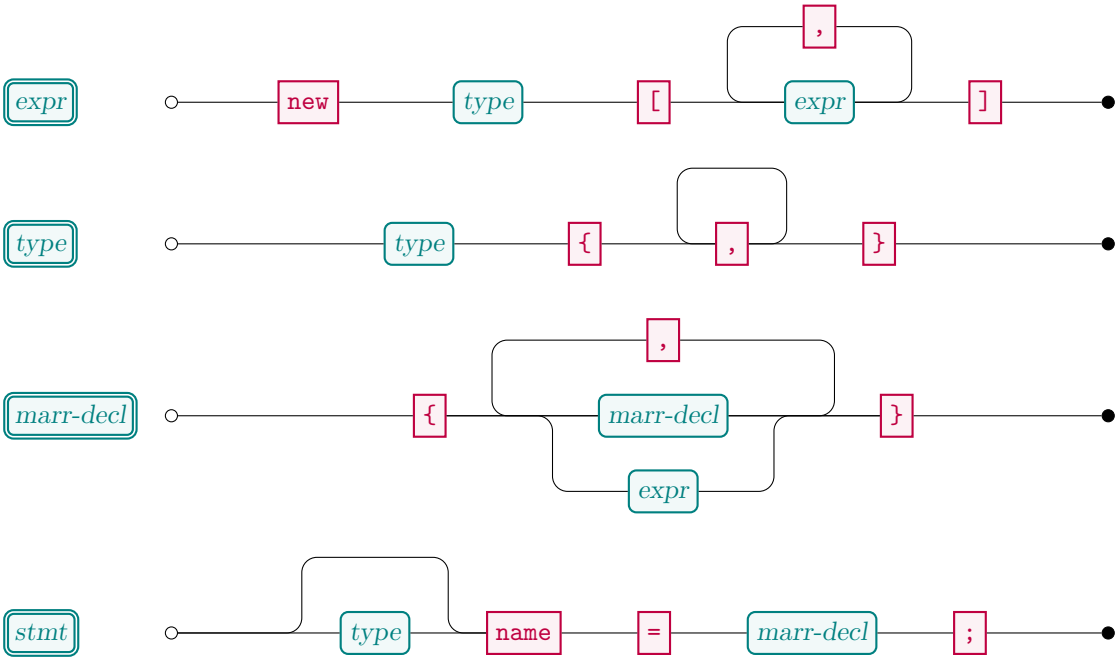


Figure 7.7: Structure of a linked list.



Syntax 7.2: Arrays



Syntax 7.3: Multidimensional arrays

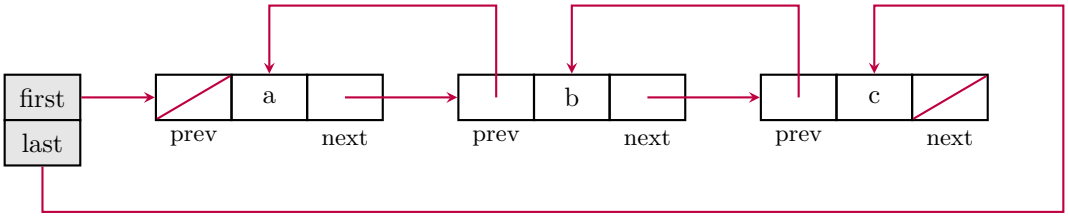
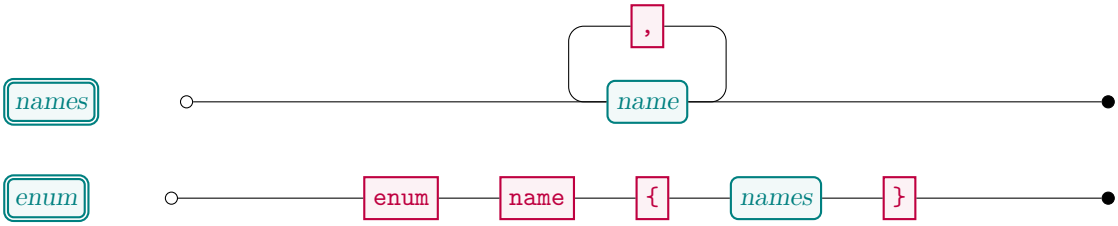


Figure 7.8: Structure of a doubly linked list.



Syntax 7.4: Enums

lists. The most revealing one matches the head and the tail of the linked list. In Elixir it looks like this:

```
iex(1)> l = [1,2,3,4]
[1, 2, 3, 4]
iex(2)> [head|tail] = l
[1, 2, 3, 4]
iex(3)> {head, tail}
{1, [2, 3, 4]}
```

7.4.3 Doubly Linked Lists

7.4.4 Looping Sequences

7.4.5 Nesting

7.4.6 Points

7.5 Enumerations

7.5.1 State Machines

7.5.2 String Parsing Example

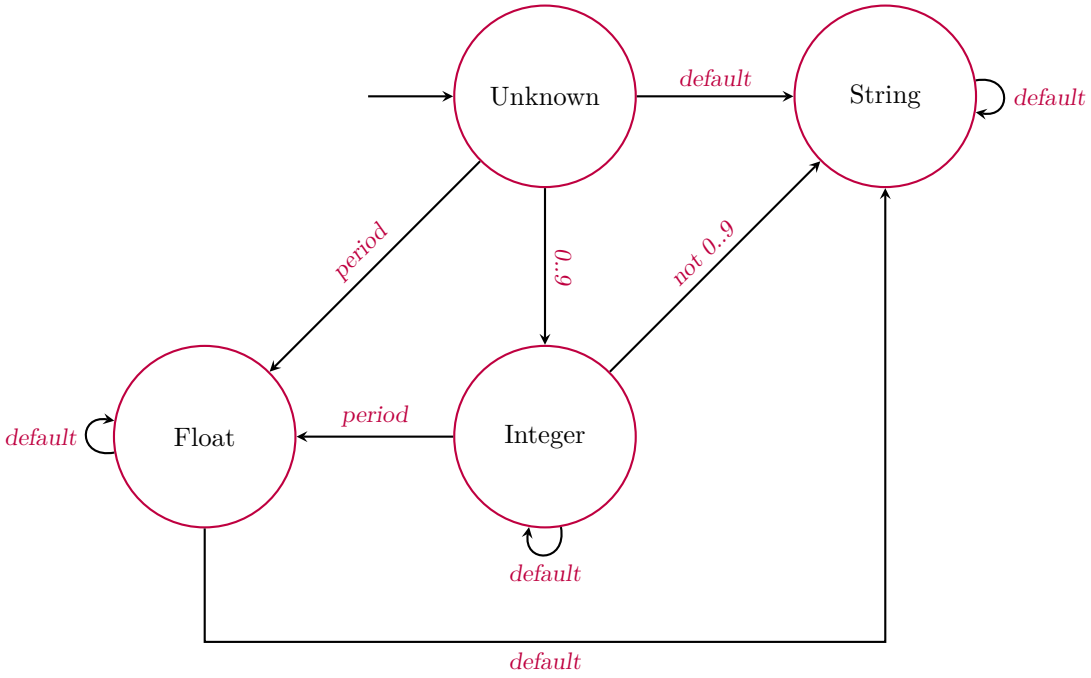


Figure 7.9: State machine for parsing strings.

7.6 Case: Dynamic Arrays

7.6.1 C#

7.7 Exercises

Exercise 7.1: Definition of Array

Topic
Creativity

What is an array?

Exercise 7.2: Array Usecase

Topic
Creativity

When should one use an array?

Exercise 7.3: Array Type

Topic
Creativity

How is the type of an array linked to the type of the data that it can contain?

Exercise 7.4: Largest in Array

Topic
Creativity

Write a program hat finds the largest number in an array of the `int[]` type, and prints out the index of this number. You decide on the contents of the array.

Exercise 7.5: Size of Array Allocation

```
Category get_category (string input) {
  Category state = Category.Unknown;

  foreach (char c in input) {
    switch (state) {
    case Category.Unknown:
      if (c=='.') {
        state = Category.Float;
      } else if (Char.IsDigit(c)) {
        state = Category.Integer;
      } else {
        state = Category.String;
      }
      break;
    case Category.Integer:
      if (c=='.') {
        state = Category.Float;
      } else if (!Char.IsDigit(c)) {
        state = Category.String;
      }
      break;
    case Category.Float:
      if (c=='.' || !Char.IsDigit(c)) {
        state = Category.String;
      }
      break;
    case Category.String:
      break;
    default:
      Console.WriteLine("Error, unknown state: "+state);
      break;
    }
  }

  return state;
}

string[] inputs = new string[] {
  "10.h",
  "10.7",
  "10",
  "Hello, world"
};

foreach (string input in inputs) {
  Category category = get_category(input);
  Console.WriteLine(category);
}

enum Category {
  Unknown,
  String,
  Integer,
  Float,
}
```

Figure 7.10: String parsing example.

Topic
Creativity

When creating a new array of a particular size (without assigning values to the elements), which syntactical element is used to declare how many elements the program should allocate space for?

Exercise 7.6: Multiplication Table

Topic
Creativity

Write a program in which

1. An integer variable named `size` is declared and initialized to some value under 30. You pick the concrete value.
2. An array is created with a length that matches the value of `size`.
3. This array is filled up with the 3 table. That is, an element at index n must have the value $3 \cdot n$.
4. Print out one or more elements to verify the correctness of what you have done. How do you pick which elements to observe in such a way that you gain a lot of value from these observations while limiting the number of observations?

Exercise 7.7: Sudoku Puzzle

Topic
Creativity

How would you represent a Sūdoku[1] puzzle in C#?

How is this datastructure laid out in memory?

Exercise 7.8: Areas of Circles

Topic
Creativity

Write a program that calculates and prints out the area ($\pi \cdot r^2$) of three circles that have the radiuses 1, 3 and 5.

Notice: This is a repetition of exercise 6.9, but this time you have a larger toolbox at your disposal.

Exercise 7.9: Daily Differences

Topic
Creativity

Write a program that has 7 hardcoded daily temperatures. It should calculate and print out the temperature differences of all consecutive days (i.e., Tuesday-Monday, Wednesday-Tuesday … Sunday-Saturday).

The daily temperatures could be:

- Monday: 21.5
- Tuesday: 23.7
- Wednesday: 19.6
- Thursday: 22.5

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

- Friday: 25.3
- Saturday: 21.7
- Sunday: 18.9

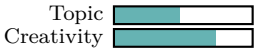
Notice: This is a repetition of exercise 5.14, but this time you have a larger toolbox at your disposal.

Exercise 7.10: Length of Month

Write a program, that given a month number in a variable prints out the number of days in this month. Do not consider leap years.

Note: This is a repetition of exercise 6.4, but this time you have a larger toolbox at your disposal.

Exercise 7.11: Primes



Write a program that calculates all primes below 1,000,000 and prints out the largest.

Do this by implementing the Sieve of Eratosthenes[2].

The square root of `i` is calculated as `Math.Sqrt(i)`.

Note: This is a progression of exercise 6.10. Here, you will find a definition of what a prime number is.

Exercise 7.12: Calendar

Write a program in which

1. A variable is initialized to be an array containing the number of days in each of the 12 months in a normal year. The first element will then contain the number of days in January.
   - What should the type of this array be?
2. Another variable is initialized to be an array containing the number of days in each of the 12 months in a leap year.
   - What should the type of this array be?
3. Iterate through the years 2000 through 2020:
   a) Use a new variable called `pointer` to refer to the array that matches the year from the iteration.
      - What should the type of this variable be?
      - How much data is copied here?
      - **Hint:** You can simplify the leap year rules so that any year divisible by four is a leap year.

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

b) For every year, the contents of the array referenced by `pointer` is printed to the screen.

Exercise 7.13: Calendar Prettyprinting

Topic
Creativity

Write a program in which

1. A datastructure is created to hold the calendar of a single year. One needs to be able to index it using a date (month and day) in order to extract which weekday it is (e.g., Monday).
   - What is the type of this data structure?
   - How do you initialize a variable of this type?
   - How can you make sure that the contents is correct?
2. Print out the contents of this data structure in a "nice" way.

Exercise 7.14: Sūdoku Checker

Topic
Creativity

Write a program in which

- A variable is initialized to hold a filled-out Sūdoku[3] puzzle.
  - How can we represent a cell that isn't filled out?
  - What should the type of this variable be?
  - How do you declare it?
  - How do you initialize it?
  - **Hint:** Find a filled out (aka solved) sūdoku puzzle online.
- write code that checks if the puzzle contains a correct solution.
  - **Hint 1:** This is the cass when all of the following hold:
    * Every cell is filled out.
    * No row contains two cells with the same value. Alternatively: All numbers 1-9 are present in all rows.
    * No column contains two cells with the same value. Alternatively: All numbers 1-9 are present in all columns.
    * No 3x3 group contains two cells with the same value. Alternatively: All numbers 1-9 are present in all 3x3 groups.

    Hint 2: In order to check if all the numbers 1-9 exist, one could create a `boolean[9]` representing wheter each of the numbers have been found, initialize this to all `false` values, iterate through all relevant cells and et the index corresponding to the value to `true`. If any `false` values are left, then at least one number is missing, and the sūdoku is not solved.

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

- The result of this check is printed to the screen.
  - Does the program also give a correct answer when you give it a wrongly filled out sūdoku puzzle?

Exercise 7.15: Person

Define a *struct* that can be used to represent a person. Which fields should it have? Perhaps persons need to have a name and an age in our program. A name can be stored in a `string` and an age could be an `int`.

Demonstrate that you can use this type.

Exercise 7.16: Direction

Define an `enum` to represent a direction that can be either north, south, east or west.

Answer the following questions, write the code, and make sure that it compiles:

1. What would be a good name for this `enum`?
2. How should the different values be named?

Exercise 7.17: Units

In this exercise we will be working with different units of distance. Our program will have an input value (measured in meters), and we want to convert it to some unit as specified through another variable.

Subtask 1

What should we call that other variable, and what would be a reasonable type for it?

We want to be able to represent the units m, cm, mm and inches.

Subtask 2

Next, we need a way mapping the unit some expression that converts between the meters of the input and the output unit. For distances, this is simply a factor.

Subtask 3

Write code that defines the type from subtask 1, implements the mapping functionality from subtask 2, and uses this to convert an input value of 1.234 meters to mm.

Chapter 8

Code Reuse

In the last chapter, we covered branching and loops as tools for control flow. This gave us mechanisms for selective and repeated execution of statements. The repetition constructs allows us to repeatedly execute some segment of our code from the same location. That is, every time we get to a specific part of a look, we execute these lines. But what if there were two places where we wanted to execute those very same lines? We certainly could duplicate those lines and call it a day, but that solution doesn't scale. You will soon experience situations where you yourself have tens of such locations, and accept that others will have situations where they have to deal with tens of thousands of such locations (some of which are outside of their control). Technically, we could have that many copies, but that would be expensive in terms of memory and – more importantly – impose a severe maintenance burden on us: What happens when we want to extend that functionality or find a bug in it. Then we need to find all of these potentially thousands of locations and update them individually. This is a task that humans are ill-equipped to deal with. For us to have any certainty of all of these updates being performed without introducing new bugs we would need the support of tooling.

That tooling turns out not to be necessary. In fact, seasoned programmers usually try to avoid the code duplication problem. Instead, they package commonly used functionality in something called a function. A function is a parameterized piece of code that typically sits somewhere outside your programs main flow. If has a well-defined interface and a descriptive name. Execution can be temporarily transferred to it from any location in your code. We refer to this as *calling the function*. This is essentially what procedural programming is.

Functions – like many of the concepts that we will introduce later on – give structure to our code in ways that allows a human to navigate it. Instead of having one large program of thousands or even millions of lines of code that we need to have an overview of all the time, we get to a situation where we can often choose to focus on an individual function of perhaps 10-50 lines. With our limited capacity for handling large amounts of information, such structure is necessary for us to scale the increase of complexity.

8.1 Mathematical Functions

In calculus, a function represents a mapping from the set of real numbers to the set of real numbers. That mapping is usually expressed through a mathematical formula and can be expressed as on a 2d coordinate system. Some functions operate in a higher dimensionality. For instance, some function may map two of the sets of real numbers to the set of real numbers. Such a

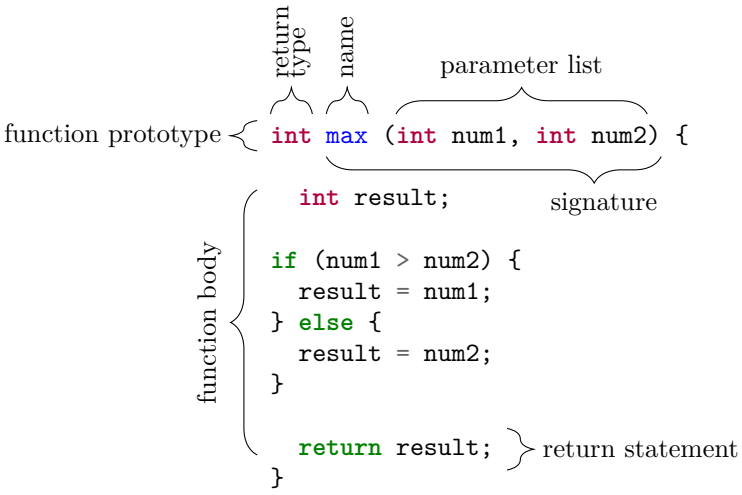


Figure 8.1: Anatomy of a function.

function could be illustrated in 3d.

By taking a few liberties, we can redefine a concrete mathematical function as a mapping from a list of values to a single value. The list of input values are matched against and assigned to the symbolic names of the function definition. The mapping itself is expressed – if we put on our programming glasses – as an expression over these symbolic names.

8.1.1 Linear Functions

In calculus, a linear function that, when illustrated forms a non-vertical straight line in a 2d coordinate system and follows the form of:

$$f(x) = a \cdot x + b$$

8.1.2 Area of Rectangle

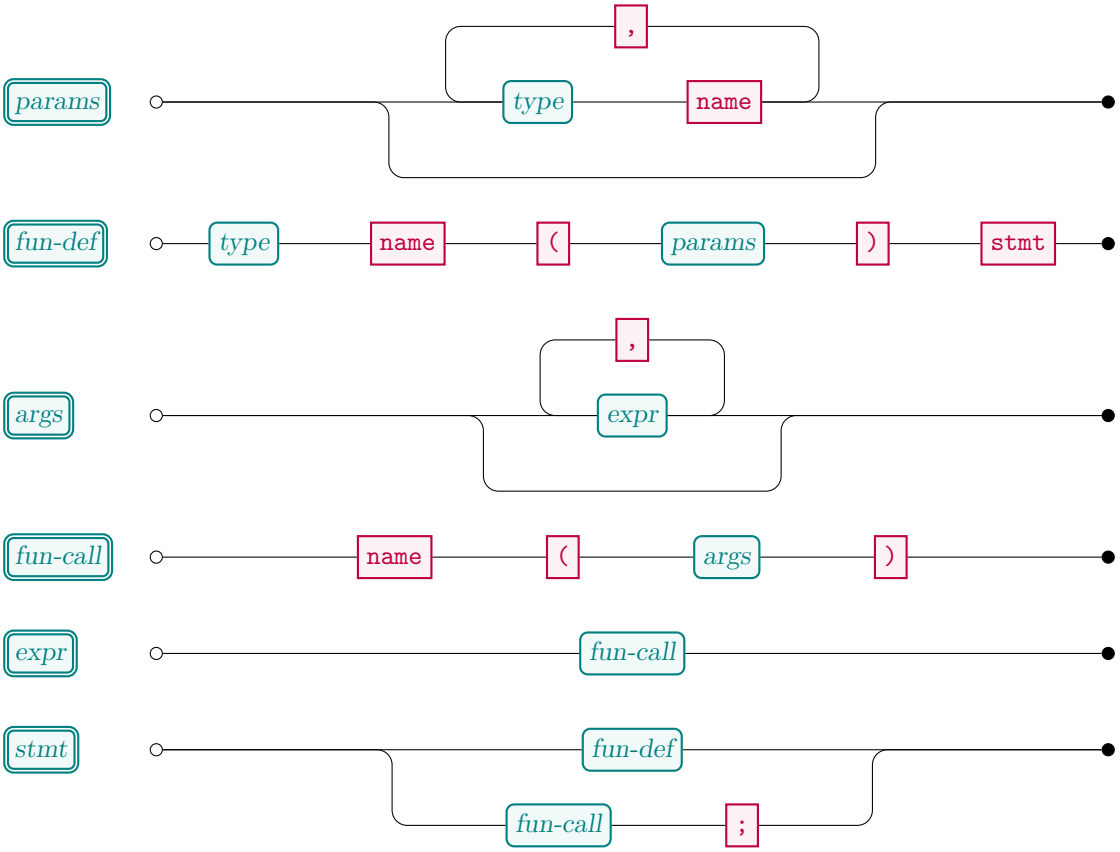
$$area(w, h) = w \cdot h$$

8.2 C#

8.2.1 Anatomy of a Function

A function is a parameterized piece of code. We can describe that function from the outside through its three interfaces, namely: The input interface, the output interface and the side-effect interface. On the input side, we have typed parameters. When invoked (or as we often say: called), the parameters available to the function body are bound to concrete values from the calling side. We can refer to these concrete values as arguments.

We can describe the interface of a function from three sides:



Syntax 8.1: Functions.

1. ***Input Interface:*** The input interface consists of the sequence of types of the function parameters.
2. ***Output Interface:*** The output interface consists of the function return type.
3. ***Side-Effect Interface:*** The side-effect interface is an umbrella term that covers all other interactions. This includes modifications to state external to the function and printouts to the screen.

8.2.2 Returning Values

The output interface of a function is explicitly described through the return type. In the subset of C# that we cover in this book, only one value can be returned. That value can be complex though. Do note that this is an artificial restriction that we add to keep things (relatively) simple.

If we want the function to evaluate to an `int` then the return type should be `int`. We can have any valid type as return type, even the complex types that we define ourselves. A call to a function can be placed anywhere in the code where a value of the functions return type can be

placed. The return type is thus defining for where the function can be called. For that reason, we often refer to a function that return an `int` as an `int` function. And that convention is, of course, not limited to the `int` type.

Unlike mathematical functions, not all C# functions evaluate to a value. Perhaps they simply print out a datastructure in a readable way. Instead of having such functions return some value that doesn't carry any meaning (e.g., a zero) have have the return type follow from that value, C# has a `void` type. If a function has a return type of `void` then it doesn't evaluate to a value and is thus not allowed to return a value.

A special return statement is used to explicitly state which value to evaluate to. The valid uses of return statements are governed by the following rules:

1. Every path through the function must end in a return statement (unless it is a `void` function).
2. A `void` function has an implicit return statement at the end.
3. No path through a function is allowed to have any statement after a return statement. How would such statement be reached? It is a clear indicator of a programming mistake.
4. All return values must match the return type of the surrounding function. If it is a `void` function, then the return statement does not take an expression.

Any function call can be used as a statement. When used as a statement, any return value will be discarded. For a function call to be used as an expression, it naturally must produce a value and that value must be of a type that is compatible with the use of that expression (i.e., its parent node in the parse tree). A `void` function does not return a value and a call to it can thus never be used as an expression.

8.2.3 Side-Effects

The primary interface of a function consists of its inputs and outputs. Functions can, however, also have *side-effects*. Side-effect is an umbrella term that covers every other effect. That is, how it affects the part of the world that is external to it. Examples of this include:

- Modifying data that is not local to the function. This would be through a reference.
- Printing to the screen.

Functions that neither have or rely on side-effects are said to be pure. Such functions are stateless and deterministic (in the sense that they will always produce the same output given the same input). This means that the function call can be substituted with the value it evaluates to. This quality is called referential transparency.

8.2.4 Caller and Callee Roles

We are always in some function, even in our hello world example from section 4.1.1. The entirety of this example is a single statement. The compiler will essentially wrap that code in a dummy function. So, whenever a function is called, there are two roles: The caller and the callee. If the function `a` calls the function `b`, then we refer to `a` as the caller and `b` as the callee.

8.2.5 Dispatch

In order to call a function, a concrete implementation must first be identified. This is done for us by the compiler. A dispatch mechanism is responsible for handing over the thread of execution to the called function.

In C#, this lookup is done based on the combination of the function name and the ordered sequence of parameter types. As a consequence of this, we are allowed to have two functions of the same name as long as their parameter type lists differ. That is, their signatures must be unique. If we have two identical signatures, then C#wouldn't be able to differentiate between them and thus be unable to uniquely dispatch in a consistent manner. Because of this, such code will result in a compile-time error.

8.2.6 The Stack

As the body of a function is an ordinary statement, functions can call functions. This leads to one challenge in particular: As the code of a function is compiled to machine code, variables are mapped to registers. That happens if a function calls itself? Are those variables then shares across the two function invocations? While that could be useful in some situations, it would certainly be confusing in most. In fact, most programming language designers (those of C# included) have decided that it is too confusing. Their solution is to include a program-wide datastructure called "the stack" that can store the registers of the calling function so that the called function is free to do with them as it pleases. The reality is a bit more complicated, but at this point it doesn't matter. As a side-effect, this greatly simplifies the compilation process as compilation can be done on a per-function basis.

Think of a stack as a stack of papers. It is a form of data structure that has two operations: You can add a value to the top of it (through a push operation) and you can remove a value from the top of it (through a pop operation). In reality, this means that the stack operates as a last-in-first-out (or LIFO) mechanism. Stacks are generic datastructures that are frequently used in programming. When we refer to "the stack" then it is the stack that keeps track of the variables that are local to calling function invocations that we mean.

Instead of placing individual values on the stack, we place stack frames. These are structures of data that match what needs to be saved. Everything that needs to be reestablished on return from the function call can then be popped in a single operation.

8.2.7 Guards

Consider the following problem: You are hosting a childrens birthday party, and you have baked a nice round cake. You want all the participants to receive a slice of equal size. How many degreees should each slice be?

What would this look like in code? We have to deliver an angle, and that angle depends on a number of participants. Note that this only makes sense for a positive number of participants, so we want to make sure to treat cases of negative and zero counts in a special way. We don't have a type that only supports positive integers. The closest we can get is to use a non-negative integer type, and that would still leave us with a special case. But all things considered, this is pretty simple. We can try out the functionality with a few different inputs:

```
for (int mouth_count=-4 ; mouth_count<5 ; mouth_count++) {
  double angle;
```

```
  if (mouth_count>0) {
    angle = 360/mouth_count;
  } else {
    angle = -1;
  }

  if (angle!=-1) {
    Console.WriteLine("Slices of "+angle+" degrees will feed "+mouth_count+" mouths");
  }
}
```

When executed, this program prints:

```
$ dotnet run
Slices of 360 degrees will feed 1 mouths
Slices of 180 degrees will feed 2 mouths
Slices of 120 degrees will feed 3 mouths
Slices of 90 degrees will feed 4 mouths
```

The code works, but the logic of calculating the angles is mixed in witht the logic for cycling throught the different inputs. It is hard to tell where one starts and the other ends. Understanding one of these two functionalities requires an overview of both functionalities. In a program of this size, that is not a problem. But that changes once we get to play with larger codebases. To deal with this, we need to find highly coherent parts of the code that can be parameterized as functions and thus be abstracted away as a function call. In this case, the natural abstraction is a function that, given a number of participants, calculates an angle. Then that function can be called for each of the numbers of participants. This allows us to reason about each of the two parts without understanding the intricacies of the other. It looks like this:

```
double slice_angle (int mouth_count) {
  if (mouth_count>0) {
    return 360/mouth_count;
  } else {
    return -1;
  }
}

for (int mouth_count=-4 ; mouth_count<5 ; mouth_count++) {
  double angle = slice_angle(mouth_count);
  if (angle!=-1) {
    Console.WriteLine("Slices of "+angle+" degrees will feed "+mouth_count+" mouths");
  }
}
```

```
double slice_angle (int mouth_count) {
  // guard: count must be positive
  if (mouth_count<=0)
    return -1;
```

```
  return 360/mouth_count;
}

for (int mouth_count=-4 ; mouth_count<5 ; mouth_count++) {
  double angle = slice_angle(mouth_count);
  if (angle!=-1) {
    Console.WriteLine("Slices of "+angle+" degrees will feed "+mouth_count+" mouths");
  }
}
```

8.3 Python

8.4 Elixir

Elixir inherits the BEAM runtime from Erlang. It it, function dispatch is performed by lookup against a table at runtime using function name and parameter count. This indirection can be used to redefine the implementation of functions at runtime, and that ability is used to issue hot updates. These are code updates to a running system. So, function call speed is traded for maintainability. This is especially important for long-running deployments.

8.5 Exercises

Exercise 8.1: Use

What are functions used for?

Exercise 8.2: Lack of Return Value

How do you declare a function that does *not* return a value?

Exercise 8.3: Sūdoku Prettyprinter

Let the following data structure represent a filled out Sūdoku puzzle:

```
int[][] puzzle = {
  new int[] {7, 3, 6, 4, 5, 2, 9, 8, 1},
  new int[] {1, 9, 8, 6, 3, 7, 4, 5, 2},
  new int[] {4, 2, 5, 9, 8, 1, 3, 7, 6},
  new int[] {3, 6, 4, 5, 2, 8, 1, 9, 7},
  new int[] {9, 5, 2, 7, 1, 4, 6, 3, 8},
  new int[] {8, 1, 7, 3, 9, 6, 2, 4, 5},
  new int[] {2, 8, 9, 1, 7, 3, 5, 6, 4},
  new int[] {6, 7, 3, 2, 4, 5, 8, 1, 9},
  new int[] {5, 4, 1, 8, 6, 9, 7, 2, 3},
};
```

Write a program in which:

1. The above data structure is defined.
2. A function is defined that, as a parameter takes such a data structure, and prints it out on the screen.
   - Which function prototype (i.e., return type and signature) should this method have?
   - Should one print out a full row at a time or a full column at a time?
   - **Hint:** The `Console.Write` method does the same as the `Console.WriteLine` method but doesn't break the line.
3. Some code that calls this function.

Exercise 8.4: Sum

Write a function that takes two integers as parameters, adds them and returns the result of this operation. Write a program that demonstrates how to use this function.

Exercise 8.5: Own Square Root

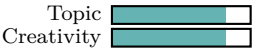
Write a program in which:

1. A function called `sqrt` is defined that calculates the square root of its input (of type `double`) to a precision of a specific (e.g., 7) number of decimals.
   - **Hint:** Solve this experimentall by starting with the most significant digit of he solution and iterate towards the least significant digit. For this, you pick a start and an end, and then iterate through these digits. These could be 1000000000 and 0.000000001. What should the value of this digit be? When you have found the right value for this digit, continue with the next. The right value is that which (when added to the previously determined digits and then squared) is $\leq$ the input. As you iterate through the digits, you will get closer and closer to the perfect answer.
2. Some code demonstrate how to call this function.

Exercise 8.6: Mathematical Equivalent

Which mathematical construction is the equivalent of a function, and where do the two differ?

Exercise 8.7: Discriminants and Roots



Write a program in which:

1. There is a function called `discriminant` that calculates a discriminant[1] basend on the parameters a, b and c.
2. There is a function called `roots` that takes the paramters a, b and c from a quadratic polynomial[2], and returns an array of roots.

[1] https://en.wikipedia.org/wiki/Discriminant
[2] https://en.wikipedia.org/wiki/Quadratic_function

- **Hint:** You can use `Math.Sqrt(9.0)` to calculate $\sqrt{9}$.

3. There is code that demonstrates how to call `roots` and prints out the result.

Exercise 8.8: Factorial function

Topic 
Creativity

Write a program in which:

1. A function calculates the factorial function[3] (e.g., $\text{fac}(4) = 4 \cdot 3 \cdot 2 \cdot 1$) without the use of a loop.
   - **Hint:** This can be done using *recursion*.
2. Some code calls this function with an argument of your choice and prints out the result.

Exercise 8.9: Properties of Circles

Topic 
Creativity

Write a program that calculates and prints out both the area ($\pi \cdot r^2$) and the circumference ($2 \cdot \pi \cdot r$) of three circles with radii of 1, 3 and 5.

Note: This is essentially a repetition of 7.8, but this time you have a larger (and more applicable) toolbox at your disposal.

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

Chapter 9

Exceptions

The normal flow of a program is that one statement is evaluated after another; some are conditionally evaluated, some are repeated, and functions are called. Parameters are transferred to the invoked function and work is being done one the stack to keep track of local caller and callee variables. Once the end of the function is reached, the thread of execution is returned to the caller that then carries on with its evaluation. Eventually, the last statement will complete and the process (or program) will exit.

Certain operations cannot be guaranteed to be successful. Functions implementing such operations will have some mechanism for reporting whether they were successful. Historically, this has often been done through the return value and/or global variables. The choice of global variables has the downside of them being shared across all thread of execution (and that make it hard to interpret). Threading is not a concept that we cover in this book, but it will likely become highly relevant soon after you finish it. Returning an error state is very common, but as most programming languages only support returning a single value, one would have to find a concrete value to represent an error. If you want to read a number of bytes from a file (that doesn't exist) and the function should return the number of bytes actually read, then any successful operation will return a non-negative number representing the number of bytes read. This means that we can use negative values to represent errors. This is cumbersome though, and we have to keep the error coding present in our minds. It also forces us to check the error after each function call (that could produce an error). The handling of the error is coupled to the function call.

Some languages have support for exceptions. An exception is a different mechanism for handling such abnormal flows. They decouple the discovery that something has gone wrong from the handling of that situation. They also provide a means for transporting the error code (and additional information) from the site of discovery to the site of handling. With exceptions, we no longer have to encode error indicators in return values or global variables.

9.1 Lifecycle of an Exception

In order to discuss what an exception is, we will take a look at its lifecycle. As shown in figure 9.1, the life of an exception goes through three phases, namely:

1. ***Construction:*** An exception is a form of data structure, and it has a type. We first construct an exception, and then we can do operations on it.

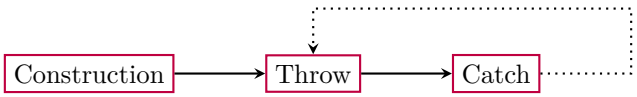


Figure 9.1: Lifecycle of an exception.

2. ***Throw:*** One of those operations is to *throw* it. This is what we do when we discover that something has gone wrong, and it will abruptly end the normal flow of execution. We will refer to this as the production site of the exception.

3. ***Catch:*** At an earlier point on our execution thread we have defined a handling site (aka catch site). This is kind-of a checkpoint. The latest defined handling site that matches the thrown exception will *catch* it. That means that execution will continue at this location.

Note: The handling site has access to the exception data structure and has the option of throwing it once again.

We can have multiple handling sites for the same exception. So, which one is used? To answer this, let's imagine that throughout the execution of our code we keep track of all the handling sites that we come across. Whenever an exception is thrown, it is caught by the handler (i.e., code) of the latest handling site. Execution immediately jumps to this point in the code. If it is not part of the current function invocation, stack frames will be popped off until the one matching the handling site is reached, and the environment stored here will be reestablished.

9.2 C#

To cover exceptions in C#, we need to introduce syntax 9.1. That is a lot of rules! However, when we group them by use, we can focus on a one or two of them at a time. So, let's do that.

9.2.1 Creating Exceptions

Like our data structure from chapter 7, exceptions are reference types. That means that we need to use `new` to allocate space for them on the heap. The syntax for creating a new exception is defined in the expr rule of syntax 9.1, and a practical application of that rule looks like this:

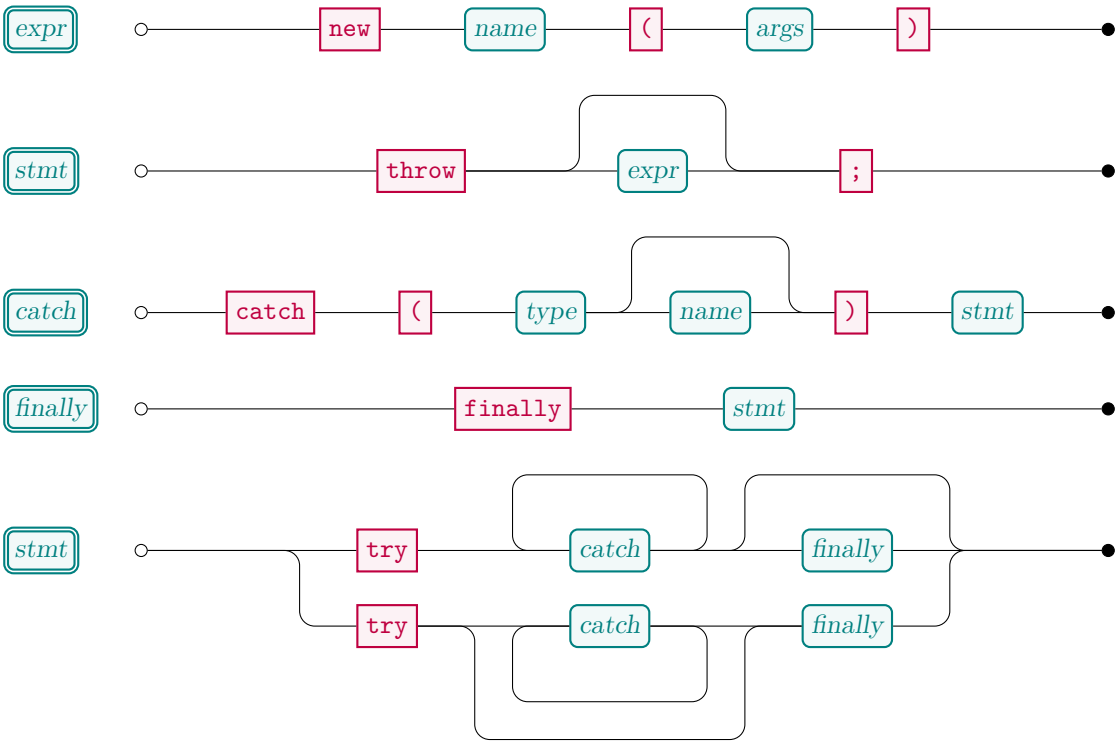
```
Exception e = new Exception("Oops!");
```

In and of itself, this program is useless: It creates an exception, but doesn't do anything with it. So, it might as well not have created it in the first place. But we will build on this code in the following sections.

9.2.2 Throwing Exceptions

Once we have an exception, we can throw it. For now, we use the expr track of the stmt rule from syntax 9.1. It looks like this:

```
Exception e = new Exception("Oops!");
throw e;
```



Syntax 9.1: Exceptions.

At the moment we have no code to catch the exception. So, when the exception occurs, the program crashes. This causes the exception to be printed:

```
$ dotnet run
Unhandled exception. System.Exception: Oops!
   at Program.<Main>$(String[] args) in Program.cs:line 2
```

A few things are going on here. Let's go through them one by one:

- The virtual machine will print all unhandled exceptions. An unhandled exception, is an exception that have not been caught by handling site and has reached the virtual machine.
- The exception has the type `System.Exception`, for which `Exception` is a a shorthand.
- Attached to the exception was a `"Oops!"` payload.
- The last line is the stack trace. It provides us with a path from the button of the stack to the production site. For each step we get the function (`Main`), the file (`Program.cs`) and the line in this file (line `2`). In a larger program, the stack trace will typically take up more lines; one for each stack frame.

9.2.3 Catching Exceptions

Any uncaught exception will make the program crash. At times, this may be what you want. But it is more common that we want to try out something, and then want to react if an exception is thrown. For casting exceptions we rely on the stmt rule of syntax 9.1. The following example is close to the simplest possible illustration of the mechanism:

```
try {
  throw new Exception("Oops!");
} catch (Exception e) {
  Console.WriteLine(e);
}
Console.WriteLine("But still alive ...");
```

Executing that code will result in the following. Please note that as the last printout makes it to the output, it must have been evaluated. So, catching an exception stops it from propagating.

```
$ dotnet run
System.Exception: Oops!
   at Program.<Main>$(String[] args) in Program.cs:line 2
But still alive ...
```

What happens is that the statements in the `try` block are evaluated in what we could call a sandbox. Here we are guaranteed that no thrown exceptions of the `Exception` type will spill out and affect the rest of the program. But, what happens if these statements call a function and the exception production site is in this function? Since the execution thread is still within the `try` block, the rules of the sandbox still applies. So, whenever an exception is thrown, it will be caught by the first matching `catch` block when moving down the stack. We can try it out with this code:

```
void fun () {
  throw new Exception("Oops!");
}

try {
  fun();
} catch (Exception e) {
  Console.WriteLine(e);
}
Console.WriteLine("But still alive ...");
```

When we execute this code we see mostly the same behavior. But since the production site is one stack frame further up, we naturally get one extra line in the stack trace:

```
$ dotnet run
System.Exception: Oops!
   at Program.<<Main>$>g__fun|0_0() in Program.cs:line 2
   at Program.<Main>$(String[] args) in Program.cs:line 6
But still alive ...
```

```
int dummy;
int[]? array;

for (int i=0 ; i<5 ; i++) {
  try {
    switch (i) {
    case 0:
      array = new int[0];
      dummy = array[1];
      break;
    case 1:
      array = null;
      dummy = array[0];
      break;
    case 2:
      dummy = 1/(i-i); // note the hack to have the compiler allow the /0
      break;
    case 3:
      throw new Exception("Oops!");
    }
    Console.WriteLine("No problem!");
  } catch (IndexOutOfRangeException) {
    Console.WriteLine("Caught IndexOutOfRangeException");
  } catch (NullReferenceException) {
    Console.WriteLine("Caught NullReferenceException");
  } catch (DivideByZeroException) {
    Console.WriteLine("Caught DivideByZeroException");
  } catch (Exception) {
    Console.WriteLine("Caught Exception");
  }
}
```

Figure 9.2: Demonstration of multiple exception handling sites for a single try block.

9.2.4 Multiple Exception Types

We have been focusing on the specific type of exception called `Exception`. But the truth is that there are more than one exception type. As they all behave the same, there is little need to dive deeper into them at this point. For now, we will simply – in figure 9.2 – show how we can have type-specific handling sites. In section 13.1, we will explore *why* all exceptions behave the same, and figure out that the order of the catch sites matter and why that is. When executed, this program prints:

```
$ dotnet run
Caught IndexOutOfRangeException
Caught NullReferenceException
Caught DivideByZeroException
Caught Exception
No problem!
```

The code gives a warning because the compiler can see that we are following a `null` reference in case 1. This warning has been removed for brevity.

9.2.5 Rethrowing

At times we may want to know whether an exception has occurred, but we don't want to have to deal with it. To do this, we first notice how when we caught an exception we got a variable of the same type as the expression we threw. That means that we can throw it again! So, we catch the exception, react to it, and then throw it again. When we catch an exception and throw it again, we say that we rethrow it. The practical code looks a bit different than what this suggests:

```
try {
  throw new Exception("Oops!");
} catch (Exception) {
  Console.WriteLine("Things are about to go badly ...");
  throw; // implicit reference to caught exception
}
```

Note how `throw` is used without an explicit reference to the exception that we want to throw? This follows the bypass of the exr rule of syntax 9.1. When throwing an exception, the stack trace field of that exception is updated to reflect the location in the code where the exception is thrown. We typically rethrow to have a peak at what is going on. Updating the stack trace would make it look like the exception originated from the part of the code that is peaking, and that would make it difficult to debug: While we would know what happened, we would not know *where* it happened. C# has this implicit variant of the `throw` mechanism to rethrow the last caught exception without updating its stack trace. Accordingly, we don't need to bind a name to that matched exception. In fact, as it would most likely be a typo, we would get a warning if we tried. Why else would we declare a variable without using it?

When we execute the program we see that the code in our handler is evaluated, and that the code still crashes as if we hadn't caught the exception:

```
$ dotnet run
Things are about to go badly ...
Unhandled exception. System.Exception: Oops!
   at Program.<Main>$(String[] args) in Program.cs:line 2
```

9.2.6 Finalization

Unfortunately, working with exceptions is more complicated than you likely realize: What happens when an exception is throws inside a `catch` block? The statement after the `try-catch` construct certainly isn't evaluated. So, if we need something to happen regardless of whether the initial exception is thrown then we cannot just put it after the `try-catch` construct. We *could* put it inside each of the two blocks, but that would cause us to duplicate the code. Even worse, it can be tricky to make sure that this code is evaluated inside the `catch` block. Luckily, there is another way: We can add a `finally` block according to the finally rule in syntax 9.1. We are guaranteed that the `finally` block is evaluated after the last statement of either the `try` block or the `catch` block (depending on whether an exception happens inside the `try` block).

The following code uses a loop to illustrate to iterate through the two different paths through a `try-catch` block and the `finally` block is evaluated in both cases:

```
for (int i=0 ; i<2 ; i++) {
  try {
    if (i==0) throw new Exception("Oops!");
    Console.WriteLine(i);
  } catch (Exception e) {
    Console.WriteLine(e);
  } finally {
    Console.WriteLine("Anyways ...");
  }
}
```

When that program is executed we get:

```
$ dotnet run
System.Exception: Oops!
   at Program.<Main>$(String[] args) in Program.cs:line 3
Anyways ...
1
Anyways ...
```

9.2.7 Side-Effects

It makes sense to think of a try construct as a means for us to try out something. But, if something goes wrong then it doesn't roll back: Any side-effects caused from within the `try` block will *spill out*. Accordingly, we may end up with side-effects of a partially evaluated `try` block, and that can cause inconsistencies in our data if we are not careful.

Let's see what that could look like. The following code calculates the average of some array of integers. It does so using a foreach loop over the input values. Inside the loop body, two things happen: a `count` variable is incremented and a `sum` is updated. To illustrate the problem, we have inserted some code in between these two actions that throw an exception for one of the inputs. So, for one of the iterations the count is updated but the sum is not. The full code looks like this:

```
int[] input = new int[]{1,2,4,5};

int sum = 0;
int count = 0;

foreach (int i in input) {
  count++;
  try {
    if (i==4) throw new Exception("Imagine that something went wrong");
    sum += i;
  } catch (Exception) {}
}

Console.WriteLine("average: "+(sum/count));
```

So, what happens when we execute this program?

```
$ dotnet run
average: 2
```

The program tells us that the average of 1, 2, 4 and 5 is 2. But this is wrong! It should be $(1+2+4+5)/4 = 3$, but because of the exception, the program calculates it as $(1+2+5)/4 = 2$. Now, this is a fairly artificially contrived scenario; no human would write code like this. But take one step back, and you should realize that while we do catch the exception (and nothing crashes) it still wrecks havoc in terms of the consistency of the variables involved in the loop. One could ask, is that better than crashing? Probably not. In this case it is fairly simple what is going on, but imagine a `try` block consisting of 20 lines and a handful of function calls. It likely won't be obvious where the exception come from, and thus which state modifications have taken effect.

9.3 C

C is an example of a language where errors are typically communicated through return values. Functions that do this dedicate a specific value (or more) from the universe of possible return values to indicate that an error has occurred. Accordingly, the full set of values indicated by the return type is not actually available. While that sounds really bad, but often guarantees can be given that some values will never be returned. For instance, no pointer – the C equivalent of a reference – with a value of zero (a so-called `null` pointer) is valid (it might indicate the absense of something though). So many functions that returns a pointer can safely use the zero value to indicate an error without restricting the practical use of that return value. When an error value is returned, convention dictates that the `errno` variable is set to a value that indicate the type of error. This variable is thread local (which in the framing of your current understanding mean global). Unline C# not further information is provided about the error.

As C has no garbage collector, memory management is a manual operation. That is, allocation and deallocation are explicit statements in the code. Memory allocation is an example of an operation that may fail (e.g., if we try to allocate more memory than what is available to us). In C, memory is allocated by calling the `malloc` function. If we want to make sure that the operation is successful then we need to check the return value. That looks like this:

```
#include <stdlib.h>

int main (int argc, char* argv[]) {
  char* data = malloc(1000);
  if (!data) {
    // handle error
  }

  // do something with data
}
```

The condition in that branch line may look quite weird to you. This stems from a combination of (i) pointers – like `data` – can be manipulated much like integers, (ii) booleans being integers and boolean operations thus operating on integers, and (iii) any non-zero integer representing a `true` value.

9.4 Go

The Go language has support for returning multiple values. This feature has played a major role in the design of both the standard library and most third party libraries. By convention, any function call that may fail will return two values. The last of these represents the error. If it is `nil` – to Go equivalent of `null` – then no error has occurred. Otherwise, an error has occurred, and it represents that error. The following code shows the idiomatic way of dealing with errors in Go:

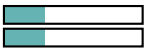
```
package main

import "os"

func main() {
  f, err := os.Open("input.txt")
  if err != nil {
      // handle error
  }
  defer f.Close()

  // do something with f
}
```

The `defer` line will make more sense when you get to section 21.5 that explains how to work with files. For now, just accept that (i) files that have been opened for access should be closed again, and (ii) this line makes sure that whenever the surrounding function returns then the file is closed (and the resources associated with it being open are released).

9.5 Exercises

EXERCISE 9.1: INDEXING

Topic
Creativity



Consider the following code:

```
int iterationer = 10;
int[] array = {1, 2, 3, 4, 5};

// increment
for (int i=0 ; i<iterationer ; i++) {
  array[i]++;
}

// print
for (int i=0 ; i<array.Length ; i++) {
  Console.WriteLine(array[i]);
}
```

Follow these steps:

1. Compile an execute the above code.
2. Which exception is thrown?
3. What causes this exception to be thrown?
4. Use a `try-catch` construct to *skip* the iteration that throw this exception.
5. Is this the correct solution to the problem, and why?

Exercise 9.2: Accounts

Consider the following code:

```
int[] accounts = {903, 716, 67};

int GetAccountNumber ()
{
  Console.WriteLine("Enter an account number: ");
  return Convert.ToInt32(Console.ReadLine());
}

void PrintAccountState (int accountId)
{
  Console.WriteLine("Account " + accountId + " contains " + accounts[accountId]);
}

while (true) {
  int accountId = GetAccountNumber();
  PrintAccountState(accountId);
}
```

The function `Convert.ToInt32` converts the `string` returned by `Console.ReadLine` to an `int`.

1. What happens if you give the program "3" as input?
2. Make a variant of the program that uses exception handling in the `while` loop to make the program more robust to this input.
3. What happens if you give the program "to" as input?
4. Make a variant of the program that uses exception handling to make the program more robust to this input. Where is the right place to add this exception handling?

Exercise 9.3: Average Grade

On some educations, the average grade is calculated solely from the passing grades a student has achieved. Lets assume that any grade below 2 (or 02) is a failing grade. In this exercise we explore how that could be implemented.

1. Create a new project.

2. Declare an `int[]` variable called `grades` and initialize this to hold the grades 4, 7, 02, 00, 10, 4, og 12.
3. Then declare a `GetGrade` function that takes an `int` as parameter and returns an `int`. Give the parameter the name `courseid`.
4. In the body of this function, extract using indexing first a value from the `courseid` position of `grades`. This value is stored in a local `int` variable called `grade`. If this is a passing grade, then return the grade. Otherwise, throw an exception.
5. Finally, write the main program code. It is split into three parts:
   a) ***Initialization*** Two variables of the type `int` are declared and named `count` and `sum`.
   b) ***Process*** Iterate through all indices (call them `courseid`) from `grades` using a `for` loop. Make sure it does not hardcode the number of courses[1]. For every `courseid`, you call `GetGrade`. Make sure that the the calls that *doesn't* result in an exception results in (i) `count` being incremented, and (ii) `sum` being incremented with the return value.
   c) ***Printout*** The average $\left(\frac{\texttt{sum}}{\texttt{count}}\right)$ is calculated and printed to the screen.
6. Verify that the code work as intended.

[1] Accordingly, the number is not constant, but rather derived at runtime from `grades`.

Chapter 10

Namespaces

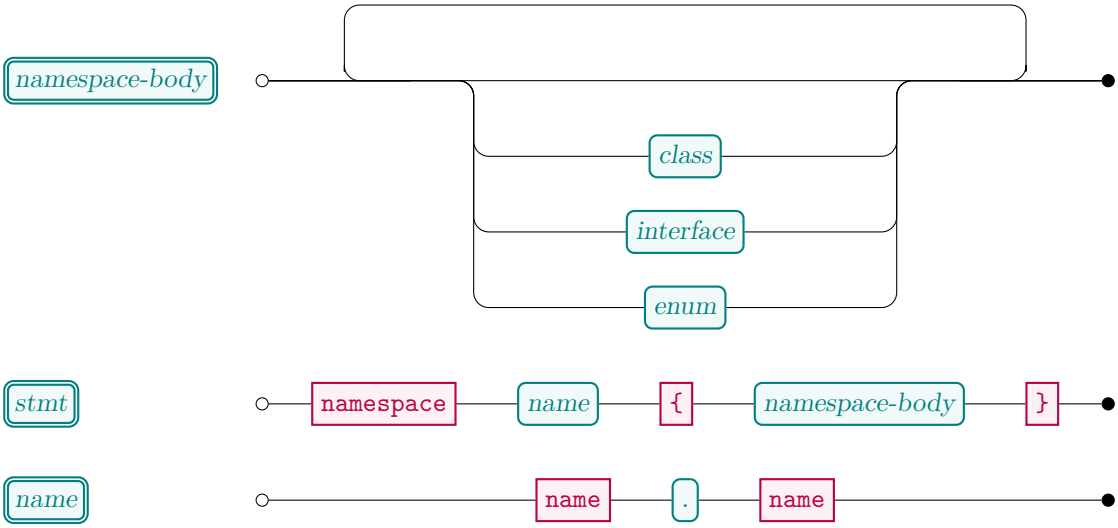
So far, most of the programs that you have written have been small. They are likely less than 20 line of code (LoC). If you solve all the exercises in this book then you will write programs of more than 100 LoC. If you complete a software education you will likely participate in projects in excess of 1000 LoC. In the industry, projects are often an order of magnitude or three larger. The Linux kernel was in 2025 approx 40M LoC. Most software projects grow over time as feature are added and improvements made.

As the codebase grows so does the number and severity of problems stemming from that size. One of the very first problems that you encounter is that of a name clash: You have found the right name for a variable or function, and now want to use the same name in a different context. Your reasoning may be as good as for the first instance, but the compiler won't allow you to redefine the name. Pure sadness! But, what would it even mean? If you had two functions by the same name, which implementation would be invoked when the name is called? Given that there two functions, perhaps the choice of implementation to invoke should depend on where in the code the call site is located? And what about variables; when are they distinct, and when are they the same? The situation appears to be a mess.

Let's introduce some order to this chaos. Up until now, all the names that we have defined have existed within a single space. This is called a namespace. Conflicts in names are not allowed within a namespace; names have to be unique. We can, however, have multiple namespaces, and a name is allowed to be defined in multiple namespaces. When we refer to a name, we either refer to the definition in the current namespace, or we explicitly refer to it through the name of a different namespace. This mechanism allows us to name distinct variables and functions without having to worry about whether they are used elsewhere. Sure, if it is used close by, there could be an issue but then you should ask yourself whether both names are expressive enough.

10.1 C#

C#'s primary mechanism for implementing namespaces is called … a namespace. Syntax 10.1 defines how we work with them. In the following section, we will go through these elements.



Syntax 10.1: Namespaces

10.1.1 Organizing Code

10.1.2 Definition

10.1.3 Lookup

10.1.4 Classes

Technically, the `class` construct that we used to define structured data in section 7.3 is also a form of namespace, but in C# terminology they are not. In section 12 onwards, we will dive deeper into what we can do with them.

Chapter 11

Outro

This is the end of the first part of the book. By now you should have a good grasp of imperative programming in C#. We have covered the primitive types, the basic data structures, the difference between value and reference types, branching, loops, functions and exceptions. With this, you can implement *any* computer algorithm. That's a big thing!

That still doesn't allow us to do a whole lot though. We need more ways of interfacing with the computer, and to make even remotely decent use of C# we also need booth the object-oriented features and the standard library.

In the next part of this book, we will look into object-oriented programming. It provides developers with a structured way of modeling high-level concepts. Working at a higher level of abstraction is convenient and typically makes developers to work more efficiently. It should be seen as an addition to what we have covered in part I; as an extension to your toolbox. C# was designed for object-oriented programming, and tradeoffs were made in favor of this programming style. If you choose not use its object-oriented features you are doing yourself a great disservice, and should probably go for a different language.

11.1 Exercises

Before looking into the following exercises, please note that there is a lot less hand-holding in the phrasing of these exercises, and they are an order of magnitude more complex than the previous exercises in this book. At this point, you do have the tools to solve them though. See them as a challenge, and don't worry too much if you end up giving up.

11.1.1 Sūdoku

Write a program that solves a puzzle.

This is – so far – the final exercise in our Sūdoku series. This series consists of:

- A puzzle representation in exercise 7.7.
- A pretty-printer in exercise 8.3.
- A checker in exercise 7.14.

11.1.2 Game of Life

Make an implementation of John Comway's Game of Life. You can find a description of this simulation on Wikipedia.

Note: A few hints:

- Illustrate each generation using characters (e.g., a space for no life and an asterisk for life).
- Pick a world size that fits on your screen.
- Insert a pause in between each generation.
- Initialize your world randomly

We have not quite covered everything needed to do follow these hints. The missing pieces, you can derive experimentally from this code:

```
Random r = new Random();

while (true) {
  Console.WriteLine(r.Next(2));
  Thread.Sleep(1000);
}
```

11.1.3 Eight Queens Puzzle

Solve the eight queens puzzle through code. To solve the problem, you need to place eight chess queens on a eight-by-eight chessboard in such a way that no two queens threaten each other. That is, each row, column and diagonal can have at most one queen. Write a program that finds one (or alternatively, *all*) solutions to this problem.

Hint: Try to iterate through all configurations of possible queen positions and check which of these are valid solutions.

Bonus: The eight queens puzzle can be generalized to an n queens puzzle. In this puzzle, you need to place n queens on an $n \times n$ chessboard. By solving this generalization, your solution will be capable of solving any specific variant (e.g, $n = 8$).

Part II

Object-Oriented Programming

Chapter 12

Objects

12.1 Syntactic Sugar

12.2 The Static Confusion

12.3 C#

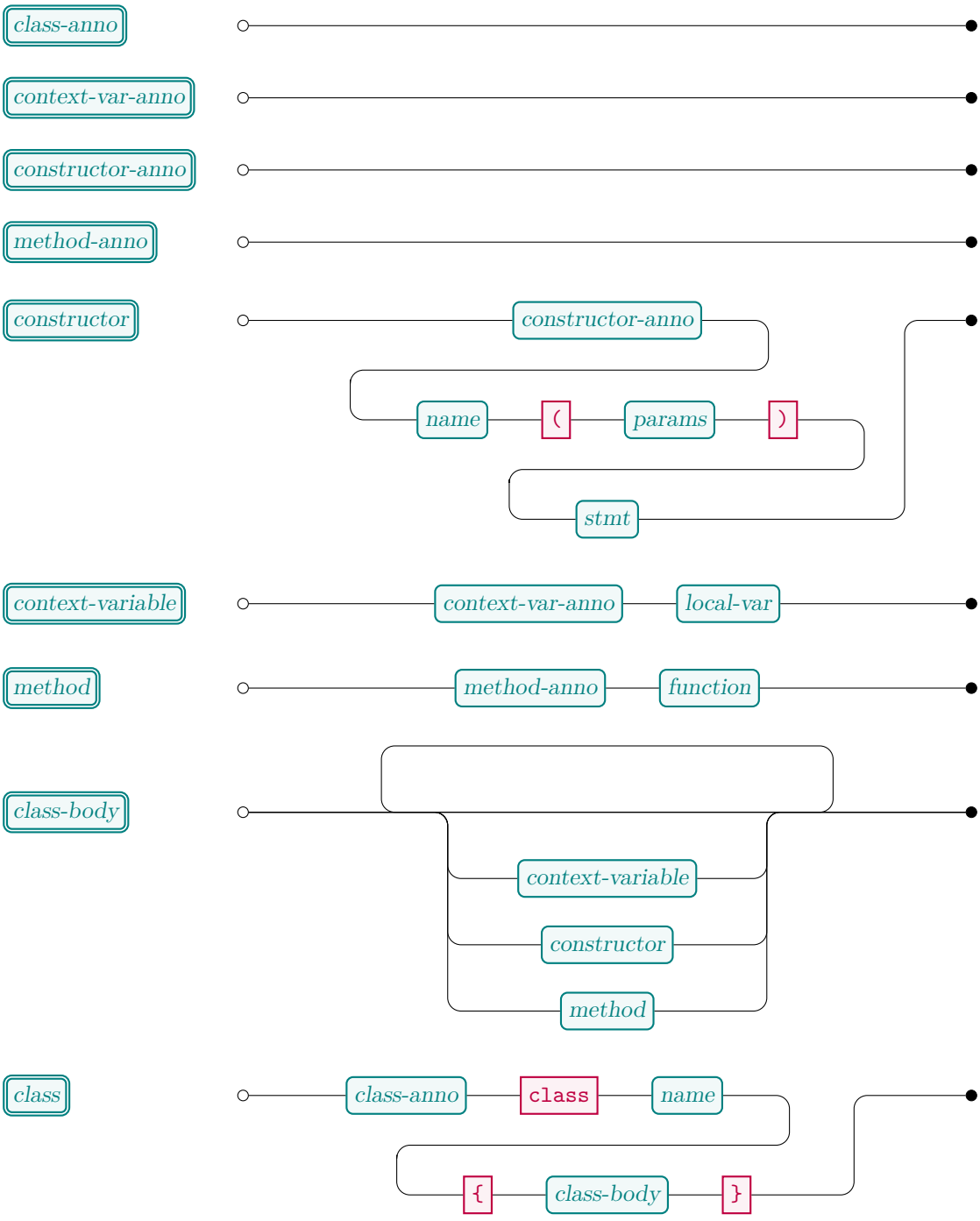
hello

12.4 Python

hello

```
Rectangle r = new Rectangle {CenterX=1 , CenterY=2, Width=3, Height=4};
Console.WriteLine(rectangle_area(r));
Console.WriteLine(rectangle_circumference(r));

double rectangle_area (Rectangle r) {
  return r.Width * r.Height;
}

double rectangle_circumference (Rectangle r) {
  return 2*r.Width + 2*r.Height;
}

void rectangle_move_to (Rectangle r, double x, double y) {
  r.CenterX = x;
  r.CenterY = y;
}

class Rectangle {
  public double CenterX, CenterY;
  public double Width, Height;
}
```

Figure 12.1: Implementation of a `Rectangle` using structs.



Syntax 12.1: Class definitions.

12.5 Elixir

hello

12.6 Case: Matrices

In this section we will briefly cover matrices. There are a number of domains where the ability for us to work with matrices is crucial. While they don't provide us with new functionality, they do represent a convenient abstraction, and they certainly help speed up graphics and graph calculations.

So what are they? They are rectangular arrays of numbers. Each of these numbers are called entries of the matrix. The size of a matrix is named after the size along each of the two dimensions: A 2×3 matrix has a height of 2 and a width of 3. If we name the matrix $\mathbf{A}$, then the entries are called:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

So, the dimensions of a matrix are 1-indexed. Madness!

A number of operations are defined on matrices. One can add two matrices of the same size, and one can subtract one matrix from another of the same size:

$$\begin{aligned} \mathbf{A} + \mathbf{B} &= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix} \\ \mathbf{A} - \mathbf{B} &= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} - \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} - b_{11} & a_{12} - b_{12} & \cdots & a_{1n} - b_{1n} \\ a_{21} - b_{21} & a_{22} - b_{22} & \cdots & a_{2n} - b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} - b_{m1} & a_{m2} - b_{m2} & \cdots & a_{mn} - b_{mn} \end{bmatrix} \end{aligned}$$

Matrices can take part in scalar multiplication and matrix multiplication. We will skip scalar multiplication in this book, and focus on matrix multiplication.

On top of this, there are a number of special matrix configurations that we often use. Sometimes we need a matrix of a particular size where all entries are zero or one. These are known as constant (value) matrices. At other times we need one that is square and has a diagonal of ones starting in the upper left corner. The rest of the entries in this matrix are zeroes. This matrix is called an identity matrix.

12.6.1 C#

So, how do we represent a matrix in C#? Well, the data fits nicely into a 2d array. For convenience, we would like to place these data inside of an object. This allows us to easily attach behavior in the form of methods and to organize related operations in the namespace of the class. Such relevant operations include methods for producing identity matrices, and constant-value matrices. This should not be too much of a stretch.

The stretch comes when we look into how to best implement the binary matrix operations (i.e., those that take a matrix as at least one of two parameters). Now that we have just learnt of instance methods, that would be an obvious vehicle for implementing this. For the add operation, we would have an instance method defined on a `Matrix` class that takes a `Matrix` as a parameter and return a new `Matrix`. But let's imagine that we want to add four matrices using such implementation:

```
Matrix result = matrix1.add(matrix2.add(matrix3.add(matrix4)));
```

While it certainly gets the job done, readability suffer.

As an alternative, we can overload the ordinary `+`, `-` and `*` operators. To do this, we need to provide C# with an implementation for each of the relevant function signatures. The mechanism that C# has for this is tied to the dispatch mechanism. A binary operator has two parameters and thus one or two types. We can provide the implementation as a static method on the class of either of these types. This static method is named "`operator +`" for the plus operation, and so on. Using the implementation from figure 12.2, the example of adding four matrices end up looking like this:

```
Matrix result = matrix1 + matrix2 + matrix3 + matrix4;
```

For the person who has *consume* that class (e.g., write code that makes use of it), this interface is a whole lot nicer!

12.7 Exercises

EXERCISE 12.1: CUSTOMERS

Topic
Creativity

Follow the following recipe for a program:

1. Create a new project with a class that contains a `Main`-method.
2. Create, in this project, a new class called `Customer`.
3. In the `Customer` class, add the following:
   - An attribute called `name` of type `string`.
   - An attribute called `id` of type `int`.
   - An attribute called `balance` of type `double`.
4. Add constructors to the `Customer` class:

```
public class Matrix
{
  public int       Width;  // number of columns
  public int       Height; // number of rows
  public double[,] Values;

  public Matrix (double[,] values) {
    Height = values.GetLength(0);
    Width  = values.GetLength(1);
    Values = values;
  }

  public static Matrix Constant (int width, int height, double val) {
    double[,] values = new double[height, width];
    for (int i=0 ; i<height ; i++)
      for (int j=0 ; j<width ; j++)
        values[i, j] = val;
    return new Matrix(values);
  }

  public static Matrix Zeroes (int width, int height) { return Constant(width, height, 0); }
  public static Matrix Ones   (int width, int height) { return Constant(width, height, 1); }

  public static Matrix Identity (int size) {
    double[,] values = new double[size, size];
    for (int i=0 ; i<size ; i++)
      values[i, i] = 1;
    return new Matrix(values);
  }

  public override string ToString () {
    string result = "";
    for (int i=0 ; i<Height ; i++) {
      for (int j=0 ; j<Width ; j++)
        result += (j==0 ? "" : ", ") + Values[i, j];
      result += "\n";
    }
    return result;
  }

  public static Matrix operator + (Matrix m1, Matrix m2) {
    if (m1.Width != m2.Width || m1.Height != m2.Height)
      throw new Exception("Parameters do not agree in size");

    double[,] result = new double[m1.Height, m1.Width];

    for (int i=0 ; i<m1.Height ; i++)
      for (int j=0 ; j<m1.Width ; j++)
        result[i, j] = m1.Values[i, j] + m2.Values[i, j];
    return new Matrix(result);
  }

  public static Matrix operator * (Matrix m1, Matrix m2) {
    if (m1.Width != m2.Height)
      throw new Exception("Incompatible parameter sizes");

    double[,] result = new double[m1.Height, m2.Width];

    for (int i=0 ; i<m1.Height ; i++)
      for (int j=0 ; j<m2.Width ; j++)
        for (int k=0 ; k<m1.Width ; k++ )
          result[i, j] += m1.Values[i,k] * m2.Values[k,j];
    return new Matrix(result);
  }
}
```

Figure 12.2: Implementation of `Matrix` class.

- One constructor takes a name and an id as parameters, and uses these values to initialize their corresponding attributes. The `balance` attribute is initialized to zero.
- One constructor takes a name, an id and a balance as parameters, and uses these values to initialize their corresponding attributes.

5. Add methods to the `Customer` class:
   - A method named `Deposit` with `void` return type that takes a parameter named `amount` of type `double`. This method must increment the value of `balance` with the value of `amount`.
   - A method named `Withdraw` with `void` return type that takes a parameter named `amount` of type `double`. This method must decrement the value of `balance` with the value of `amount` if and only if `balance` is greater than or equal to `amount`. In other words, the balance is not allowed to go negative.
   - A method named `GetBalance` that returns a `double`. When called, this method should return the value of `balance`.
6. In the `Main` method, declare a variable called `aCustomer` of type `Customer`.
7. Then, assign to `aCustomer` a (reference to a new) `Customer` object. This is done using an assignment statement where the right-hand side creates a new `Customer` object by calling a constructor for `Customer`. You decide which arguments to give it.
   - **Hint:** "`new`" is used to call the constructor of a class.
8. Deposit money to the `Customer` object by calling the `Deposit` method using an amount of your choice.
9. Withdraw money from the `Customer` object by calling the `Withdraw` method with an amount of your choice (that is less than the amount that you deposited).
10. Print out the value of the `balance` attribute of the `Customer` object by calling the `GetBalance` instance method and then calling `Console.WriteLine` with the return value of `GetBalance` as argument.
11. Verify – by observing the printout – that the code works as you would expect.

Exercise 12.2: Customer Database

Topic
Creativity

Extend the result from 12.1 with the following:

1. Create a new class called `CustomerDatabase`.
2. Add an attribute named `customers` of type `Customer[]` to `CustomerDatabase`.
3. Add a constructor to `CustomerDatabase` that doesn't take any parameters. It should initialize the `customers` attribute with an empty `Customer`-array with a length of 10.
4. Add the following methods to `CustomerDatabase`:
   - A method that takes a `Customer`-object as parameter, and adds it to the `customers` array at an empty position. What would be suitable names for the method, and its parameter?

- A method that takes an `int` parameter, and searches through the `customers` array to check if the `id` attribute of any element matches this parameter. If that is the case, the matching object is removed from the array. Otherwise, nothing is changed. What would be suitable names for this method, and its parameter.
  - **Hint:** The `customers` array holds references to `Customer` objects. The value `null` is used to indicate that a reference is invalid (aka, that the space is unused).
- A method that returns (all the contents of) the `customers` array. What would be a good name for this method?
- A method that prints out all the `Customer` objects in `customers`. What would be a good name for this method?

5. Add a `main` method to test the new functionality. How would you do this?

Chapter 13

Inheritance

Hello

13.1 Exceptions

13.1.1 C#

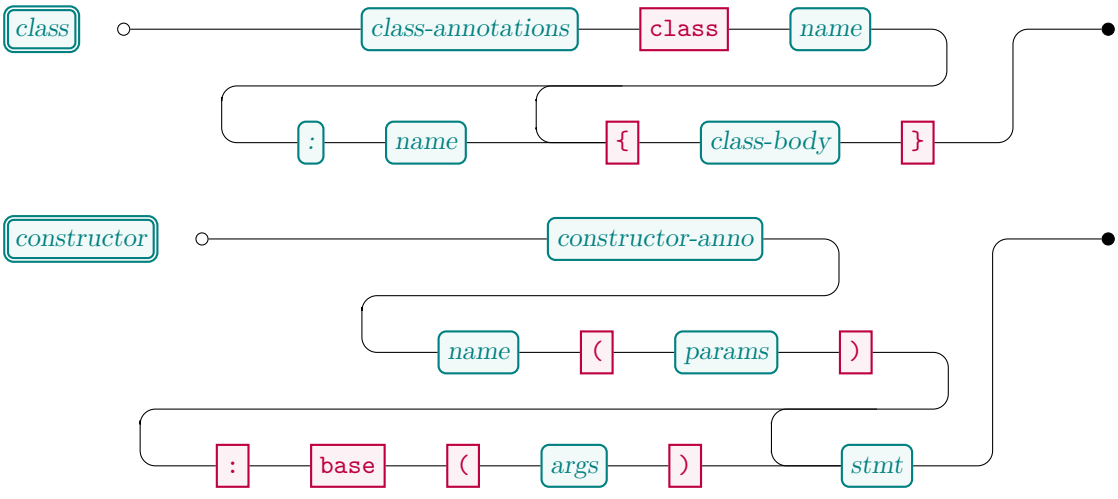
13.2 Exercises

Exercise 13.1: Inventory

Topic
Creativity

Subtask 1

The UML class diagram in figure 13.1 shows parts of an inventory system that makes use of inheritane. Implement the classes depicted in this figure.



Syntax 13.1: Class definition with inherence

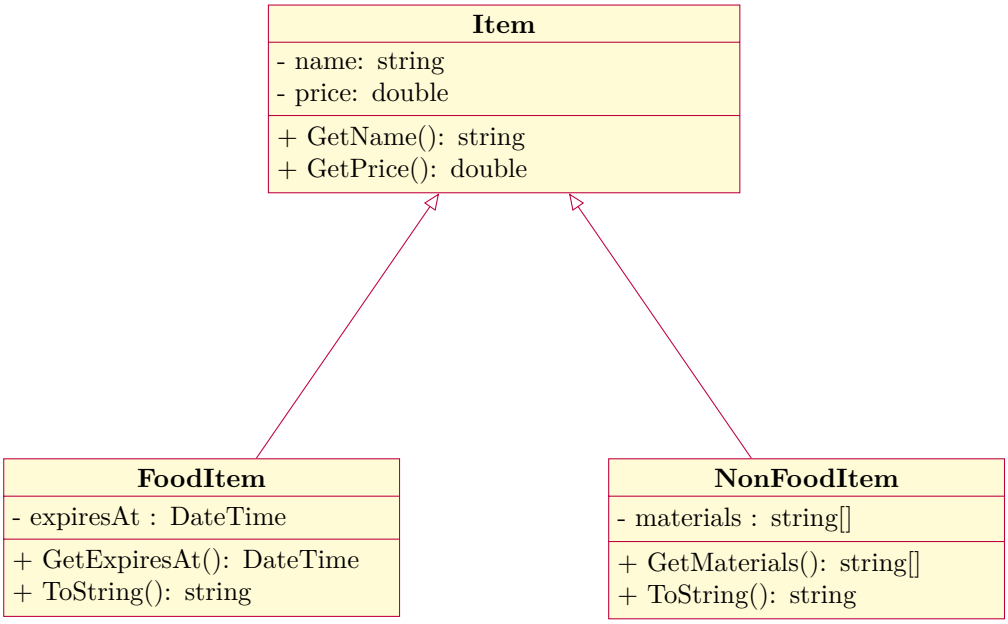


Figure 13.1: Arvehierarki for lagersystem.

The `ToString` methods in `FoodItem` and `NonFoodItem` must be annotated with the `override` keyword as they override the method from the `Object` class (not shown on the diagram).

The `ToString` method in `FoodItem` must return the name, price, and expiration date as a `string`.

The `ToString` method in `NonFoodItem` must return the name, price, and list of materials as a `string`.

Subtask 2

Make in your own `Main` method an array that can hold 10 `FoodItem` objects. Use a loop to add a `FoodItem` object to each element in this array.

Then add a loop in which you call the `ToString` method on each of the `FoodItem` objects in your array and print out the resulting string with `Console.WriteLine`.

Subtask 3

Repeat the exercise from the previous subtask, only this time for `NonFoodItem` objects.

Chapter 14

Naming

14.1 Visibility Modifiers

14.1.1 Problem

14.1.2 Solution

14.1.3 Accidental Features

14.1.4 C#

14.2 Exercises

Exercise 14.1: Kittens

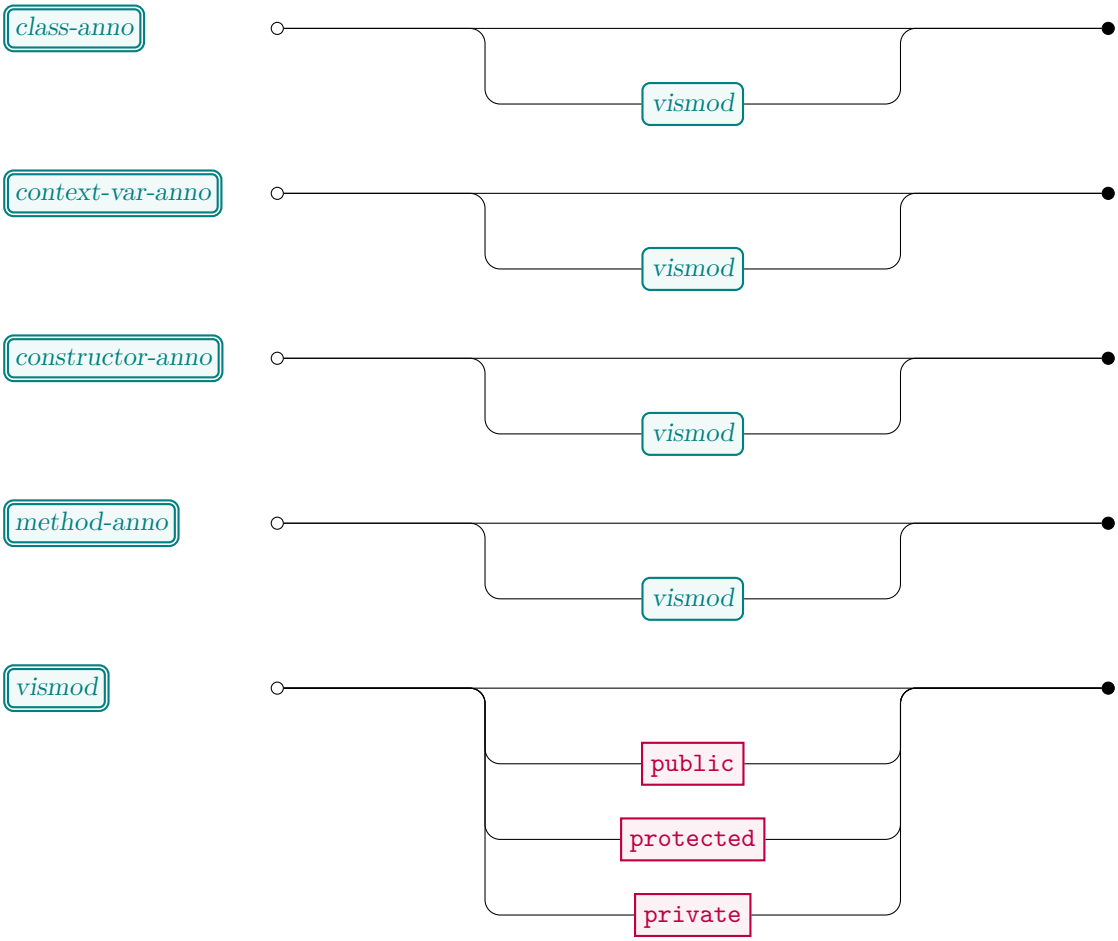
Write a program in which:

- There is a `Kitten` class.
- Objects of this type have a `cuteness` attribute of type `double` that is initialized through a constructor.
- A class variable (aka a *static* variable) named `count` of type `int` that is initialized to 0, and incremented every time a kitten is constructed.
- Add a `Main` method that – via a `for`-loop – constructs a number of `Kitten` objects with varying `cuteness`, and then finally prints out the value of `count`.
- Execute the program and verify that `count` has the expected value.

Exercise 14.2: Addition

Write a program following this recipe:

- Create a new project with a class called `Adder`.



Syntax 14.1: Visibility modifiers

- Create a static method named `Solve` that takes two `int` parameters and returns a `String`. This method must construct (and return) a `String` that expresses what the sum of the two parameters is. For the inputs 3 and 5, this could be "`3 + 5 = 8`".
- Create another static method by the same name that takes two `double` parameters and returns a `String`. This method, just like the first on, constructs its string in such a way that it expresses what the sum of the two parameters is. This could be "`3.14 + 5.12 = 8.26`".
- Create a `Main` method which:
  1. Calls `Solve` with two `int` values and prints out the result to the screen.
  2. Calls `Solve` with two `double` values and prints out the result to the screen.
- Compile and execute the program. Verify that the result is correct.
- What happens if we try to call `Solve` with an `int` and a `double`, and why does that happen?

Exercise 14.3: Operator

Topic
Creativity

Build and execute the following program:

```
class OperatorTest {
  public static int Operator (int a, int b, bool c) {
    return a + b;
  }

  public static int Operator (int a, int b, char c) {
    return a - b;
  }

  public static void Main (string[] args) {
    for (int a=0 ; a<5 ; a++) {
      for (int b=0 ; b<5 ; b++) {
        Console.WriteLine(a+" (?1?) "+b+" = "+Operator(a, b, true));
        Console.WriteLine(a+" (?2?) "+b+" = "+Operator(a, b, false));
        Console.WriteLine(a+" (?3?) "+b+" = "+Operator(a, b, 'a'));
        Console.WriteLine(a+" (?4?) "+b+" = "+Operator(a, b, 'b'));
        Console.WriteLine(a+" (?5?) "+b+" = "+Operator(a, b, '.'));
      }
    }
  }
}
```

Consider the output (i.e., what is printed to the screen), and explain what is going on. Specifically:

1. Which operators are evaluated on the individual lines? Or, in other words, what does `(?1?)`, `(?2?)`, `(?3?)`, `(?4?)` and `(?5?)` represent?
2. How do these relate to the 3rd argument to the `Operator` calls?
3. What is the important part of that 3rd parameter? Which rule is being exploited?
4. Is this a reasonable interface for `Operator`?

Exercise 14.4: Scope

Topic
Creativity

In the following program, where can (and should) each of the variables `i`, `d`, `tmp`, `sum` and `bonus` be declared?

```
class Scope {
  // location 0
  bonus = 42;
  // location 1

  static int Doubler (int value) {
```

```
    // location 2
    d = value * 2;
    // location 3
    return d;
    // location 4
  }

  // location 5

  public static void Main (string[] args) {
    // location 6
    sum = 0;
    // location 7
    for (/* location 8 */ i=0; /* location 9 */ i<100 ; /* location 10 */ i++) {
      // location 11
      tmp = Doubler(i);
      // location 12
      sum += tmp;
      // location 13
    }
    // location 14
    Console.WriteLine(sum + bonus);
    // location 15
  }

  // location 16
}
```

What happens if a variable is declared twice?

Exercise 14.5: Encapsulation

Topic
Creativity

Consider the following program:

```
class Circle {
  public double x, y;
  public double r;

  public Circle (double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.r = radius;
  }
}

class TestCircle {
  static void Main (string[] args) {
    Circle c = new Circle(1.24, 2.83, 12.7);
    Console.WriteLine("x=" + c.x + " y=" + c.y + " r=" + c.r);
```

```
    c.r *= 1.37;
    c.x += 0.65;
    Console.WriteLine("x=" + c.x + " y=" + c.y + " r=" + c.r);
  }
}
```

Follow these instructions:

1. Describe in english what the programs `Main` method does. Focus on the details. Verify by evaluation.
2. Make a copy of this program. You will not continue the work on the old version, but you need it for step 11.
3. Update the new copy so that the attribute `r` (that represents a radius) is replaced by a `d` attribute (that represents a diameter).
4. Write down the changes you have made to achieve this.
5. Make a copy of this program. You will not continue the work on the old version, but you need it for step 11.
6. Encapsulate all attributes. To encapsulate an attribute, you need to:
   - Declare the attribute `private`.
   - Add relevant getters and setters.
7. Make a copy of this program. You will not continue the work on the old version.
8. Create a new class by the name `Coordinate` that represent an (x, y) coordinate.
9. Update the new program to make use of the `Coordinate` class.
10. Write down the changes you have made to achieve this.
11. Steps 4 and 10 represents similar changes with different starting points. The difference is whether the attributes are encapsulated. Compare not you notes from these steps. What does encapsulation for for you when you modify the representation of `Circle`?

Chapter 15

Polymorphism

15.1 Exercises

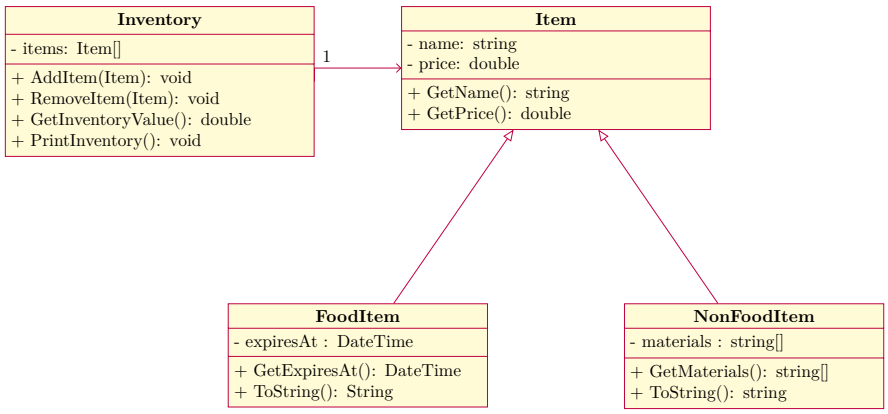
Exercise 15.1: Inventory

Topic
Creativity

This exercise builds on the exercise from section 13.1. You can find a reference solution in section 81. It is your choice whether to build on this, or your own solution.

Subtask 1

Extend the implementation to reflext this UML class diagram:



The `GetInventoryValue` must iterate through all `Item` objects in `items`, call `GetPrice` on each of the objects, sum up the resulting values and return this value.

The `PrintInventory` method must print out a textual representation of all the objects in the `items` array using `Console.WriteLine`. Use a loop of your choice to accomplish this.

`AddItem` and `RemoveItem` must add and remove objects to and from the `items` array.

Subtask 2

Move the `Main` method out into a `Test` class and update it in such a way that you add items of both `FoodItem` and `NonFoodItem` to an `Items` array in an instance of `Inventory`.

Call the `PrintInventory` and `GetInventoryValue` methods to validate their functionality.

Exercise 15.2: A Day at the Races

Topic
Creativity

In this exercise, you'll write a program that can simulate races between different types of racers. The program should be flexible, so that no matter what set of entities enter a race (for example a car and a horse), the overall implementation doesn't need to change. You'll use inheritance and polymorphism to achieve this.

Subtask 1

Create a class called `Racer`. It should contain the following:

- A `public` instance variable called `Name` of type `string`.
- A `protected` instance variable called `speed` of type `double`.
- A `protected` instance variable called `distance` of type `double`.
- A constructor which takes a `string` value and a `double` value as parameters and uses them to initialize `Name` and `speed`.
- A `virtual` method called `Race` which returns a `double`. The purpose of this method is to calculate the racer's distance over time while in a race. It should take a `double` value called `time` as parameter. The method should multiply the `time` parameter with the racer's `speed` and add the result to its `distance`. It should then return the distance.

Subtask 2

Create a class called `RaceCar` and make it inherit from `Racer`. In this new class, do the following:

- Add two more instance variables of type `double`, one called `fuel` and one called `fuelEfficiency`.
- Add a constructor which takes the same parameters as the `Racer` constructor plus an additional two `double` parameters. It should call the `base` constructor to initialize `Name` and `speed`, and use the two new `double` variables to initialize `fuel` and `fuelEfficiency`.
- Override the `Race` method. Add new code inside the method which:
  1. Subtracts `time` divided by `fuelEfficiency` from `fuel`.
  2. Checks if `fuel` is now less than or equal to zero, and sets speed to zero if it is.
  3. Finally adds `speed` multiplied by `time` to `distance` and returns `distance` (call the `base` method implementation to reuse existing code).

Create a class called `RaceHorse` and make it inherit from `Racer`. In this new class, do the following:

- Add one more instance variable called `stamina` of type `double`.
- Add a constructor which takes the same parameters as the `Racer` constructor plus an additional `double` parameter. It should call the `base` constructor to initialize `Name` and `speed`, and use the new `double` variable to initialize `stamina`.
- Override the `Race` method. Add new code inside the method which:

1. Subtracts `time` divided by `stamina` from `speed`.
2. Sets `speed` to a minimum value (for instance two) if it is now less than that value.
3. Finally adds `speed` multiplied by `time` to `distance` and returns `distance`.

Extension: Create one or more other classes that inherit from `Racer`. Override the `Race` method in each of these classes and implement your own logic that determines how `distance` over `time` should be calculated.

Subtask 3

Before proceeding to the final step of simulating the races themselves, you should test your current implementation to see if everything works as expected. This can be done by instantiating at least one object of each class that inherits from `Racer` and calling every implementation of the `Race` method. You can then write the results to the terminal to verify that they come out as expected.

Subtask 4

To simulate races, create a method somewhere in your program which:

1. Takes the following parameters: `racers` of type `Racer[]`, `raceDistance` of type `double`, and `playbackSpeed` of type `double`.
2. Declares a local `bool` variable called `finished` and sets it to `false`.
3. Contains a loop which repeats while `finished` is `false`. In each iteration of this loop, iterate through the `racers` and for every `Racer` object:
   a) Calculate that `racer`'s current distance raced by calling `Race(playbackSpeed)` and saving the result in a local variable called `distance`.
   b) For every integer number from zero until (but not including) `distance`, write "`-`" to the screen without linebreaks.
   c) Write the name of the racer to the screen and then break the line.
   d) Check if distance is greater than or equal to `raceDistance`, and set `finished` to `true` if it is.
4. After iterating through all the `racers`, write this line of code at the end of your loop: `Thread.Sleep(1000);`. It will pause the execution of your program for 1000 ms.

Once you've finished making the method, it's time to have some races. Do the following in your program's `Main` method:

- Instantiate at least one object of each class that inherits from `Racer`.
- Create an array of type `Racer[]`, and add all of your newly instantiated `Racer` objects to it.
- Call the method you just made and pass the array of racers as argument, along with your choice of `raceDistance` and `playbackSpeed`.
- Repeat this process a number of times and use different parameters to see how the results differ.

Extension: Adjust the method for simulating races to your liking to make it more interesting to look at. You could, for instance, make the racers wait at the beginning of the race before they start moving, show visually where the goal is, or announce the winner of the race at the end.

Subtask 5

But how do you determine – in code – who won the race? What happens when two racers cross the finish line in the same simulation step?

Chapter 16

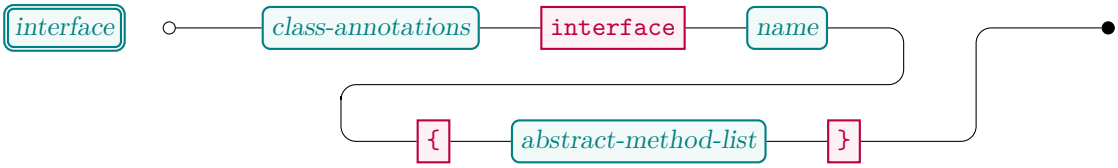
Abstract Methods

Hello

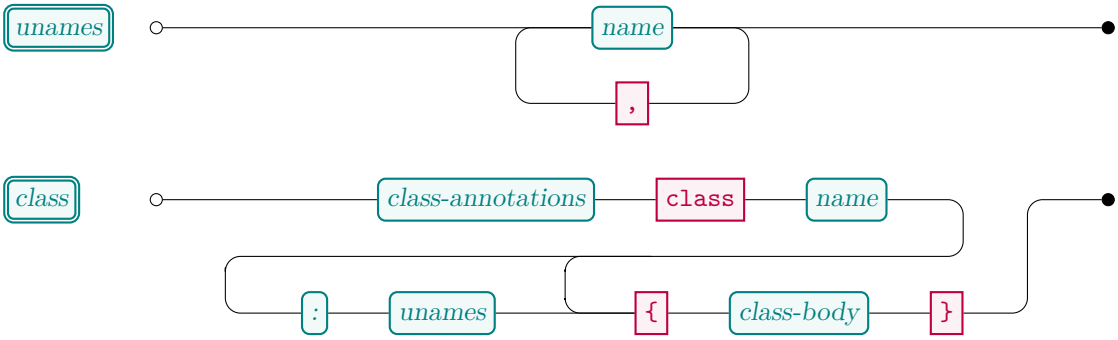
16.1 Interfaces

16.1.1 C#

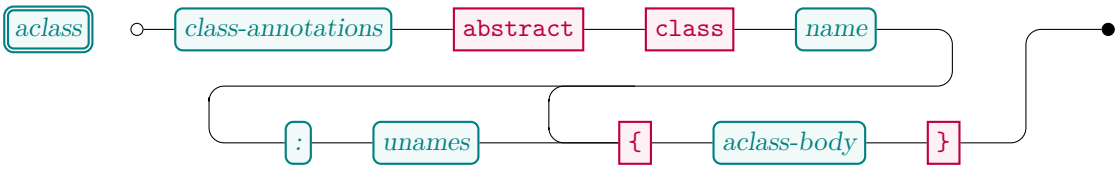
For instance, many things can be locked and unlocked. These things are *lockable*, and that is a quality that we might want to associate with both buildings and vehicles. We can use an interface to group two methods that define this behavior; one that locks, and one that unlocks.



Syntax 16.1: Interface definitions.



Syntax 16.2: Class update to allow for interface implementation.



Syntax 16.3: Abstract class definitions.

16.1.1.1 Naming Convention

It is convention, in C#, to prefix the name of an interface with a capital `I`. The interface defining the *lockable* behavior should thus be called `ILockable`. This convention is not enforced.

16.2 Abstract Classes

16.2.1 C#

16.3 Exercises

EXERCISE 16.1: INVENTORY

Topic
Creativity

This exercise builds on the exercise from section 15.1. You can find a reference solution in section 87. It is your choice whether to build on this, or your own solution.

Subtask 1

To ensure that items are always instantiated as either `FoodItem` or `NonFoodItem` objects, you must declare the `Item` class abstract. What happens if you then try to instantiate the `Item` class directly, and why is this the case?

Subtask 2

Create an interface called `IExpireable` that contains a method with the following prototype:

```
bool IsExpired();
```

Subtask 3

Implement this interface in `Item`.

Subtask 4

As `FoodItem` and `NonFoodItem` inherits from `Item`, they also inherit the `IsExpired` method. *Override* this method in the `FoodItem` class and make its implementation rely on the `expiresAt` attribute for defining when the item is too old.

Hint: The `DateTime` type in the .NET framework can be used to represent dates (and a time of day).

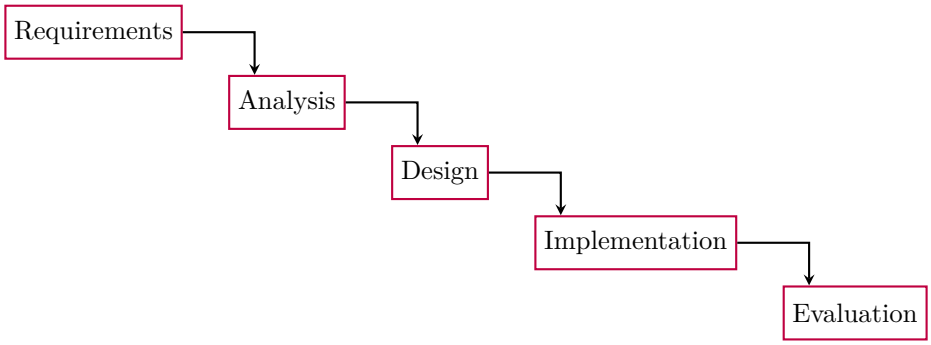


Figure 17.1: Classical illustration of the waterfall model.

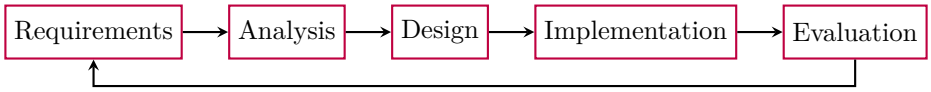


Figure 17.2: Illustration of a typical iterative model.

Chapter 17

Software Development

> *"Going to disk is 25 million times slower than hitting a general purpose register. Design accordingly."*
>
> — *Robert Love*[7]

17.4.3 Difference

17.4.4 Quality

17.4.5 Organization

17.4.6 Critique

17.5 Local Maximum Trap

17.6 Exercises

EXERCISE 17.1: ENROLLMENT SYSTEM

Consider the following program specification:

The enrollment system contains a list of students and courses. It can display the list of students which are enrolled in a particular course. It is possible to enroll students to a course and remove them from a course.

From it, the following requirements have been derived:

- ***Functional requirements:***

ID	*Name*	*Description*
F01	Display a list of students	It must be possible to display a list of students
F02	Display a list of courses	It must be possible to display a list of courses
F03	Add and remove association	It must be possible to associate students to a course, and to remove such associations

- ***Non-functional requirements:***

ID	*Name*	*Description*
NF01	Capacity	The system must be able to handle 10 courses
NF02	Interoperability	The system must be able to integrate with itslearning

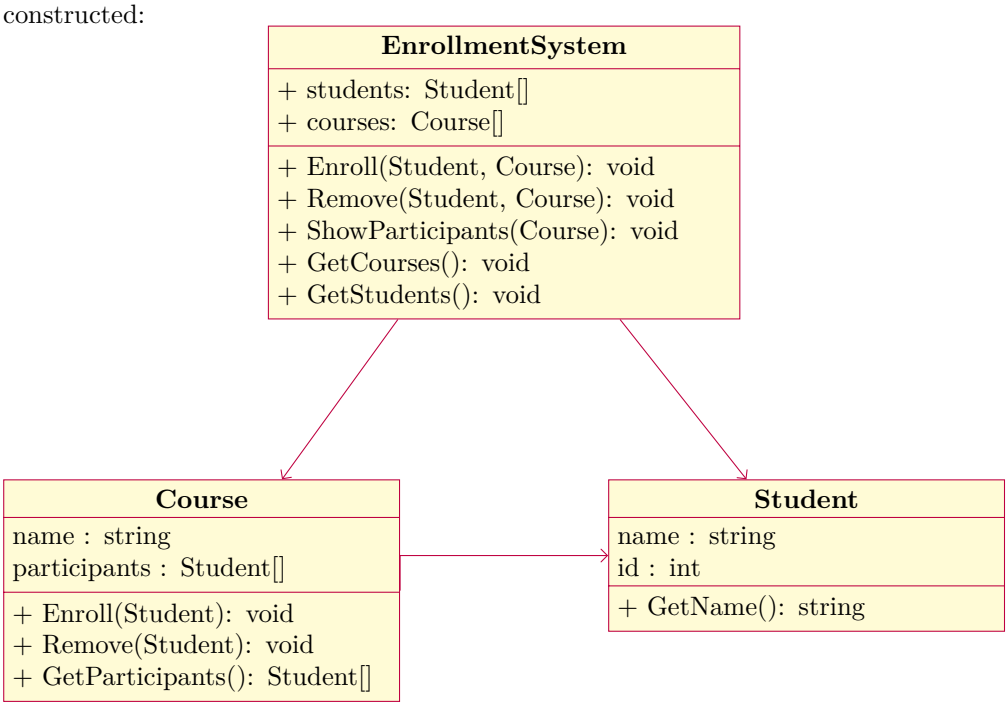
A noun-verb analysis of the program specification has resulted in:

The `enrollment` system `contains` a `list` of `students` and `courses`. `It` can `display` the `list` of `students` which `are` enrolled in a particular `course`. It is possible to `enroll` `students` to a `course` and `remove` `them` from a `course`.

This has resulted in the following CRC card:

EnrollmentSystem	
Responsibilities	*Collaborators*
• Contains list of `Course`-objects • Contains list of `Student`-objects • Can enroll `Student`-objects in `Course`-objects • Can remove `Student`-objects from `Course`-objects • Can display `Student`-objects enrolled in any given `Course`-object	• Student • Course

Based on this, and similar for the remainding classes – the following class diagram has been constructed:



This has resulted in the following implementation:

```
public class Student {
  public string name;
  public int id;

  public Student (string nameValue, int idValue) {
    name = nameValue;
    id = idValue;
  }

  public string GetName () {
```

```
    return name;
  }
}

public class Course {
  string    name;
  Student[] participants;

  public Course (string nameValue) {
    name = nameValue;
    participants = new Student[10]; // NOTE: This constant is BAD!
  }

  public void Enroll (Student student) {
    for (int i=0 ; i<participants.Length ; i++) {
      if (participants[i] == null) {
        participants[i] = student;
        return;
      }
    }
  }

  public void Remove (Student student) {
    for (int i=0 ; i<participants.Length ; i++) {
      if (participants[i] == student) {
        participants[i] = null;
      }
    }
  }

  public Student[] GetParticipants () {
    // count number of entries
    int count = 0;
    foreach (Student student in participants) {
      if (student != null) {
        count++;
      }
    }

    // make a copy
    Student[] result = new Student[count];
    int index = 0;
    foreach (Student student in participants) {
      if (student != null) {
        result[index++] = student;
      }
    }

    return result;
  }
```

```
}

public class EnrollmentSystem
{
  public Student[] students;
  public Course[] courses;

  public void Enroll(Student student, Course course) {
    course.Enroll(student);
  }

  public void Remove(Student student, Course course) {
    course.Remove(student);
  }

  public void ShowParticipants(Course course) {
    foreach (Student student in course.GetParticipants()) {
      Console.WriteLine(student.GetName());
    }
  }

  public void GetCourses() {
    Console.WriteLine("void for a getter?");
  }

  public void GetStudents() {
    Console.WriteLine("void for a getter?");
  }
}
```

With all of that as a starting point:

1. The program specification has the significant deficiency that it doesn't describe any possibility to add or remove courses or students. Update the specification so that this functionality is included in the specification.
2. Update the list of requirements to reflect these changes.
3. Redo the noun/verb analysis on the reviced program specification, and update the list of methods and/or classes.
4. Update the CRC card so that it matches the new noun-verb analysis and requiremetns. Add new cards if necessary.
5. Update the UML cladd diagram so that it reflects your updated CRC card.
6. Update the implementation of the program so that it reflects your new UML class diagram.

Part III

Practicalities

Chapter 18

Parameterized Types

In chapter 13, we covered the concept of inheritance. We presented it as a way of avoiding duplicate code while allowing for polymorphic access to variants of behavior implementation. It worked by extending some basic definition in order to specialize the type definition. So, a shared starting point with individual extensions. But what if the basic definition is the part that we want variations over? Then we use parameterized types (aka generics).

18.1 Problem

A minimal example of this is a "pair". That is, two of something. The concept could relate to integers, socks, dice or something different altogether. In object-oriented programming, it seems natural to use a class to wrap two elements of the relevant type. Figure 18.1 illustrates how such a pair class could be defined for integers and for points.

It doesn't quite feel right though. The concept of the pair is intertwined with the type of the elements. The `ToString` method and the body of the constructor are exactly the same in the two classes. If we look past the types, then so are the instance variables and constructor parameters. These are all things that are linked to the concept of a pair; not the type of the elements. The only reason for the classes (and thus their constructors) to be named differently is that they

```
class IntPair {
  public int a;
  public int b;

  public IntPair (int a, int b) {
    this.a = a;
    this.b = b;
  }

  public override string ToString () {
    return "("+a+","+b+")";
  }
}
```

```
class PointPair {
  public Point a;
  public Point b;

  public PointPair (Point a, Point b) {
    this.a = a;
    this.b = b;
  }

  public override string ToString () {
    return "("+a+","+b+")";
  }
}
```

Figure 18.1: Two similar classes with varying base definition.

can't have the same name.

The consequences of this extend beyond mere practicalities. For starters it does not scale with respect to element types: We need one definition of a pair for each element type we want to support. So, if I want to have 17 different elements types in pairs, I need 17 different pair classes. That is a lot of code that I have to manually keep up to date as my perception of a pair evolves. This evolution is a reference to how the code changes over time as features are added or bugs corrected. We say that this results in low maintainability.

But this still leaves us with a missed opportunity for polymorphism. It would be nice to be able to be able to define code that can can handle any pair. We could accomplish this by placing be pair behavior functionality in an interface that all pair types implement. Then that interface would represent the type that all pairs satisfy. But we would still have the scalability problem.

We could define the class as a `Object` pair. That would allow any type to satisfy the instance variables. But we would this would come with two negative consequences, namely:

1. ***Loss of type specificity:*** We would bypass the compilers mechanisms for ensuring type safety and shift that uncertainty to the runtime. Without the guarantees of the compiler, we would need to downcast our instance variables in order to access any instance variable or method that we might find interesting. If at any point we ended up constructing a pair of a type that does not satisfy the type of the downcast, then an exception will be thrown. If we don't write code for handling that, then our program will crash. We could make an agreement developer-to-developer that we will never do this, and that there is thus no need to handle any exception. But this is where human mistakes come into play.

2. ***Higher cost of use:*** In order to *be able to* throw an exception, the runtime performs a check before downcasting. And that takes a bit of time. Otherwise, an invalid downcast would result in either undefined program behavior, a program crash or one followed by the other. So, the downcast comes with a runtime penalty.

18.2 Solution

To solve this problem many programming languages have introduced a notion of a parameterized (or generic) type. A type (think class or interface) is parameterized when certain types withing its definition are replaced by named placeholders. The recipe can be written down as:

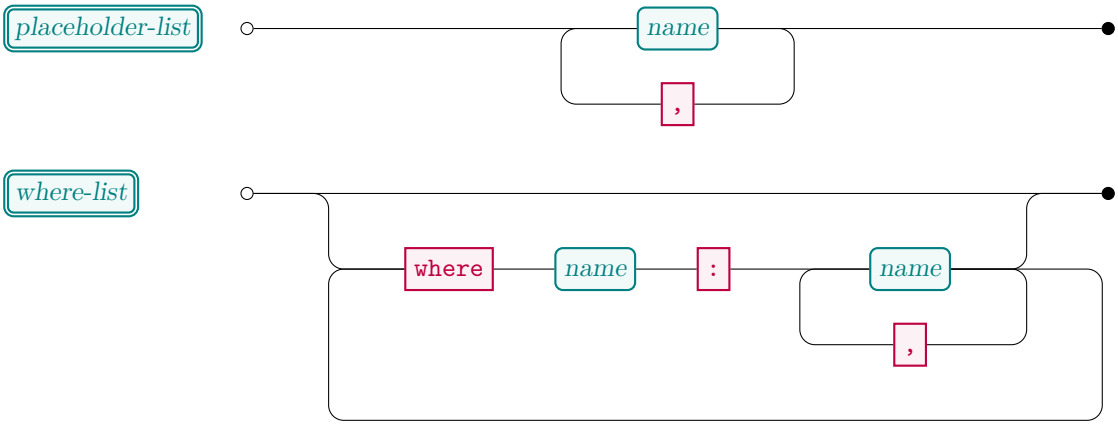
1. ***Starting Point:*** Take a complex type definition as a starting point. As an example we will use the definition of the `IntPair` class from figure 18.1.

2. ***Locate References:*** Find the references to types that we wish to parameterize. In `IntPair`, we need to find the locations of the types that have to match the two elements that the class wrap. These are the types of the two instance variables and the parameter types for the constructor. It is not the return type of the `ToString` method. That has to be `string` in order to match the method prototype that we are trying to override, and it is independent of the of the type of the wrapped elements.

3. ***Insert Placeholders*** For each parameterized type, come up with a placeholder name, and replace each found type reference with the matching placeholder name. For placeholder names we usually use single upper-case characters (often `T`). As both elements of our pair needs to be of the same type, we only need a single placeholder name. Let's just use `T`.

```
class Pair<T> {
  public T a;
  public T b;

  public Pair (T a, T b) {
    this.a = a;
    this.b = b;
  }

  public override string ToString () {
    return "("+a+","+b+")";
  }
}
```

Figure 18.2: Generic pair definition.



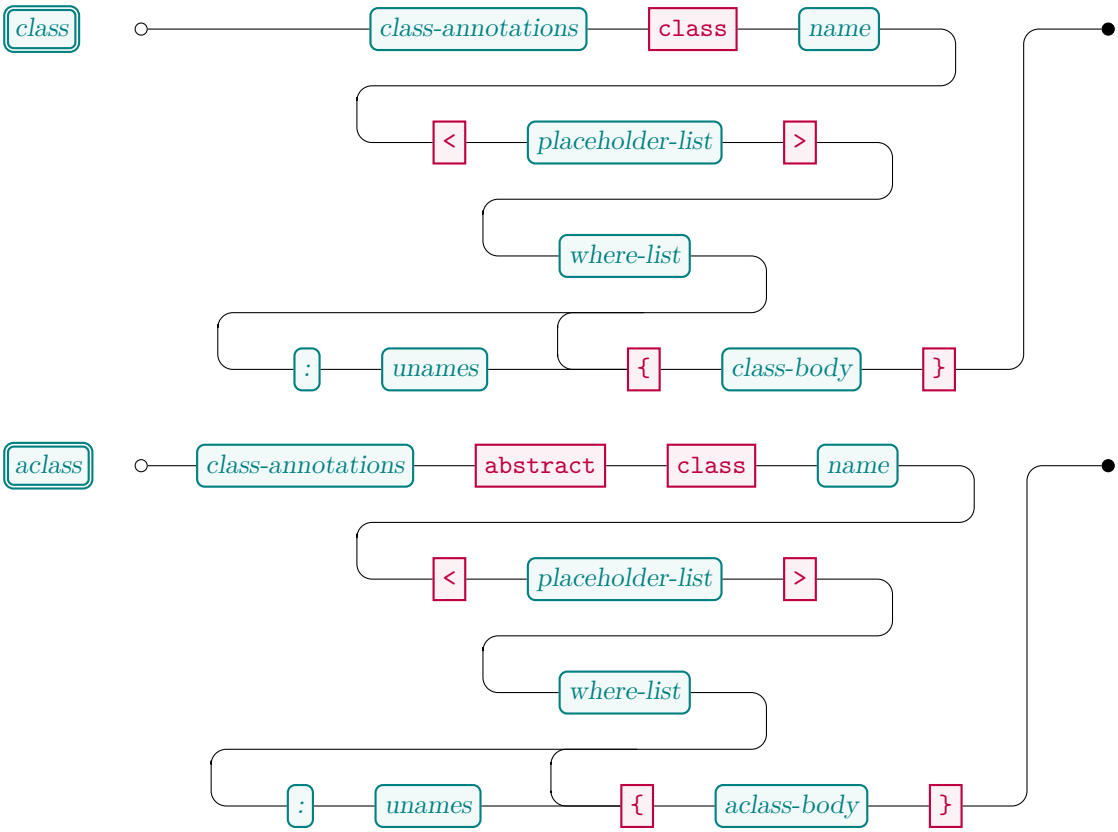
Syntax 18.1: Support definitions for parameterized types

4. ***Declare Placeholders*** Declare that the type is parameterized over the set of placeholder names.

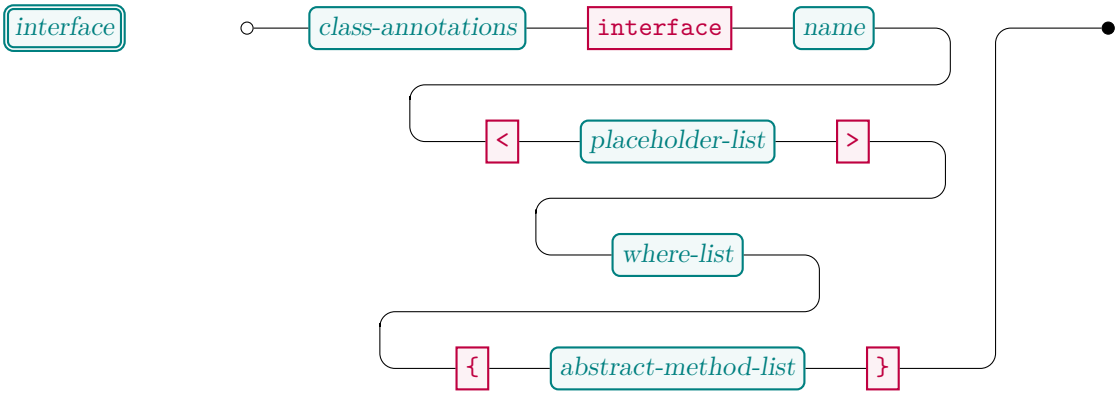
If we follow the recipe on the definition of `IntPair` or `PointPair`, we end up with the generic `Pair` definition in figure 18.2. Given such parameterized definitions, the compiler will automatically generate concrete type variants on demand and that demand is driven by actual use. If the code refers to a pair of `int` values, then the equivalent of `IntPair` will be created. If it refers to a pair of dogs, then the equivalent of a `DogPair` will be created. Note that these *creations* never end up in files.

18.3 C#

By now, we have already seen what a parameterized class looks like in C#. Syntax 18.1 and 18.2 formalizes this. Just like classes can be parameterized, so can interfaces. The rules for these can be found in syntax 18.3. Let's dig into the most interesting details!



Syntax 18.2: Class definitions as parameterized types



Syntax 18.3: Interface definitions as parameterized types

Upon inspection, we see that the updates to the class, aclass and interface rules are similar and restricted to the addition of placeholder-list and where-list. These are defined in syntax 18.1. The placeholder-list is simply a way of declaring which specific placeholders the class or interface is parameterized over. Section 18.3.1 covers this in detail. The where-list is a mechanism for putting restrictions on the allowed placeholder replacement types. Section 18.3.2 covers this in detail. Why don't we need to update the definitions of the class and interface bodies? Because the placeholder name already matches the definition of a type name (see syntax 5.5).

18.3.1 Multiple Parameters

We have established that we can have a need for a single type parameter. If we accept that, then it's not really a stretch to imagine that we can have a need for more than one type parameter: It just requires two relatively independent representation needs to be present in the same class or interface. Syntactically, this is declared as a comma separated list of placeholder names as described in figure 18.1.

18.3.2 Restrictions on Type

We still haven't actually solved to problem of not having guarantees on the available functionality. In our instance and class variables, we can store references to objects, we can update these references and pass them along to other pieces of code. We need to make sure that our code is valid irrespective of the type others choose as parameters, otherwise the compiler won't be happy. Without such such guarantees, we can't do much else with those fields.

One non-idiomatic way of getting around this would be to create a cascade of is checks. That way, we end up with one `if` case per supported type. It tests whether the type of the actual object satisfies some specific type, and – if so – hands it over to some code that knows how to deal with it. Otherwise, it hands control over to the next stage in the cascade. These checks would have to be evaluated in sequence and at runtime, and this would – on top of the readability issues stemming from a deep nesting of `if` statements – carry a performance penalty.

Another solution would be to rely on an interface to define the type contract and polymorphism for dispatch. Any class that we would want to use would then have to implement this interface. While this would be in line with the way interfaces are intended to be used, it is not always a good solution. When we are not in charge of the types that we want to refer to, then these need to be wrapped. That adds complexity and is generally awkward.

Adding restrictions on types improve the situation a bit: Primitive types does not need to be wrapped, and there is more flexibility in how to deal with types that are outside of ones control. Furthermore, restrictions on type also allow for expressing the requirements through multiple interfaces, and for passing along these type restrictions to constructors. These restrictions are expressed in syntax 18.1 under the where-list rule. Here, what is on the left-hand side of the colon has to satisfy what is on the right-hand side.

18.4 Case: Line Segments

Let's imagine that we are writing a `LineSegment` class from which we can create line segment instances. A line segment is a straight line between two points. Clearly, each instance needs to be associated with two points then. One purpose of that class is to calculate the length of such a line segment. This, however, depends of the dimensionality of the space referenced by these points. For the purpose of this example, we would like to support points in either 2d or

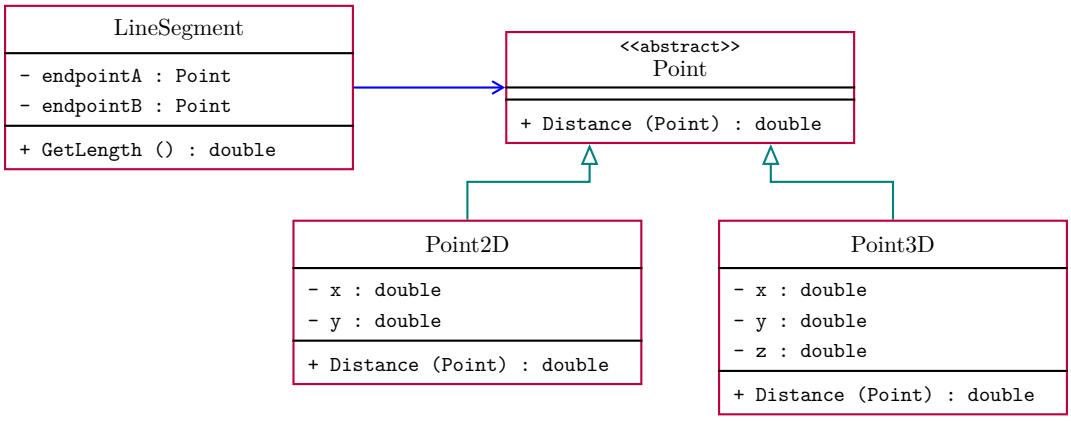


Figure 18.3: `LineSegment` representation using an abstract class.

3d space. As illustrated in figure 18.3, we cound construct such an implementation using the following structure:

- `LineSegment`: A class that represents a line segment that has two `Point` instance variables.
- `Point`: An abstract class that has an abstract function for calculating the distance to another point.
- `Point2D`: A class that represents a point in two dimensions, inherits from `Point` and knows how to calculate the distance to another `Point2D`.
- `Point3D`: A class that represents a point in three dimensions, inherits from `Point` and knows how to calculate the distance to another `Point3D`.

On the surface, this looks good, and it would indeed work. But it would also require us to bypass the type checker, and that comes with some consequences that we would rather avoid. Let's explore this dissonance through a random subclass of `Point`: How would `Point2D` go about calculating the distance to another `Point` in the `DistanceTo` method? It would have to cast this to a `Point2D`, and yet we have no guarantee that such cast would succeed. Perhaps a developer has constructed a `LineSegment` object from a `Point2D` and a `Point3D`. This would be a messup from the developers side, but a messup that we should expect to developer to commit nevertheless. As we have used the downcast to bypass the typechecker we don't really have a good solution available to us. A bad one would be to introduce factory methods for constructing `LineSegment` instances with same endpoint types, and then to use visibility modifiers to hide all other paths to the constructor of `LineSegment`. But then we have essentially just shifted the problem.

The core of the problem is that `LineSegment` needs its `endpointA` instance variable to have a declared type – let's call it `T` – on which a `DistanceTo` method is declared with a parameter of type `T` while at the same time decople `LineSegment` from the specific `Point` subclass. How does this affect our types?

- `LineSegment`: Still a class, but this time we make it parameterized over `T`. This allows us to define `endpointA` and `endpointB` as `T`. While this appears as us just replacing one fairly generic type (`Point`) with another (`T`), this has two implications:

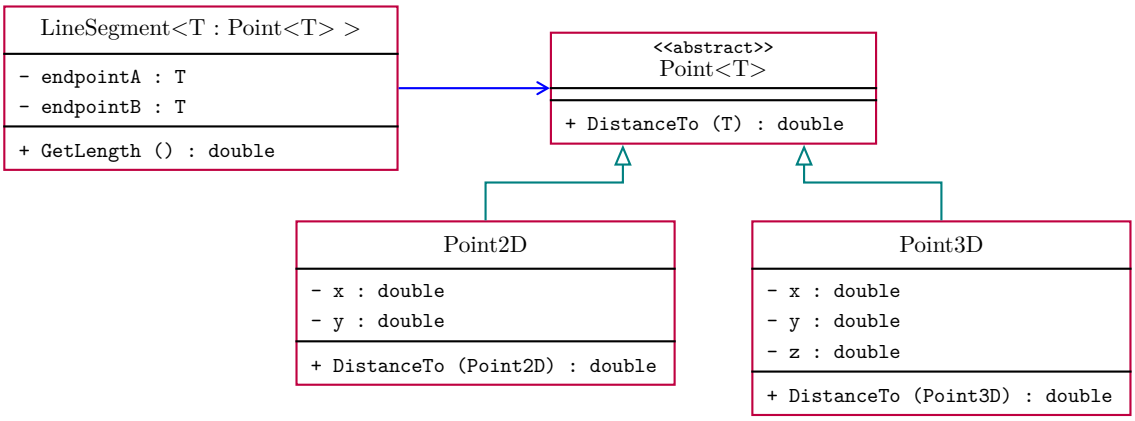


Figure 18.4: `LineSegment` representation using a generic class.

1. The types of `endpointA` and `endpointB` have to satisfy the same concrete value of `T`.
2. `T` is symbolic, and can thus appear as a parameter to a restriction on `T`.

- `Point`: An abstract class that has an abstract function for calculating the distance to an object of the type `T`. We do this for the singular purpose of linking the type of the parameter to the `DistanceTo` method to the concrete descendants of `Point`. Using a parameterized type gives us a lot of flexibility, but we never end up exercising it. Instead, it is locked down in the `Point2D` and `Point3D` classes.
- `Point2D`: A class that represents a point in two dimensions, inherits from `Point` and knows how to calculate the distance to another `Point2D`. That is, it locks `T` to `Point2D`.
- `Point3D`: A class that represents a point in three dimensions, inherits from `Point` and knows how to calculate the distance to another `Point3D`. That is, it locks `T` to `Point3D`.

What we end up with is illustrated in figure 18.4. Granted, it is a rather complicated solution. The choice of applying such complexity should always be considered. In this case we are gaining type guarantees at the expense of complexity. Is that a trade that we are willing to make?

18.5 UML

Looking at figure 18.4, we see syntax elements that are much like C# syntax for parameterized types. This is not proper UML. UML does have ways of describing parameterized types, but it is an offering that suffers from the complexity of the problem that it is trying to solve. Accordingly, it does not belong in this book.

18.6 The C Preprocessor

Behind all of this, C# has a preprocessor for producing variants of parameterized types. The C programming language does not have parameterized types, but it does have a preprocessor. This preprocessor is more generic in nature though, and is not nearly as integrated into the language as the one for C#. Let's take a look at figure 18.5 that describes how the the C preprocessor operates. It is applied to each individual source file before they reach the compiler.

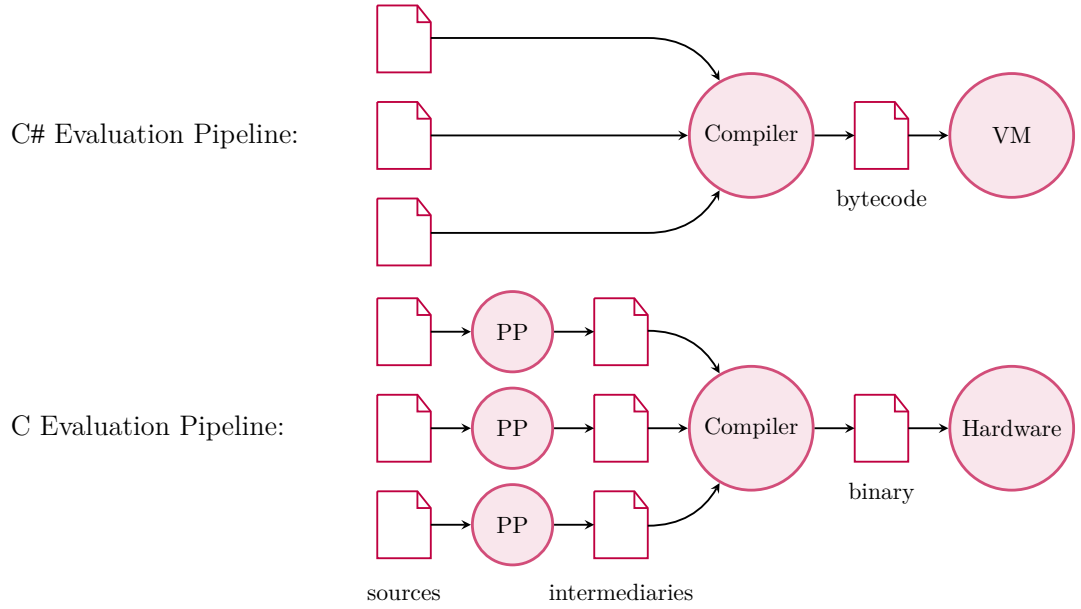


Figure 18.5: Comparing the C and C# evaluation pipelines. A preprocessing step is applied to each individual source file before being handed over to the compiler.

The C preprocessor supports a suite of directives. Such directives appears in the code at the beginning of a line and starts with a hashtag. Let's take a look at a few of these directives:

- Macros can be defined with `#define`. The process of replacing every occurrence of them is referred to as macro expansion. This *expansion* is quite simply a string replacement operation, so it may violate imagined operator precedence.
- We can branch on the definition (or lack thereof) with `#ifdef` and `#ifndef`. Such a branch is ended using `#endif`.
- The contents of another file can be injected using `#include`.

18.6.1 Macro Expansion

The following example makes use of three macro definitions, namely `DEFAULT`, `RADIUS2DIAMETER` and `MULTIPLIER`. The first two are defined in the source file itself. One is parameterized, and one is not:

```
#include <stdio.h>

#define DEFAULT (12)
#define RADIUS2DIAMETER(r) ((r)*2)

int main (int argc, char* argv[]) {
  int radius = DEFAULT*MULTIPLIER;
```

```
  int diameter = RADIUS2DIAMETER(radius);
  printf("r=%d => d=%d\n", radius, diameter);

  return 0;
}
```

The last definition is provided at compile time as a parameter to the compiler:

```
$ gcc -O0 -Wpedantic -DMULTIPLIER=3 define.c -o define
$ ./define
r=36 => d=72
$
```

18.6.2 Conditional Compilation

This example is a bit more complicated, but luckily most of the complexities have to do with operations on time and you only need to understand them at a high level of abstraction. The central piece of the code does a loop with 2^{30} iterations. That is enough for it to take a bit of time. One version of the program can be built that does just this. Another version can be built that times this operation, and prints out how long time it took. The same source code holds the potential to become both programs without any overhead for the first version. It looks like this:

```
#include <stdio.h>
#include <sys/time.h>

#define ITERATION_COUNT (1<<30)

double time_diff (struct timeval t0, struct timeval t1)
{
  double t0_us = (double)t0.tv_sec * 1000000 + (double)t0.tv_usec;
  double t1_us = (double)t1.tv_sec * 1000000 + (double)t1.tv_usec;
  return (t1_us - t0_us)/1000000;
}

int main (int argc, char* argv[]) {
#ifdef BENCHMARK
  struct timeval t0, t1;
  gettimeofday(&t0, NULL);
#endif

  for (long i=0 ; i<ITERATION_COUNT ; i++) ;

#ifdef BENCHMARK
  gettimeofday(&t1, NULL);
  printf("Performed %u iterations in %fs\n", ITERATION_COUNT, time_diff(t0, t1));
#endif

  return 0;
```

```
}
```

At compile time you decide which program to build using a parameter to the compiler. It looks like this:

```
$ gcc -O0 -Wpedantic ifdef.c -o ifdef
$ ./ifdef
$ gcc -O0 -Wpedantic -DBENCHMARK ifdef.c -o ifdef_test
$ ./ifdef_test
Performed 1073741824 iterations in 2.163524s
$
```

18.7 Exercises

EXERCISE 18.1: DYNAMIC ARRAY

Topic
Creativity

Arrays, as you know, have a fixed size. This is fine for many uses, but sometimes you need to be able to insert and remove elements at will. This can be done by keeping track of three values:

1. `data` The underlying array of type `T[]`. The data held by the dynamic array is at the beginning of this array. If you run out of space, another array that is twice as large is created, and the data is copied over. If at some point you use less than half the space, another array that is half as large is created, and the data is copied over.
2. `capacity` The size of `data`.
3. `fill` Number of elements currently used in `data`. This means that `fill` $\leq$ `capacity`.

In this exercise, you must implement a dynamic array that can be parameterized with an arbitrary type in a `DynArray<T>` class. This class must implement the following interface:

```
interface IDynArray<T>
{
  void Append (T element);
  void Insert (int i, T element);
  void Remove (int i);
  void Set (int i, T element);
  T    Get (int i);
  int  GetFill ();
}
```

You can optionally use the following program to test your code:

```
IDynArray<int> a = new DynArray<int>();

Console.WriteLine("Add elements:");
Console.WriteLine(a);
for (int i=0 ; i<20 ; i++) {
```

```
  a.Append(i);
  Console.WriteLine(a);
}
Console.WriteLine("");

Random random = new Random();
Console.WriteLine("Remove elements:");
Console.WriteLine(a);
for (int i=19 ; i>=0 ; i--) {
  a.Remove(random.Next(a.GetFill()));
  Console.WriteLine(a);
}
```

By creatively *overriding* the `ToString` method on `DynArray<T>`, you can make this program illustrate how the individual operations work.

Chapter 19

Collections

19.1 Dynamic Arrays

19.1.1 Implementation

19.1.2 C#

19.2 Linked Lists

19.2.1 C#

19.2.2 Elixir

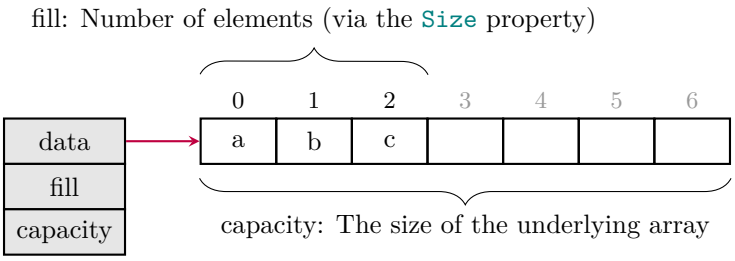


Figure 19.1: Structure of a dynamic array.

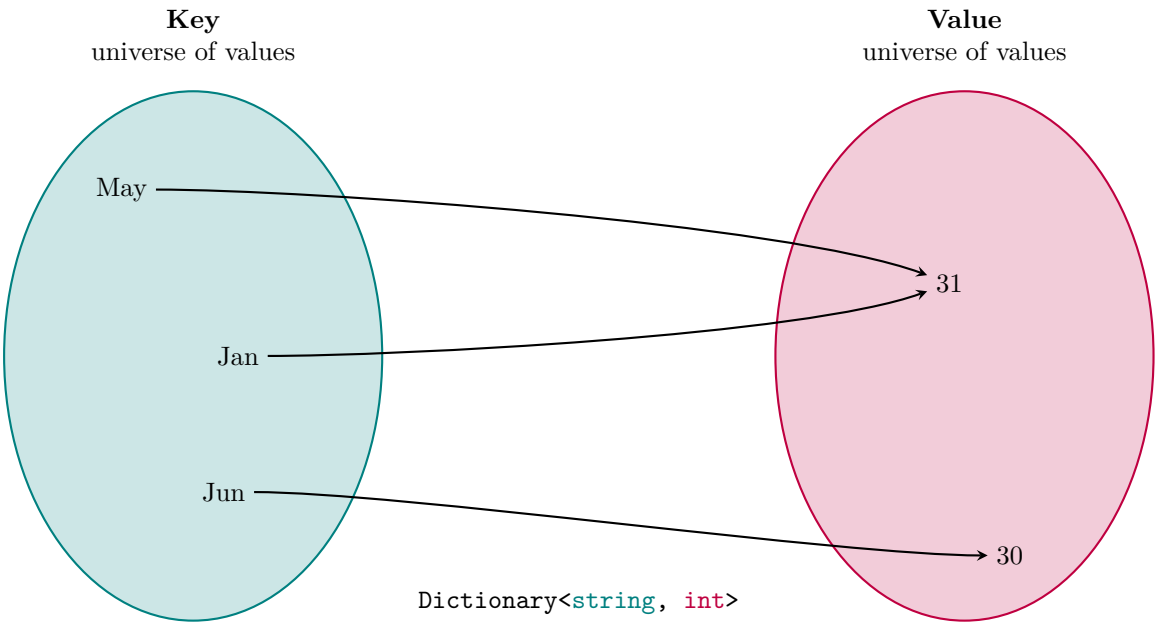


Figure 19.2: Logical model of a map.

19.3 Maps

19.3.1 Hash Tables

19.3.1.1 Hashing

19.3.1.2 Organization

19.3.2 C#

19.3.3 Elixir

19.4 Sets

19.4.1 Set Operations

19.4.2 C#

19.4.3 Elixir

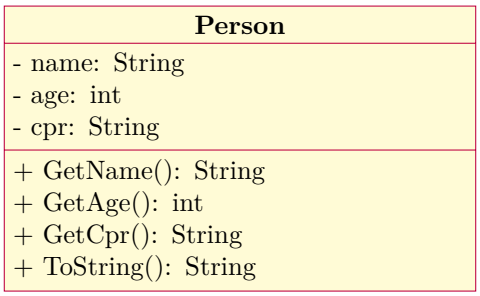
19.5 Exercises

Exercise 19.1: CPR Registry

Topic
Creativity

Subtask 1

Implement in C# a class for this UML diagram.



Note: The `cpr` field refers to a Danish social security number format called CPR.

Hint: When the `ToString` method is defined in the UML class, then you have to override the method and give it a meaningful implementation.

Subtask 2

Make a `List` that contains 5 `Person` objects. You decide the attribute values. One person must have the CPR number `010101-0101`.

Subtask 3

Loop through your `List` object and find the person with the CPR number `010101-0101` by calling the method `GetCpr` on each object.

Print the result of an (explicit or implicit) call to the `ToString` method on this object.

Subtask 4

Create a `Dictionary<string, Person>` map and add the `Person` objects to this map that you previously added to the `List` object in subtask 2. Use the CPR number as the *key*.

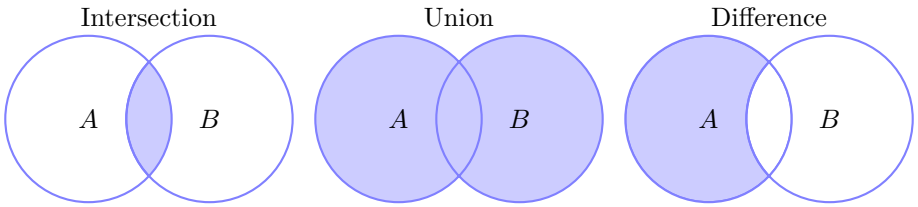
Subtask 5

Find the person with the CPR number `010101-0101` in the `Dictionary` object and print the value returned from a call to the `ToString` method on this object.

Exercise 19.2: Nondestructive Sets

Topic
Creativity

From highschool, you know the following set operations:



These operations can be implemented using the instance methods provided by `Set`. These operations, however, are destructive and that's often not what you want. Lets make a different variant.

Subtask 1

First, declare a `NondestructiveSet<T>` interface (i.e., a parameterized `NondestructiveSet`) with the methods:

- `Intersection` The intersection contains all elements that are in both sets.
- `Union` The union contains all elements that are in at least one of the sets.
- `Difference` The difference contains all elements from one set that are not in another set.

These methods all take a `NondestructiveSet<T>` parameter and return a new `NondestructiveSet<T>`.

Subtask 2

Next, you need to implement a `NondestructiveHashSet<T>` class that inherits from `HashSet<T>` and implements `NondestructiveSet<T>`. Make sure that none of the methods from `NondestructiveSet<T>` affect their inputs.

Subtask 3

Demonstrate that your implementation works as expected.

Subtask 4

Extend your `NondestructiveSet<T>` class with methods for:

- `IsDisjoint` which returns `true` if and only if the `NondestructiveSet<T>` argument it receives has no elements in common with itself.
- `IsSubset` which returns `true` if and only if all elements in its parameter (of type `NondestructiveSet<T>`) exist in itself.

Implement these in `NondestructiveHashSet<T>` and demonstrate correctness.

Chapter 20

Sorting

20.1 Problem Generalization

20.1.1 Comparison

20.1.1.1 C#

20.1.1.2 Elixir

20.2 Sorting Algorithms

20.2.1 Bubble Sort

20.2.1.1 C#

20.2.2 Random Sort

20.2.2.1 C#

20.2.3 Insertion Sort

20.2.3.1 C#

20.2.4 Merge Sort

20.2.4.1 C#

20.2.4.2 Elixir

Chapter 21

Input and Output

21.1 Command-Line Arguments

21.1.1 C#

21.1.2 Elixir

21.1.3 Python

21.2 Environment Variables

21.2.1 C#

21.2.2 Elixir

21.2.3 Python

21.3 Strings

21.3.1 C#

21.3.2 Elixir

21.3.3 Python

21.4 Standard Streams

21.4.1 C#

21.5 Files

21.5.1 C#

21.6 Interprocess Communication

Let's look into what happens when you open a file in your file mananger. Do you know? First, the file type is determined. How this is done depends on the specific file manager, but once the system knows the type of the file, it will use that to look up the default program for handling

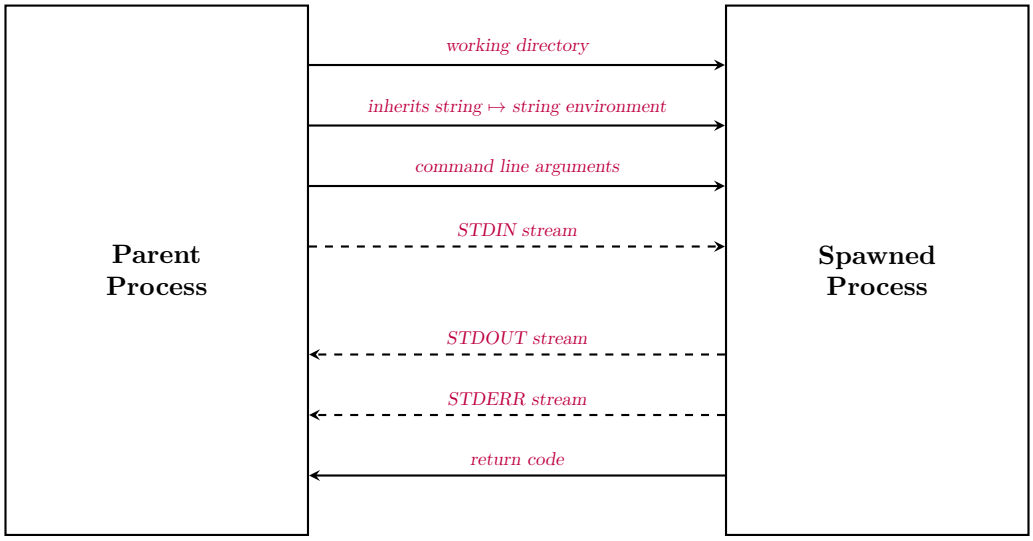


Figure 21.1: The interface between a process and a child process.

this type of file. Once it knows which program should handle it, it will locate the executable of that program and start it with the path to the chosen file as a parameter.

So, one process starts another one. We often refer to the first process as the parent process and the second one (that is being spawned) as the child process. In reality, this parent-child relationship represents a single edge in a logical process tree that is rooted in an init process. There is a lot more complexity to it, but this will do for now. Let's take a look at how that pair of processes can communicate. This is (mostly) summarized in figure 21.1.

When a process spawns another process, it does so in a context. The parent process can configure this context as it sees fit. It consists of:

- A working directory (see section 2.2.3).
- A list of parameters. This is the primary means of conveying what concrete task the invoked process should perform.
- An environment. This is a mapping from strings to strings. Certain keys have well-defined roles. One of the most important keys will be covered in the following paragraph.

When the parent process asks the kernel to start a new process, it needs to point to the executable code that this new process should execute. This is often done by searching the paths listed in the `PATH` environment variable for a filename. This environment variable contains a list of directories (separated by colons) that should be searched for executables. The first match is what will be executed and if no match is found then an error will be reported instead.

The spawned process is tied to the parent process through three data streams. Through one data stream the parent process can send data at will to the child process, and that process can read these data. Through two channels (of streaming data) the child process can send data back to the parent process: One is the primary channel and one is the channel for communicating error messages. We will cover these in section 21.4.

Once a process has finished executing, it will report back an integer to its parent process. This integer is called the return code. By convention, a zero means that everything went well while a non-zero number indicates a specific type of error.

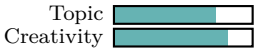
Technically, there are a few other ways of communication available to the pair of processes. All processes have a unique process id. The parent process will know both the process id of itself and that of the of child. Throgh a process if, you can send a signal to that process. Signals are typed and have no payload, so they can really only be used to flag events. Processes can also communicate through files: One process writes to it while anonther reads from it. Lastly, processes can communicate through sockets. We will skip sockets completely in this book, but they are the fundamental abstraction through which Internet communication flows.

21.7 Exercises

Exercise 21.1: Command Line Arguments

Write a program that takes two parameters from the command line, converts them to integers, adds them together, and prints the result to the screen. The two arguments come from the parameter to the `Main` method, which is usually called `args`.

Exercise 21.2: Keyboard Input



Write a program that waits for the user to enter some text, and then prints this text in reverse order, so that "`hello`" becomes "`olleh`".

Note: This is a surprisingly difficult problem to solve, and an even more difficult problem to understand. The problem here is that while it is easy to access a specific `char` in a `string`, it is complicated to find out how many `char` values make up a character; and the typical interpretation of the problem works at the character level. If you want a correct implementation of the reverse order, see here: `https://rosettacode.org/wiki/Reverse_a_string`. However, this is not a key element of the problem.

Exercise 21.3: Writing to File

Define a data structure in which one can look up the name of the ith month, and write a method that stores it to a file.

Exercise 21.4: Reading from File

Write a program that – from a file – populates a datastructure so that one can look up the name of the ith month.

Exercise 21.5: Highscores

Many games maintain a list of *highscores*.

Describe abstractly – for the purpose of implementing this – which data structure should be used to implement such a list.

Next, create a class that implements a set of highscores:

1. A class named `Highscores`.
2. This class has a constructor that takes a file name as an argument.
3. This class has a `Load` and a `Save` method that does what you expect them to do.
4. This class has a `NewScore` method that should be called every time a game is finished.
5. This class also has methods to extract the relevant information.

Finally, create a program that tests this class.

Exercise 21.6: CSV Files

The CSV (Comma Separated Value) file format is a simple way to store a 2D table of data (as we know it from a spreadsheet). In this format, one row is stored at a time. A line break is used to differentiate between two rows. In each row, the individual cells are separated by a comma. Microsoft Excel is known to misinterpret formats by replacing the comma in the role of *separator character* with a semicolon. The following is an example of a CSV file:

```
1,1.2,1.5,1.0,1.0
2,2.4,3.0,4.0,1.0
3,3.6,4.5,9,0,1.0
4,4.8,6.0,16.0,1.0
5,6.0,7.5,25.0,1.0
6,7.2,9.0,36.0,1.0
```

What happens if you store the text "`1,2`" in a cell?

Now create a new class with the name `Matrix`, and implement the following:

1. An instance attribute with the name `data` of the type `double[][]`.
2. A constructor that takes a file name as an argument and loads `data` from it.
3. A constructor that takes two integers as arguments and assigns `data` a "2d" array of this size where all cells have the value 0.
4. Appropriate getter and setter methods. By "appropriate" we mean the getter and setter methods that are needed. Note that these take two `int` parameters – in addition to the ones you normally see – in order to be able to specify a cell.
5. A `Save` method that saves the object's current state (read: `data`) to a file whose name is taken as a parameter.

Now create a program that uses this class to create a 4*6 table with 0 values. Verify using a spreadsheet program that the result is correct. That is, that it can save a file with the contents:

```
0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0
```

```
0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0
```

Finally, create a program that reads the resulting file, adds one to the diagonal where $x = y$ and saves the result in a new file. This file should end up having the following content:

```
1.0,0.0,0.0,0.0
0.0,1.0,0.0,0.0
0.0,0.0,1.0,0.0
0.0,0.0,0.0,1.0
0.0,0.0,0.0,0.0
0.0,0.0,0.0,0.0
```

Exercise 21.7: Filesystem Traversal

Create a program that, given a path to a folder on your computer, searches the part of the file system below it and along the way builds a list of all CSV files. Finally, it prints out the paths to these.

Exercise 21.8: Shell

Create a program in which the user is presented with a prompt through which commands can be entered. More precisely:

1. Create a `Main` method.
2. In this method, create a `bool` variable named `done` and initialize it to `false`.
3. Place a `while`-loop inside the `Main` method and use as condition `!done`. Try to pronounce this line!
4. The rest of the code will be placed inside this `while` loop.
5. First, a prompt without line breaks is printed. This could be the text string "`$ `".
6. Next, a line is read.
7. Split this line on spaces.
8. A reference to the resulting list is stored in a variable named `command_args`. What would a reasonable type for this be?
9. A reference to the first element of `command_args` is stored in a variable named `command`.
10. Use a `switch` construct on `command` to implement a series of commands. Implement at least the following commands:
    a) ***exit*** This sets the variable `done` to the value `true`.
    b) ***echo*** This prints the same text as it received.

c) ***time*** This prints the current time.

11. Make sure to print an informative error message if `command` is not recognized.

Test that the above functionality works.

Exercise 21.9: Turtle Interpreter

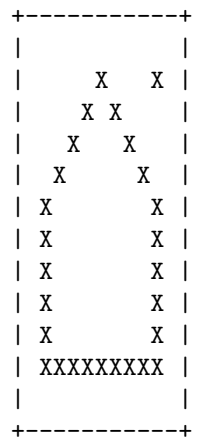
Topic
Creativity

In this assignment, we are going to create a drawing program. So we start by creating a class called `Canvas` whose constructor takes two integer parameters (a width and a height) and assigns an attribute called `data` a `bool[][]` of this size. Add getters and setters, and a method called `Print` that prints the contents in a nice way.

We draw by giving instructions to a turtle with a pen. So we create a `Turtle` class with two `int` attributes (`x` and `y`) for position, a `bool` attribute called `draw` to indicate whether to draw when it moves, and a `Canvas` attribute that tells which canvas it is on. Also add getters, setters, and "movers". By "movers" we mean methods such as `GoNorth` that adjust the turtle's coordinate and draw on the canvas if `draw` is true.

Now define a command language that makes it possible to perform a sequence of operations on such a turtle. There could be, for example, one command per line. Commands could optionally be parameterized so that you could write `north 5` to move the turtle 5 spaces north. Implement a test program with a `Main` method that reads a file that meets this format (i.e., the command language), executes the commands, and prints the result to the screen.

Create a file that meets your format and draws the following image:



Exercise 21.10: Sūdoku Visualization

Topic
Creativity

Subtask 1

Save the following as `example.svg` and open the file in a browser:

```
<svg width="600" height="600" xmlns="http://www.w3.org/2000/svg">
  <rect x="0" y="0" width="600" height="600" style="fill:#ff8800"/>
  <rect x="10" y="10" width="580" height="580" style="stroke:black;stroke-width:5;fill:white"/>
```

```
  <line x1="10" y1="10" x2="590" y2="590" stroke="black" stroke-width="2"/>
  <text x="400" y="200" text-anchor="middle" font-size="28">Some text</text>
  <text x="200" y="400" text-anchor="middle" font-size="28" font-weight="bold">Other text</text>
  <circle cx="600" cy="600" r="300" fill="none" stroke="#0000ff" />
</svg>
```

This is an example of a file format called Scalable Vector Graphics (SVG). It is the only format for vector graphics that is widely supported on websites. The above is just a small sample of what you can do in SVG.

Subtask 2

Try modifying the individual values (but do not touch the reference to `http://www.w3.org/2000/svg`) to get an understanding of how the different elements affect the way the browser *draws* the file.

Subtask 3

Find your solution from the exercise in section 8.3, or – alternatively – start from the reference solution for it in section 69.

Your task is now to create a new variant of this code base that produces an SVG file instead of printing some characters to the screen. Make it look pretty!

Part IV

Tooling

Chapter 22

Tooling

Hello

22.1 Refactoring

22.2 Debugger

22.3 Profiler

22.4 Exercises

Exercise 22.1: Timing

Subtask 1:

Write a program in which:

1. There is a single class called `Timing`.
2. This class contains a single static method called `Fun`. This method takes two `double` parameters (`x` and `y`), and returns a `double`. This method is recursively defined so that:
   - If $y \leq 1$ then it returns the value of `x`.
   - Otherwise ($y > 1$), it turns the value of the following expression: `Fun(x,y-1) * Fun(x,y-1)`.
3. The same class contains a `Main` method which for `x = 1.0000001` iterates through all `y` values from 1 to 32 (both included). For every combination of `x` and `y`, it evaluates `Fun(x,y)`, prints out the resulting value, and times how long time the evaluation takes.
   - The static method `DateTimeOffset.Now.ToUnixTimeMilliseconds` returns a timestamp measured in milliseconds in the form of a `long`.
   - By subtracting one such timestamp from another, you get the time in between.

Subtask 2:

Does this implementation calculate `Fun(x,-1)` correctly, and why is this the case?

Subtask 3:

Is this otherwise a good implementation of `Fun(x,y)`?

Exercise 22.2: Date

Topic
Creativity

Write a program which

1. Constructs a `DateType` object for each of the timestamps 10^3, 10^4, 10^5 and 10^6.
   - First, use the `System.DateTimeOffset.FromUnixTimeMilliseconds` method to create a `DateTimeOffset` object from the timestamp.
   - Then acess the `DateTime` attribute.
2. For each of these, print out the time represented by the object by using its implementation of the `ToString` method.
3. Verify that these printouts makes sense by reading the documentation[1].

Exercise 22.3: Terminal Use

Topic
Creativity

Subtask 1

Make a program that prints out the width and the height of the terminal (measured in characters) every second. Verify the at program works by resizing the window.

Hints:

- The width and height of the terminal is available throgh the `WindowWidth` and `WindowHeight` properties on the `Console` class.
- The `Thread.Sleep` method takes an integer parameters and sleeps for this many ms.
- You can exit the program by killing the process (e.g., by pressing Ctrl-C).

Subtask 2

Make a program that draws a crosshairs in the terminal, and updates this every second. That is, a vertical and a horizontal line should split the screen in four areas of equal size. As this will only be possible with combinations of odd widths and height, we can accept the dimensions of the areas to be off by a single character.

Subtask 3

Now we have a canvas to draw on. Lets simulate a ball bouncing around inside the window.

Hints:

- Draw the ball as a "`O`" character.

[1] https://learn.microsoft.com/en-us/dotnet/api/system.datetimeoffset.fromunixtimemilliseconds

- Hardcode the starting position to be (0,0). That is, the upper left corner.
- Hardcode the initial direction to be (1,1). That is, south east bound.
- Assume that the terminal will not be resized.
- Reduce the sleep time from 1000ms to 100ms in between each simulation step.

Subtask 4

The implementation from the last subtask had some significant limitations. For instance, the initial direction was fixed and a single step on each axis. That simplified things. Lets unsimplify it by allowing any combination of two integers to define the initial direction. You can start by getting (1,2) to work and then also make it work with (3,2).

Note: This is a significantly harder task.

Chapter 23

Debugging

23.1 Exercises

Exercise 23.1: Locate the Mistake

Topic
Creativity

The following program – given an arbitrary value for the variable `points` – is supposed to tell you whether you have passed a course:

```
class Score {
  public static void Main (string[] args) {
    int points = 0; // imagine an arbitrary number of points between 0 and 100

    Console.Write("You ");

    if (points >= 50) {
      Console.Write("PASSED");
    }

    if (points <= 50) {
      Console.Write("FAILED");
    }

    Console.WriteLine("!");
  }
}
```

However, a bug has crept in!

Locate and fix the bug.

Exercise 23.2: Correctness

Topic
Creativity

To test how good C#'s built-in random number generator is, we create two arrays of random numbers and subtract one from the other. We repeat this 10 times, and for each iteration we print the average deviation.

What should we expect to happen with the result of this test?

This test is implemented as follows:

```
class RandomTest {
  const int ITERATIONS = 10;
  const int LENGTH     = 48;

  static Random r = new Random();

  static int[] Init (int length) {
    int[] a = new int[length];
    for (int i=0 ; i<a.Length ; i++) {
      a[i] = r.Next() % 100;
    }
    return a;
  }

  static int[]? Diff (int[] a, int[] b) {
    // guard: uneven length of inputs
    if (a.Length!=b.Length) return null;

    int[] d = new int[a.Length];
    for (int i=0 ; i<d.Length ; i++) {
      d[i] = b[i] - a[1];
    }
    return d;
  }

  static void Print (int[]? a) {
    // guard
    if (a==null) {
      Console.WriteLine("NULL");
      return;
    }

    int sum = 0;
    Console.Write("[");
    for (int i=0 ; i<a.Length ; i++) {
      // break line nicely
      if (i!=0 && i%24==0) {
        Console.WriteLine("");
        Console.Write(" ");
      }

      Console.Write("{0,5}", a[i]);
      sum += a[i];
    }
    int mean = sum/a.Length;

    double vsum = 0;
```

```
    for (int i=0 ; i<a.Length ; i++) {
      double diff = a[i]-mean;
      vsum += diff*diff;
    }

    Console.Write(" ] avg={0,4} stdev={1,5:0.00}", mean, Math.Sqrt(vsum/a.Length));
    Console.WriteLine("");
  }

  public static void Main (string[] args) {
    for (int i=0 ; i<ITERATIONS ; i++) {
      int[] a1 = Init(LENGTH);
      int[] a2 = Init(LENGTH);
      int[]? ad = Diff(a1, a2);
      Print(ad);
    }
  }
}
```

When you run it, you get:

```
[   -1  -82   44  -74  -13   67   -5   13  -18   54  -45    6   99  -75   21  -83   93   53   17   56    1   85  -13  -32
    18   86    8   24  -78    2   49   52   91  -19  -67   16  -74   10   99  -80   83  -14  -10  -21  -10  107  -88  -32 ] avg=   6 stdev=56.07
[  151  170  112  112   79  108   76  152  186   30  135   48    1   57  196   77  185   67   96  168   92   57  190   42
     32   94   30   62  130  116    6   49  124   62  124  166  194   22  150  159  171   36   43   19  146  196  100  181 ] avg= 104 stdev=58.27
[  -95  -94  -37  -59  -18  -70 -108  -61 -158   36  -76 -141  -12 -105  -66  -86  -54  -52 -131    2  -73 -107  -35   40
    -89  -72   24 -127  -33   16  -35  -76   -4  -82 -106   23   22  -33  -99 -155 -134  -67  -59  -68    4 -132 -142  -48 ] avg= -63 stdev=52.35
[  -31  -66  -55   36   52  -71   -3   38   59  -52   69  122    0   97   18    5   63   93  118  118   12   27   87   41
    103    0  -41   48  -18  119   37  -59  -10   59  -37  114   89   14  -25   39  -14  112   19  -27  -65   43   37  100 ] avg=  29 stdev=57.15
[   16  -86   30   18  -22   52   49 -124  -56   34  -40  -78 -118  -19 -117   38   53  -53   -7   29   57   30 -123 -115
     38  -73   -1    7  -90   58   56   -5  -22   64   31  -96   35   30   29  -76   62   64   31   11 -129  -55 -106  -78 ] avg= -15 stdev=62.75
[  -63  -78  -74  -97  -33    9  -87   22 -127  -86 -165  -22  -71  -15  -77  -44  -18  -12  -12  -30 -146  -39 -133   18
     18  -65  -37  -92 -145 -169 -128 -105    0   25 -102 -155  -75   -2 -153  -19  -81  -65 -157 -151   -8 -121   26  -63 ] avg= -66 stdev=57.82
[  -89 -146   31 -139   27   22  -28   30 -115 -125  -50  -50 -140   27  -37  -85  -80 -111  -52   29  -32  -43  -12 -140
    -67    9  -64    8   28 -145 -131 -109  -28  -51 -118   11 -131 -117 -108  -28 -117  -77 -145  -89 -142   42   42  -23 ] avg= -59 stdev=61.75
[ -107 -126  -47 -138   19   26  -33   35   35    0  -73   -9  -66  -20  -59  -48  -67 -117  -33 -128 -113    7   45 -104
    -47   33 -130  -19 -112    6 -115  -34 -121 -134   42  -27 -111  -94  -31  -52  -60 -118  -72   -1  -10   33 -112  -44 ] avg= -51 stdev=56.10
[  -15   38  -95   78  -98 -114  -73  -49   -7   66  -48   41  -71   -7  -86   67  -15  -88  -86   73   -6   46   27  -17
     14   48  -55    3    1  -15    2  -78  -12   59   42  -23  -16  -18  -97   72   61  -25  -41  -21   -9   16  -75  -48 ] avg= -13 stdev=53.06
[  134  114  167   20  151  134   17  -14   22  145  126   98  156  168  180  119  -14  129   22    4   34  149   83   88
     90   28  -12   21   39  182  177  119   55  143  118   -1  166  153   79   70  178   69   -7  122  129  117  126   24 ] avg=  92 stdev=61.70
```

Is this a correct result? If so, is it a good result? If not, what went wrong and how can it be fixed?

Exercise 23.3: Sum

Topic
Creativity

In the following program, there is a `Sum` method that is written to sum up the values of a series of numbers. The program also has a `Main` method that defines two test cases and prints the result of calling `Sum` on each of them. However, it seems that `Sum` is not written correctly. Can you figure out what is wrong and fix the code?

```
public class Adder
{
  public static int Sum (int[] numbers) {
    int result = 0;
    for (int i=1 ; i<=numbers.Length ; i++) {
```

```
      result += i;
    }
    return result;
  }

  public static void Main (string[] args) {
    int[] testcase1 = {1, 3, 5, 2, 3, 7};
    int[] testcase2 = {5, 6, 2, 3};

    Console.WriteLine(Sum(testcase1));
    Console.WriteLine(Sum(testcase2));
  }
}
```

Exercise 23.4: Happiness

Topic
Creativity

The following program describes a situation where a person wants to buy 15 items, each costing 6000 DKK. This person has an amount in his bankbook and owns a house of modest value. In addition, the individual has a distorted perception of his own happiness. The following program *simulates* how the individual purchases affect the individual's happiness.

```
public class Happiness
{
  const int ITEM_PRICE  = 6000;
  const int ITEM_COUNT  =   15;
  const int HOUSE_VALUE = 8000;

  static int  account   = 30000;
  static bool ownsHouse = true;

  static bool AccountIsPositive () {
    return account>0;
  }

  static bool BuyItem (int price) {
    account -= price;
    return AccountIsPositive();
  }

  // returns true if able to sell house or no sale necessary
  static bool SellHouseIfNecessary () {
    if (!AccountIsPositive()) {
      if (!ownsHouse) return false;

      account += HOUSE_VALUE;
      ownsHouse = false;
    }
    return true;
  }
```

```
  static void PrintState () {
    Console.WriteLine("account="+account+" ownsHouse="+ownsHouse);
  }

  public static void Main (String[] args) {
    bool happy;

    for (int i=0 ; i<ITEM_COUNT ; i++) {
      happy =  (AccountIsPositive() && BuyItem(ITEM_PRICE))
            || SellHouseIfNecessary();
      Console.WriteLine("I am "+(happy ? "" : "not ")
                        +"happy and my account is "+account);
    }
  }
}
```

Subtask 1

The `Main` method consists of a loop where each iteration (through some method calls) works on the variables `account`, `ownsHouse` and `happy`.

Start by running the program manually. This means that you sit down with a piece of paper (you decide whether it should be physical or virtual). On this paper you draw a table with a column for each of the three variables and a line for each of the 15 iterations. For each iteration, you now fill in the values that the three variables have after calculating happy.

How do you do that? Well, you pretend to be the CLR virtual machine that can interpret compiled C# code, follow the method calls that are made and note which variables are written to along the way.

What will the program print out?

Subtask 2

Get the CLR virtual machine to execute the program. What does it actually print out?

Subtask 3

Is there a match between what you have guessed it will print out and what it actually prints out?

In the event of a mismatch, you must track down where this mismatch occurs, and investigate whether there is a problem in your understanding of how C# interprets the code, or whether you had simply overlooked something. To do this, you insert printouts at appropriate places in the code.

Chapter 24

Testing

Hello

24.1 Impossible Completeness

24.2 Equivalence Partitions

24.3 Edge Cases

24.4 Test Methodologies

24.5 Unit Testing

24.5.1 C#

24.5.2 Elixir

24.6 System Testing

24.7 Acceptance Testing

24.8 Exercises

Exercise 24.1: Pin

When paying wirelessly with a credit card, there are rules for when to provide a PIN. One of the rules could be that you must always provide a PIN when the amount exceeds 350 DKK. Another could state that you – regardless of the amount – must provide a PIN at least every fourth transaction.

The method Expend in the following class has been developed to decide whether it is necessary to ask for a PIN:

```
public class Pin
{
  const int CRITICAL_AMOUNT = 350;
  const int MAX_TRANSACTIONS = 4;
  private int time2pin = MAX_TRANSACTIONS;

  bool Expend (int amount) {
    if (time2pin==0 || amount>CRITICAL_AMOUNT) {
      time2pin = MAX_TRANSACTIONS;
    }
    return amount > CRITICAL_AMOUNT || --time2pin == 0;
  }

  public static void Main (string[] args) {
    Pin pin = new Pin();

    for (int i=0 ; i<20 ; i++) {
      bool give_pin = pin.Expend(42);
      Console.WriteLine(i + ": 42 -> " + give_pin);
    }
  }
}
```

Subtask 1

Investigate how the method Expend works.

Subtask 2

The code is built to comply with the two rules described at the beginning of this exercise. For each of these rules, you must now find the boundary cases. Boundary cases are those situations that define the edge of there a condition shifts between true or not. For example, 17 and 18 are boundary cases for some logic that determines whether a person has the right to vote (assuming that you have to be 18 to vote).

Subtask 3

Some rules are expressed in the problem text, and others may have arisen during the implementation. Create a test case that represents each combination of boundary values, and execute these test cases. Now verify that the method Expend works correctly by looking at the results of executing the program that you build from the test code.

Subtask 4

How would the testing strategy change if there was a random number generator in the picture such that the probability that you would be asked to provide a PIN code was 20%?

Chapter 25

Software Architecture

Hello

25.1 Purpose

25.2 Layered Architectures

25.2.1 3-Layer Architecture

25.3 Microkernel Architecture

25.4 Node Graph Architectures

25.5 Microservice vs Monolith

25.5.1 Elixir

25.6 Exercises

Exercise 25.1: Inventory

Topic
Creativity

These tasks build on the series of tasks about a storage system. The task includes the code from section 25.6.0.1 below. There is quite a bit.

Subtask 1

Create a new project in your development environment and add the files from section 25.6.0.1. Make sure that the `Main` method in `Test` runs and that it behaves in a reasonable way.

Subtask 2

The code is divided into 3 namespaces. Look through the code and count how many references there are from one file to another. A reference means a method call or an access to an attribute/variable. It is not so important that the numbers are exact and you can just focus on references that cross namespaces. Furthermore, we are not interested in whether the references

are at the object or class level (count both). Complete the following table so that one field contains the number of references from a file on the x-axis to a file on the y-axis. Time how long it takes to complete the table.

	data/FileBackend.cs	domain/Item.cs	domain/FoodItem.cs	domain/NonFoodItem.cs	domain/ItemNameLengthComparator.cs	domain/Expirable.cs	domain/ExpiredItemAddedException.cs	domain/Inventory.cs	presentation/Test.cs
data/FileBackend.cs									
domain/Item.cs									
domain/FoodItem.cs									
domain/NonFoodItem.cs									
domain/ItemNameLengthComparator.cs									
domain/Expirable.cs									
domain/ExpiredItemAddedException.cs									
domain/Inventory.cs									
presentation/Test.cs									

Subtask 3

Describe the responsibilities of each namespace.

Subtask 4

Based on the analysis in subtask 2, describe the extent to which these responsibilities depend on each other.

If you want to change a file in the `Domain` namespace, which files could you risk having to adjust in order not to introduce an error? In which namespaces are these files located? How do these answers depend on *which* file the change is made in?

Subtask 5

How would the answer to the previous task change if we had another project (that we did not know about) that used our `Domain` namespace?

Subtask 6

Now create a copy of the project and add a namespace named `Interface` with an interface named `IDomain`. This interface must contain the following content:

```
namespace Interface {
  public interface IDomain {
      public void Load ();
      public void Store ();
      public void AddNonFoodItem (string name, double price, string[] materials);
  }
}
```

Subtask 7

Now create a class named `Domain` in the namespace `Domain`. This class must be declared `public` and implement the interface `Interface.IDomain`. This class must now function as the only link between the `Domain` namespace and the `Presentation` namespace.

To do this, the constructor must take a filename of type `string` as an argument. This filename must be stored in an object attribute named `filename`. In addition, the constructor must create an `Inventory` object and store a reference to it in an object variable named `inventory`.

Override all the methods from `Interface.IDomain`:

- `Load` This method should call `Load` on `inventory` with `filename` as argument.
- `Store` This method should call `Store` on `inventory` with `filename` as argument.
- `AddNonFoodItem` This method should create a `NonFoodItem` object with the parameters it has received, and call `AddItem` on `inventory` with this object as argument.

Subtask 8

Modify the access modifiers in ALL other classes in the namespace `Domain` to be default (in other words, remove the `public` keyword before the class names).

Update all classes in the namespace `Presentation` to create an object of type `Domain.Domain`, store this in a variable of type `Interface.IDomain` and make sure that all references to the `Domain` namespace go through this object.

Make sure that the `Main` method in `Test` can run and that it behaves in a reasonable manner.

Subtask 9

Now we have two versions of this codebase: the released one and the updated one. They do the same thing, but have different code qualities.

Let's repeat **subtask 4** for the updated codebase: If you want to change a file in the `Domain` namespace, which files could you risk having to adjust in order not to introduce an error? In which namespaces are these files located? How do these answers depend on *which* file the change is made in?

Let's repeat **subtask 5** for the updated codebase: How would the answer to the previous task change if we had another project (that we didn't know about) that used our `Domain` namespace?

Subtask 10

In what ways is which codebase better?

25.6.0.1 Provided Code

The following code is divided into 3 namespaces, each representing a layer:

1. `Data`
2. `Domain`
3. `Presentation`

Code to save state to files (`data/FileBackend.cs`):

```
namespace Data
{
  public class FileBackend
  {
    private string filename;

    public FileBackend (string filename) {
      this.filename = filename;
    }

    public bool Store (List<string> entries) {
      try {
        StreamWriter writer = new StreamWriter(filename);

        foreach (string entry in entries) {
          writer.WriteLine(entry);
        }

        writer.Close();
      } catch (Exception e) {
        Console.WriteLine(e.Message);
        return false;
      }

      return true;
    }

    public List<string> Load () {
      List<string> entries = new List<string>();

      try {
        string? line;
        StreamReader reader = new StreamReader(filename);

        while ((line = reader.ReadLine()) != null) {
          entries.Add(line);
        }

        reader.Close();
```

```
      } catch (Exception e) {
        Console.WriteLine(e.Message);
        return new List<string>();
      }

      return entries;
    }
  }
}
```

Code for an item (`domain/Item.cs`):

```
namespace Domain
{
  public abstract class Item : IExpirable, IComparable<Item>
  {
    public string Name { get; set; }
    public double Price { get; set; }

    public Item (string name, double price) {
      this.Name = name;
      this.Price = price;
    }

    public abstract bool IsExpired ();

    public int CompareTo (Item? it) {
      // guard
      if (it==null) return -1;

      double tmp = Price - it.Price;
      if (tmp < 0) {
        return -1;
      } else if (tmp > 0) {
        return 1;
      } else {
        return 0;
      }
    }

    public abstract string GetItemType ();
    public abstract string[] GetState ();

    public string Marshal () {
      string type = GetItemType();
      string[] parameters = GetState();
      return type + " " + string.Join(" ", parameters);
    }

    public static Item? Unmarshal (string input) {
```

```
      string[] args = input.Split(' ');
      string   type = args[0];

      switch (type) {
        case "FoodItem":
          return FoodItem.Unmarshal(args);
        case "NonFoodItem":
          return NonFoodItem.Unmarshal(args);
        default:
          return null;
      }
    }
  }
}
```

Code for an edible item (`domain/FoodItem.cs`):

```
namespace Domain
{
  public class FoodItem : Item
  {
    private DateTime ExpiresAt;

    public static FoodItem? Unmarshal (string[] args) {
      if (args.Length != 4) return null;

      string name = args[1];
      double price = double.Parse(args[2]);
      DateTime date;

      try {
        date = DateTime.Parse(args[3]);
      } catch (FormatException) {
        return null;
      }

      try {
        return new FoodItem(name, price, date);
      } catch (ExpiredItemAddedException) {
        return null;
      }
    }

    public FoodItem (string name, double price, DateTime expiresAt)
        : base(name, price) {
      this.ExpiresAt = expiresAt;
      if (IsExpired()) {
        throw new ExpiredItemAddedException();
      }
    }
```

```
    public override string ToString () {
      return "FoodItem name='"+Name+"' price='"+Price+"' ExpiresAt='"+ExpiresAt+"'";
    }

    public override bool IsExpired () {
      return ExpiresAt.CompareTo(DateTime.Now) < 0;
    }

    public override string GetItemType () {
      return "FoodItem";
    }

    public override string[] GetState () {
      return new string[] { Name, Price.ToString(), ExpiresAt.ToString() };
    }

    // self-test
    public static void Test () {
      FoodItem?[] items = new FoodItem[10];

      for (int i=0 ; i<items.Length ; i++) {
        try {
          items[i] = new FoodItem("Item " + i, 12.3 * i, DateTime.Now.AddDays(i * 30));
        } catch (ExpiredItemAddedException) {
          items[i] = null;
        }
      }

      foreach (FoodItem? item in items) {
        Console.WriteLine(item);
      }
    }
  }
}
```

Code for a non-edible item (`domain/NonFoodItem.cs`):

```
namespace Domain
{
  public class NonFoodItem : Item
  {
    public List<string> Materials { get; set; }

    public static NonFoodItem? Unmarshal (string[] args) {
      if (args.Length != 4) return null;

      string name = args[1];
      double price = double.Parse(args[2]);
      string[] Materials = args[3].Split(',');
```

```
      return new NonFoodItem(name, price, Materials);
    }

    public NonFoodItem (string name, double price, List<string> materials)
        : base(name, price) {
      this.Materials = materials;
    }

    public NonFoodItem (string name, double price, string[] materials)
        : base(name, price) {
      this.Materials = new List<string>(materials);
    }

    public override bool IsExpired () {
      throw new NotSupportedException("Item does not support this operation.");
    }

    public override string ToString () {
      string m = "[";
      for (int i=0 ; i<Materials.Count ; i++) {
          m += (i == 0 ? "" : ",") + Materials[i];
      }
      m += "]";
      return "NonFoodItem name='"+Name+"' price='"+Price+"' Materials='"+m+"'";
    }

    public override string GetItemType () {
      return "NonFoodItem";
    }

    public override string[] GetState () {
      return new string[] { Name, ""+Price, string.Join(",", Materials) };
    }

    // self-test
    public static void Test () {
      NonFoodItem[] items = new NonFoodItem[10];

      for (int i=0 ; i<items.Length ; i++) {
        items[i] = new NonFoodItem("Item "+i, 12.3 * i,
                                   new string[] { "butter", "cream" });
      }

      foreach (NonFoodItem item in items) {
        Console.WriteLine(item);
      }
    }
  }
}
```

A comparator for product lengths (`domain/ItemNameLengthComparer.cs`):

```
namespace Domain
{
  class ItemNameLengthComparer : IComparer<Item>
  {
    public int Compare (Item? i1, Item? i2) {
      if (i1==null) return  1;
      if (i2==null) return -1;
      return i1.Name.Length - i2.Name.Length;
    }
  }
}
```

Interface for whether something has expired (`domain/IExpirable.cs`):

```
namespace Domain
{
  interface IExpirable
  {
    bool IsExpired ();
  }
}
```

Exception for an expired item (`domain/ExpiredItemAddedException.cs`):

```
namespace Domain
{
  public class ExpiredItemAddedException : Exception
  {
    public ExpiredItemAddedException () :
           base("Attempted to add expired product to database")
    {}
  }
}
```

Code for an inventory (`domain/Inventory.cs`):

```
using Data;

namespace Domain
{
  public class Inventory
  {
    private List<Item> items;

    public Inventory (List<Item> items) {
      this.items = items;
```

```
}

public Inventory () {
  this.items = new List<Item>();
}

public bool Load (string filename) {
  FileBackend fb = new FileBackend(filename);
  if (fb == null) return false;

  List<string> entries = fb.Load();
  if (entries == null) return false;

  foreach (string entry in entries) {
    Item? item = Item.Unmarshal(entry);
    if (item != null) AddItem(item);
  }

  return true;
}

public bool Store (string filename) {
  FileBackend fb = new FileBackend(filename);
  if (fb == null) return false;

  List<string> entries = new List<string>();
  foreach (Item item in items) {
    entries.Add(item.Marshal());
  }

  return fb.Store(entries);
}

public void AddItem (Item item) {
  if (!items.Contains(item)) {
    items.Add(item);
  }
}

public void RemoveItem (Item item) {
  items.Remove(item);
}

public double GetInventory () {
  double total = 0.0;

  foreach (Item item in items) {
    total += item.Price;
  }
```

```
    return total;
  }

  public void PrintInventory () {
    Console.WriteLine("Inventory:");
    foreach (Item item in items) {
      Console.WriteLine(" - " + item);
    }
  }

  public static void PrintStatus (Inventory inventory) {
    inventory.PrintInventory();
    Console.WriteLine("Total: " + inventory.GetInventory());
    Console.WriteLine("");
  }

  public void RemoveExpiredFoods () {
    for (int i=0 ; i<items.Count ; i++) {
      Item item = items[i];
      try {
        bool expired = item.IsExpired();
        if (expired) {
          items.Remove(item);
        }
      } catch (NotSupportedException e) {
        Console.WriteLine("Item "+item.Name+" does not support this operation: "+
                          e.Message);
        return;
      }
    }
  }

  public List<Item> SortedByNameLength () {
    List<Item> result = new List<Item>(items);
    result.Sort(new ItemNameLengthComparer());
    return result;
  }

  public List<Item> SortedByPrice () {
    List<Item> result = new List<Item>(items);
    result.Sort();
    return result;
  }

  public static FoodItem? SafeFoodItem (string name, double price, DateTime date) {
    try {
      return new FoodItem(name, price, date);
    } catch (ExpiredItemAddedException e) {
      Console.WriteLine("Item " + name + " is expired: " + e.Message);
      return null;
```

```
      }
    }

    // self-test
    public static void Test () {
      Inventory inventory = new Inventory();
      Item? i1 = SafeFoodItem("chocolate", 19.95, DateTime.Now.AddDays(1));
      Item? i2 = SafeFoodItem("coffee", 24.95, DateTime.Now.AddDays(1));
      Item? i3 = SafeFoodItem("Milk", 12.95, DateTime.Now.AddDays(1));
      Item? i4 = SafeFoodItem("Egg", 4.95, DateTime.Now.AddTicks(100));
      Item? i5 = new NonFoodItem("USB Charger", 17.45,
                                 new string[] { "plastic", "stuff" });
      Item?[] items = new Item?[] { i1, i2, i3, i4, i5 };

      PrintStatus(inventory);
      foreach (Item? item in items) {
        // guard: don't add null values
        if (item == null) continue;

        inventory.AddItem(item);
        Console.WriteLine("Adding " + item);
        PrintStatus(inventory);
      }

      Console.WriteLine("Sorting by name length:");
      foreach (Item? i in inventory.SortedByNameLength()) {
        Console.WriteLine(" - " + i.Name + " (" + i.Price + ")");
      }
      Console.WriteLine("");
      PrintStatus(inventory);

      Console.WriteLine("Sorting by price:");
      foreach (Item i in inventory.SortedByPrice()) {
        Console.WriteLine(" - " + i.Name + " (" + i.Price + ")");
      }
      Console.WriteLine("");
      PrintStatus(inventory);

      if (i1!=null) inventory.RemoveItem(i1);
      PrintStatus(inventory);

      inventory.RemoveExpiredFoods();
      PrintStatus(inventory);
    }
  }
}
```

Sample code (`presentation/Test.cs`):

```
using Domain;
```

```
namespace Presentation
{
  class Test
  {
    public static void Add () {
      Item i1 = new NonFoodItem("USB_Charger", 17.45,
                                new string[] { "plastic", "stuff" });

      Inventory inventory = new Inventory();
      inventory.Load("inventory.txt");
      inventory.AddItem(i1);
      inventory.Store("inventory.txt");
    }

    public static void List () {
      Inventory inventory = new Inventory();
      inventory.Load("inventory.txt");
      inventory.PrintInventory();
    }

    public static void Main (string[] args) {
      Add();
      List();
    }
  }
}
```

Chapter 26

Paradigms

Hello

26.1 Exercises

Exercise 26.1: Livebook Installation

In this exercise, you will install Livebook[1] on your laptop. The installation instructions are here: https://livebook.dev/#install.

If you are on Windows or OSX, you can install *Livebook Desktop*. If you are on Linux, you can install via Docker. Note that there are different ways to start Livebook through docker. Choose one that suits your preferences.

Once Livebook has been started, it exposes a web interface. As part of the startup process, it prints out the URL for this interface, and it is through this that we will work with Livebook. Point a browser to this URL and click under "Settings". Under "Code editor" you will find a number of options, including:

- ***Show completion list while typing*** Make sure this is enabled.
- ***Show function signature while typing*** Make sure this is enabled.
- ***Render ligatures*** Make sure this is enabled.
- ***Wrap words in Markdown*** Make sure this is enabled.

Exercise 26.2: Elixir Introduction

First, check out the following repository: https://github.com/aslakjohansen/elixir-intro. Here, under `start.livemd`, you will find a guided tour (with tasks) through Elixir. Open this file in Livebook and solve the beginner tasks.

Exercise 26.3: Demo Repository

[1] https://livebook.dev

Topic
Creativity

First, you need to check out the following repository: https://github.com/aslakjohansen/livebook-demos. It contains some demo code that can be run through Livebook.

The exercise is to:

1. Open your installation of Livebook in a browser.
2. Click "Open" and select `index.livemd` from your checkout of the above repository.
3. Hover your mouse over the bottom cell and click "Evaluate". What you see is an index of the demos available in this repository. You can open these in new tabs.
4. Spend some time (e.g. 15 min) clicking through the following demos:
   - Basic data types
   - Control structures
   - Pattern matching
   - Pipelines

Exercise 26.4: Filtering of Even Numbers

Topic
Creativity

Using recursion, implement a function (in a module) in Elixir that, given a list of integers, produces a new list of the numbers in the input list that are even.

Exercise 26.5: Elementary Equality

Topic
Creativity

Using recursion, implement a function (in a module) in Elixir that, given two lists (`a` and `b`), produces a list whose ith element is a boolean value that is true if the ith element of `a` is equal to the ith element of `b`.

Note: In this exercise we assume that the lists in `a` and `b` are of equal length.

Exercise 26.6: The Fibonacci Sequence

Topic
Creativity

The Fibonacci sequence is an infinite sequence of integers determined by the following formula:

$$\text{fib}(n) = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{for } n > 1 \end{cases}$$

This sequence can be observed in a number of places in nature and is also used in a number of data structures.

Subtask 1

Create a new notebook in Livebook and in an Elixir code cell create a module named `FibPrimitive`. Inside this module, you now create three functions called `calc`:

1. The first takes the value 0 as a parameter and evaluates to the value 0.

2. The second takes the value 1 as a parameter and evaluates to the value 1.
3. The third takes the value n as a parameter and evaluates to the sum of the result of a recursive call with the parameter $n - 1$ and the result of another recursive call with the parameter $n - 2$. Add a guard that checks that n is positive.

Test by creating a new Elixir code cell and entering the following code:

```
0..40
|> Enum.map(&FibPrimitive.calc/1)
```

This should yield:

```
iex(1)> 0..40|> Enum.map(&FibPrimitive.calc/1)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229,
 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817,
 39088169, 63245986, 102334155]
```

Subtask 2

Unfortunately, it takes a significant amount of time to calculate these first 40 numbers in the sequence. Why that is?

Subtask 3

Let's try to implement an alternative definition of the sequence. Here, we work with the last two values and the number of remaining iterations. As a starting point we use the first two numbers in the sequence; that is, 0 and 1. In the following, r_n stands for the nth result in the sequence.

The new definition:

$$\text{fib}(n) = \text{fib}_2(0, 1, n)$$

$$\text{fib}_2(r_{-1}, r_{-2}, n) = \begin{cases} r_{-1} & \text{for } n = 0 \\ \text{fib}_2(r_{-1} + r_{-2}, r_{-1}, n - 1) & \text{for } n > 0 \end{cases}$$

Create a new Elixir code cell with a module named `FibThoughtfull`. Inside this module you now create functions called `calc` for the above definitions.

Verify that the code is working correctly. How fast is it compared to the last version and why?

Subtask 4

Lets try to visualize that differense.

Add the latest version of the following modules to your workbooks *dependencies*:

- `:vega_lite`
- `:kino_vega_lite`

Then copy the following code into your notebook in appropriate cells:

```elixir
# convenience alias
alias VegaLite, as: Vl

# module for timing
defmodule Timer do
  def measure(f, inputs, label) do
    inputs
    |> Enum.map(
      fn input ->
        {time, _} = :timer.tc(f, [input])
        %{
          "n" => input,
          "time" => time,
          "implementation" => label,
        }
      end
    )
  end
end

# perform timing measurements
inputs = 0..40
measurements_primitive = Timer.measure(&FibPrimitive.calc/1, inputs, "Primitive")
measurements_thoughtfull = Timer.measure(&FibThoughtfull.calc/1, inputs, "Thoughtfull")
measurements = measurements_primitive ++ measurements_thoughtfull

# plot results
Vl.new(width: 400, height: 300)
|> Vl.data_from_values(measurements)
|> Vl.mark(:line)
|> Vl.encode_field(:x, "n", type: :quantitative, title: "n/[n]")
|> Vl.encode_field(:y, "time", type: :quantitative, title: "Time/[us]")
|> Vl.encode_field(:color, "implementation", type: :nominal, title: "Implementation")
```

Run these cells.

Which solution would you choose for $n = 50$?

Exercise 26.7: Pipelines

Topic
Creativity

Let's consider the following C# program:

```
Human[] humans = new Human[] {
  new Human {sex=Sex.Female, age=17, name="Buffy Summers"},
  new Human {sex=Sex.Male, age=12, name="Luca"},
  new Human {sex=Sex.Female, age=22, name="Sofie"},
  new Human {sex=Sex.Male, age=34, name="Klaus"},
  new Human {sex=Sex.Female, age=19, name="Perinne"},
  new Human {sex=Sex.Female, age=23, name="Fie"},
  new Human {sex=Sex.Male, age=20, name="Asger"},
  new Human {sex=Sex.Female, age=27, name="Pernille"},
  new Human {sex=Sex.Male, age=21, name="Peter"},
  new Human {sex=Sex.Female, age=26, name="Dominika"},
  new Human {sex=Sex.Male, age=7, name="Mathias"},
```

```
};

int minAge (Human[] humans) {
  int best = -1;

  foreach (Human human in humans) {
    if (human.sex==Sex.Male && human.age>=18 && (best==-1 || human.age<best)) {
      best = human.age;
    }
  }

  return best;
}

Console.WriteLine(minAge(humans));

enum Sex {
  Female,
  Male,
};

struct Human {
  public Sex sex;
  public int age;
  public string name;
}
```

It defines a variety of people, and implements a function to calculate the age of the youngest male who is 18 or older.

The Elixir equivalent of the struct definition and input data looks like this:

```
defmodule Human do
  defstruct sex: :female, age: 0, name: "Buffy Summers"
end

humans =
  [
    %Human{age: 17},
    %Human{sex: :male, age: 12, name: "Luca"},
    %Human{sex: :female, age: 22, name: "Sofie"},
    %Human{sex: :male, age: 34, name: "Klaus"},
    %Human{sex: :female, age: 19, name: "Perinne"},
    %Human{sex: :female, age: 23, name: "Fie"},
    %Human{sex: :male, age: 20, name: "Asger"},
    %Human{sex: :female, age: 27, name: "Pernile"},
    %Human{sex: :male, age: 21, name: "Peter"},
    %Human{sex: :female, age: 26, name: "Dominika"},
    %Human{sex: :male, age: 7, name: "Mathias"},
  ]
```

Subtask 1

Use recursion to implement code in Elixir via Livebook that is functionally equivalent to the C# function.

Subtask 2

Now instead use higher-order functions from the standard library[2] to implement functionally equivalent code in Elixir via Livebook.

Subtask 3

Discuss which implementation is better and why?

Exercise 26.8: Higher-Order Functions

Topic
Creativity

Elixir has an `Enum` module that offers, among other things, a `map` and a `filter` function. In this exercise, you will write these functions from scratch in a module that you call `HigherOrder`.

Exercise 26.9: C♭

Topic
Creativity

This exercise deals with the language C♭ (pronounced C flat). This (programming) language is characterized by being remarkably similar to the first C# you wrote.

At https://github.com/aslakjohansen/cflat/blob/main/examples/cflat_demo.livemd you will find a demo. Open this in Livebook, either by downloading the file, or by checking out the repository.

Subtask 1

In this exercise, you must paste a few of your solutions to the exercises from before you learned about arrays into the text field in this notebook, and then click through the remaining cells. Discuss why the abstract syntax tree looks the way it does.

Subtask 2

Now go to the above repository and find the code for the interpreter[3]. How is a `while` loop evaluated?

Exercise 26.10: Airports

Topic
Creativity

The following link contains a list of airports in the world, with country code and coordinates:
https://raw.githubusercontent.com/ip2location/ip2location-iata-icao/master/iata-icao.csv

Subtask 1

Copy the following code into a Livebook notebook (use multiple cells):

```
alias MapLibre, as: Ml

# List of countries of various categories
# source: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Country_codes
countries_eu =
  "BE,BG,CZ,DK,DE,EE,IE,EL,ES,FR,GR,HR,IT,CY,LV,LT,LU,HU,MT,NL,AT,PL,PT,RO,DI,DK,FI,SE"
  |> String.split(",")
countries_efta =
  "IS,LI,NO,CH"
```

[2] For example, those from the `Enum` and `List` modules.
[3] https://github.com/aslakjohansen/cflat/blob/main/lib/cflat/interpreter.ex

```
  |> String.split(",")
countries_eu_cand =
  "BA,ME,MD,MK,GE,AL,RS,TR,UA"
  |> String.split(",")
countries_enp =
  "AM,BY,AZ,DZ,EG,IL,JO,LB,LY,MA,PS,SY,TN"
  |> String.split(",")
countries_other =
  "AR,AU,BR,CA,CN,CM_X_HK,HK,IN,JP,MX,NG,NZ,RU,SG,ZA,KR,TW,UK,US"
  |> String.split(",")
countries_interest =
  [
    countries_eu,
    countries_efta,
    countries_eu_cand,
    countries_enp,
    countries_other
  ]
  |> List.flatten()

# data source
url = "https://raw.githubusercontent.com/ip2location/ip2location-iata-icao/master/iata-icao.csv"

# fetch data and split it into lines
file =
  url
  |> Req.get!()
  |> Map.get(:body)
  |> String.replace("\",\"", "\"|\"", global: true)
  |> String.replace("\"", "", global: true)
  |> String.split("\r\n")

# split lines into header and data lines
[header_line|contents_lines] = file

# parse the header into a name-to-index map
index =
  header_line
  |> String.split("|")
  |> Enum.with_index()
  |> Map.new()

# parse data lines according to header map to obtain a list of airports
airports =
  contents_lines
  |> Enum.filter(fn line -> line != "" end)
  |> Enum.map(
    fn line ->
      elements =
        line
        |> String.split("|")
      index
      |> Enum.reduce(%{},
        fn {key, i}, acc ->
          Map.put(acc, key,
            case {key, Enum.at(elements, i)} do
              {"latitude", value} -> Float.parse(value) |> elem(0)
              {"longitude", value} -> Float.parse(value) |> elem(0)
              {_, value} -> value
            end
          )
        end
      )
```

```
    end
  )

# define list of markers to place on map
markers =
  airports
  |> Enum.map(
    fn %{"latitude" => lat, "longitude" => long, "airport" => _name, "country_code" => country} ->
      [
        {long, lat},
        scale: 0.5,
        color:
          cond do
            country=="DK" -> "red"
            true -> "black"
          end
      ]
    end
  )

# produce map
Ml.new()
|> Kino.MapLibre.add_markers(markers)
```

Three external modules are used:

- `:req` to retrieve things from the internet.
- `:maplibre` to work with maps.
- `:kino_maplibre` to display maps.

These modules must be added at the top under "Notebook dependencies and setup". Select the latest versions of these modules.

You should now be able to click through the notebook and get a world map with a bunch of pins. You can zoom in in this map.

Subtask 2

What does the function `Req.get!` do?

Subtask 3

What is the role of `index`?

Subtask 4

The module `Enum` has a `member?` function that tells whether a term is a member of a list. The same module has a `filter` function. Use a combination of these to ensure that only the airports located in a city from a country mentioned in `countries_interest` are shown.

Subtask 5

Now we need to color the markers according to the following rules:

- Members of `countries_eu` should be colored `"blue"`.
- Members of `countries_efta` should be colored `"cyan"`.

- Members of `countries_eu_cand` should be colored `"green"`.
- Members of `countries_enp` should be colored `"orange"`.
- Members of `countries_other` should be colored `"white"`.

Verify that it works!

Subtask 6

Why is "Heathrow Airport" not on the final map?

Exercise 26.11: World of Zuul

Find the Livebook overview of demos from exercise 26.3. Here you will find and open "World of Zuul". Try to get an overview of the code by clicking through it.

The exercise is:

1. Explain how the data structure that the variable `spaces` refers to affects the actors/processes that are started?
2. How does the `go` command work, and what determines where you can *go*?
3. Try opening a new tab in your browser, and point it to exactly the same URL. Scroll down to the bottom and choose a new name. What is needed for the two users to talk to each other, and how is this implemented?

Part V

Back End

Appendix A

Exercise Answers

A.1 Background

Solution for exercise 2.1: Terms

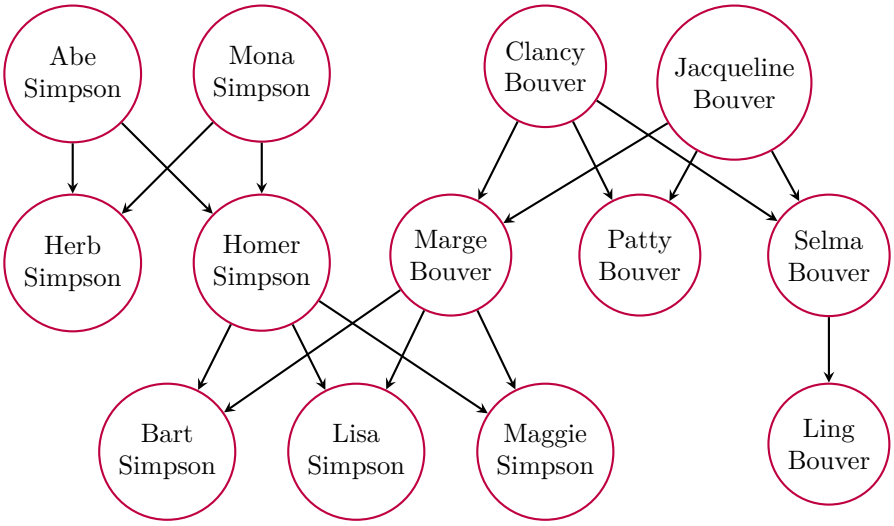
1. This is a directed and weighted graph.
2. There are nine nodes.
3. There are 11 edges.
4. n_9, n_8, n_5 and n_6 make a graph of 4 edges.
5. Due to the direction of the edge from n_2 to n_9, there is no path from n_8 to n_4. Had this path been reversed, the total distance would be $1.4 + 1.6 + 2.4 + 1.5 + 1.3 = 8.2$.

Solution for exercise 2.2: Tree Structures

- ***a*** has a cycle and is thus not a tree.
- ***b*** is connected, and has $n = 9$ nodes and exactly $n - 1$ unique edges. It is thus a tree.
- ***c*** is connected, and has $n = 6$ nodes and exactly $n - 1$ unique edges. It is thus a tree.
- ***d*** is connected, and has $n = 7$ nodes and exactly $n - 1$ unique edges. It is thus a tree.
- ***e*** is connected, and has $n = 9$ nodes and 9 unique edges (i.e., $\neq n - 1$). It is thus not a tree.
- ***f*** consists of two subgraphs which each is a tree. A graph that consists of all trees is called a forest.

Solution for exercise 2.3: Ancestry

A (partial) graph of the ancestry of the Simpsons could be constructed like this:



In this example, the following graph-related terms are in use:

- A node is used to represent a person.
- An edge is used to represent an parent-child relation. That is, it goes from a parent to a child.

Solution for exercise 2.4: Text Editor

This exercise does not have a reference answer.

Solution for exercise 2.5: File Creation

This exercise does not have a reference answer.

Solution for exercise 2.6: Directory

The relevant commands are `mkdir` and `mv`.

Solution for exercise 2.7: GitHub Account

There is no reference solution for this exercise

Solution for exercise 2.8: Git Installation

There is no reference solution for this exercise

Solution for exercise 2.9: Git Setup

There is no reference solution for this exercise

Solution for exercise 2.10: Repository Creation

There is no reference solution for this exercise

Solution for exercise 2.11: Markdown

Github will render an interpretation of the text that highlights according to the markup.

Solution for exercise 2.12: Commit Log

The following information is listed for every commit:

- Commit ID.
- Author with name and email.
- Commit date (not push date).
- Comment.

Solution for exercise 2.13: Changes

This exercise does not have a reference solution.

Solution for exercise 2.14: Conflicts

This exercise does not have a reference solution.

Solution for exercise 2.15: Repository Structure

Subtask 1

Every exercise belongs to a theme, and there is a chapter for every theme. It would seem that every theme have multiple exercises associated with them.

Subtask 2

We can follow the same structure by creating a directory for each theme, and inside these create directories for each exercise. These last directories can contain the solutions to the exercises.

A.2 The First Program

Solution for exercise 4.1: Framework

This exercise does not have a reference answer.

Solution for exercise 4.2: Framework Version

The `project.csproj` file contains the following lines (or very similar):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
```

```
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

The highlighted line specifies that version 8.0 of the .NET framework should be used. Here, `net` is used as a shorthand for .NET.

Solution for exercise 4.3: Hello World

See chapter text.

A.3 Primitive Types

Solution for exercise 5.1: Temperature

A temperature is a number, and temperatures are not limited to integral numbers. `float` and `double` are thus natural choices.

Solution for exercise 5.2: Month

There are 12 different months. Often 1 is said to represent January … It is unclear what 1.1 would mean then, so limiting ourselves to integral values we can make sure that we never have to find out. For a similar reason, we might as well look at the *unsigned* variants. As we only have 12 months, an `int` is more space efficient than a `long`. An `sbyte` is even smaller (and has no problem representing 12 values). It is, however, slower to work with on modern computers.

Solution for exercise 5.3: Limits of Integers

In C#, there are 4 (primitive, signed and portable) integral types: `sbyte`, `short`, `int` and `long`.

An `sbyte` can represent any integral number from (and including) -128 until (and also including) 127. To find out what happens when we exceed the largest number it can represent, we writw the following program:

```
sbyte b = sbyte.MaxValue;
Console.WriteLine("byte: " + b);
b += 1;
Console.WriteLine("byte: " + b);
```

When executed, it prints:

```
127
-128
```

Accordingly, 127+1 turns out to evaluate to −128. In other words; when we add one to the largest number we an represent we get the lowest number we can represent. This is not a coincidence, and holds for all the primitive integer types.

Solution for exercise 5.4: Casting

We are always allowed to cast explicitly, but this is considered a bad practice as it is unnecessary and distracting. The following code shows the minimal amount of explicit casting:

```
int i = 127;
Console.WriteLine("int i = "+i);

long l = i;
Console.WriteLine("long l = "+l);

i = (int)l;
Console.WriteLine("i (cast from long to int) = "+i);

float f = 3.14F;
Console.WriteLine("float f = "+f);

double d = f;
Console.WriteLine("double d = "+d);

f = (float)d;
Console.WriteLine("f (cast from double to float) = "+f);

f = i;
Console.WriteLine("f (cast from int i to long) = "+f);

i = (int)f;
Console.WriteLine("i (cast from float to double) = "+i);
```

As the typechecker executes at compiletime it (generally) cannot take concrete values into account. It thus has to work at the type-level, and act according to a worst-case scenario.

There is no guarantee that a `long` value can *fit* into the smaller size of an `int` variable, or similarly, that a `double` value can fit into a `float` variable. Naturally, a floating point number cannot fit into an integer number.

The rule is that an explicit cast is necessary if the operation has the potential to result in a loss of data.

Solution for exercise 5.5: Definition of Expression

An *expression* is a continuous piece of code that evaluates to a value.

Solution for exercise 5.6: Assignment

Try this:

```
int var1;
int var2 = var1 = 42;
Console.WriteLine(var1);
Console.WriteLine(var2);
```

Solution for exercise 5.7: Expression vs Statement

Expressions and statements are different syntactic constructs. The C# compiler expects expressions at some places in the code and statements at other. If your program does not adhere to these *rules* then the compiler will give a syntax error and give up.

The most important difference is that an expression – as opposed to a statement – evaluates to a value. The main structure of logic is made up of statements, and expressions are used to describe intermediate results within specific statements.

Solution for exercise 5.8: Areas of Circles

```
const double pi = 3.14;
double r;

r = 1;
Console.WriteLine("r=" + r + " -> area=" + (pi * r * r));

r = 3;
Console.WriteLine("r=" + r + " -> area=" + (pi * r * r));

r = 5;
Console.WriteLine("r=" + r + " -> area=" + (pi * r * r));
```

Solution for exercise 5.9: Summed Circumference of Circles

```
const double pi = 3.14;
double r, c;
double s = 0.0;

r = 1;
c = 2.0 * pi * r;
s += c;
Console.WriteLine("r=" + r + " -> circumference=" + c);

r = 3;
c = 2.0 * pi * r;
s += c;
Console.WriteLine("r=" + r + " -> circumference=" + c);

r = 5;
c = 2.0 * pi * r;
s += c;
Console.WriteLine("r=" + r + " -> circumference=" + c);

Console.WriteLine("Sum is " + s);
```

Solution for exercise 5.10: Celcius to Fahrenheit

```
double celcius = 21.7;
double fahrenheit = celcius * 9 / 5 + 32;
Console.WriteLine(celcius + "°C -> " + fahrenheit + "°F");
```

Solution for exercise 5.11: Epoch

```
const int secsPerDay = 24 * 60 * 60;
const int secsPerYear = 365 * secsPerDay;

long epoch = 99023040;

long years = epoch / secsPerYear;
long days = (epoch - years * secsPerYear) / secsPerDay;

Console.WriteLine(epoch + " seconds -> " + years + " years and " + days + " days");
```

Solution for exercise 5.12: Incrementing a Month

Observation: I used 9 as the initial value. This value did not change as I added 0.5, and still remained the same as I did it again.

Explanation: When adding values of different types, C# performs a conversion to the type on the right-hand side of the operator. As this type is integral, the value 0.5 will be rounded to zero. So, in reality we are adding zero to nine …twice.

Solution for exercise 5.13: Value vs Variable

A variable is a symbolic reference (or indirection) to a value. We say that the variable *refers* to the value. We can change which value the variable refers to by assigning it a new value. In reality the value is placed either in the computers memory, or a register. The variable will refer to this location (memory address or register number).

This is an abstraction that makes it possible to write code that is independent of specific values.

Solution for exercise 5.14: Daily Differences

```
double day1 = 21.5;
double day2 = 23.7;
double day3 = 19.6;
double day4 = 22.5;
double day5 = 25.3;
double day6 = 21.7;
double day7 = 18.9;

Console.WriteLine((day2 - day1) + "°C");
Console.WriteLine((day3 - day2) + "°C");
Console.WriteLine((day4 - day3) + "°C");
Console.WriteLine((day5 - day4) + "°C");
Console.WriteLine((day6 - day5) + "°C");
Console.WriteLine((day7 - day6) + "°C");
```

It does the calculation wrongly because of the ways floating-point numbers are represented in a computer (throuh IEEE floats). This method is not exact, and will often produce small errors. There are more preceise ways of representing floating-point numbers (e.g., using fractions of integers), but they are quite a bit slower.

Solution for exercise 5.15: Average Age

When executed the program prints out the following:

```
Average livespan of a male computer scientist: 76.666664
Average livespan of a female computer scientist: 68.666664
Average livespan of a computer scientist: 72.666664
Males lives this much longer than female: 8.0
```

The program consists of (i) 6 variable declarations with initial assignment of values, (ii) a number of variable declarations where each variable is initialized to hold the result of some calculations, and (iii) printouts of the resulting values of these variable.

In the first part, six `int` variables are declared. Each of these are named after a well-known computer scientist, and is initialized to their lifespan in years. For each of these a commented out link to their page on Wikipedia is provided.

Next comes four calculations, each of which is stored in a `float` variable.

In the first calculation, all the "male" variables are summed up. The result of this is cast to a `float` and then divided by three (i.e., the number of people). Finally the result is stored in the `male_avg` variable. In similar fashion, the average for the female computer scientists is calculated and stored in the `female_avg` variable. In the third calculation, these two averages are summed, then divided by two, and finally stored in the `avg` variable. This is correct as the number of males and females are the same. As `male_avg` is a `float`, it is not necessary to cast in order to avoid a potential rounding error in the division. In the last calculation, `female_avg` is subtracted from `male_avg` and the result is stored in the variable `diff`.

Finally, each of these 4 variables are printed to the screen. To do this, two statements are used for each. In the first statement, a string that describes what the contents of the variable represents is printed without newline. In the second and last statement, the value is printed with newline.

Solution for exercise 5.16: Printf

The following happens:

- `i` is printed normally.
- `l` is padded to a width of four and printed.
- `f` is printed normally.
- `d` is padded to a width of six with two decimals and printed.p

This is useful for making sure that commas are aligned across multiple lines.

Solution for exercise 5.17: Definition of Boolean

A `boolean` can either refer to a type, or a variable of this type. In the latter case, the variable can have one of two values; namely true and false. This is a single bit of information, but is typically assigned more memory than that. A number of *boolean* operators are defined on this type. They are used to implement boolean logic.

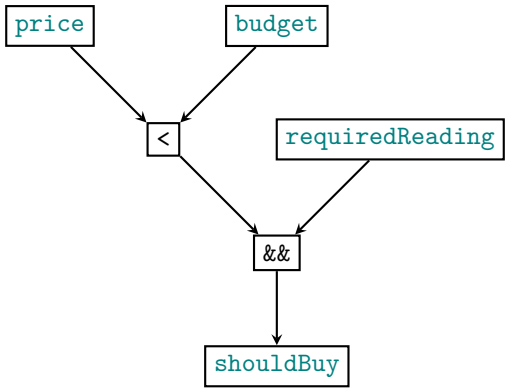
Solution for exercise 5.18: Creation of a Boolean

Boolean values are produced:

1. When the `true` or `false` literal is used.
2. As a result of a comparison.
3. As result of a boolean operation.

Solution for exercise 5.19: Decision of Purchase

The calculation is done according to the parse tree:



The variable `shouldBuy` represents the outcome of the decision of whether to purchase a book. The book should be purchased when then price is below budget, and the book is required reading.

Solution for exercise 5.20: Dice

```
int dice = 6;

bool test = (dice & 1) != 1 && dice > 3;
Console.WriteLine(test);
```

Comments:

- `dice & 1`: This expression is a bitwise AND operation between `dice` and `1`. If `dice` is even then the result will be zero (as the least significant bit of an even number is zero).
- `dice > 3`: This expression checks wheter the value of `dice` is larger than three.

Solution for exercise 5.21: Manual Type Inference

Let's work our way backwards:

1. We know that `b` is an `int`, and that `1` is an integer type.
2. This means that C#must allow an implicit cast of any result of "`a + 1`" to `int`.
3. The `+` operator is defined on a number of types:
   - If any of the operands is a floating point type, then the result of the operation is a value of a floating point type. No floating point type can be implicitly cast to an `int`. So, `a` is not a variable of a floating point type.
   - If both operands are integer types and at least one of them is signed, then the result is of a signed integer type. But since `b` is unsigned, we know that `a` is unsigned as well.
   - If both operands are integer types then the resulting value will have an integer type. That type will match the largest of the operand types. This restricts the type of `a` to integers of a size that is no larger than an `int`. Those are `sbyte`, `short` and `int`.

So, `a` must have one of the following types: `sbyte`, `short` and `int`.

A.4 Flow Control

Solution for exercise 6.1: Epoch

According to Danish culture, it's Christmas on December 24th.

```
const int secsPerDay = 24 * 60 * 60;
const int secsPerYear = 365 * secsPerDay;

long epoch = (11 * 30 + 23) * 24 * 60 * 60;

long years = epoch / secsPerYear;
long days = (epoch - years * secsPerYear) / secsPerDay;
long month = days / 30;
long day = days % 30;

if (month == 11 && day == 23) {
  Console.WriteLine("It is Christmas!");
}
```

Solution for exercise 6.2: Epoch Diff

```
const int secsPerDay = 24 * 60 * 60;
const int secsPerYear = 365 * secsPerDay;
const int xmas = (11 * 30 + 23) * 24 * 60 * 60;

long epoch = xmas + secsPerDay; // 1 day after xmas.
```

```
long years = epoch / secsPerYear;
long days = (epoch - years * secsPerYear) / secsPerDay;
long month = days / 30;
long day = days % 30;

if (month == 11 && day == 23) {
  Console.WriteLine("It is Christmas!");
} else {
  long daysLeft = (xmas + secsPerYear - epoch) % secsPerYear / secsPerDay;
  Console.WriteLine("It is Christmas in " + daysLeft + " days");
}
```

Solution for exercise 6.3: Christmas Sale

```
const int secsPerDay = 24 * 60 * 60;
const int secsPerYear = 360 * secsPerDay;
const int xmas = (11 * 30 + 23) * 24 * 60 * 60;

// long epoch = xmas - secsPerDay;
// long epoch = xmas + secsPerDay;
long epoch = xmas;

long years = epoch / secsPerYear;
long days = (epoch - years * secsPerYear) / secsPerDay;
long month = days / 30;
long day = days % 30;

double price = 599.95 * (month == 11 && day == 23 ? 0.7 : 1.0);

Console.WriteLine(price);
```

Solution for exercise 6.4: Length of Month

```
int month = 6;
int length = -1;

if (month == 1 || month == 3 || month == 5 || month == 7 || month == 8 ||
    month == 10 || month == 12) {
  length = 31;
} else if (month == 2) {
  length = 28;
} else if (month == 4 || month == 6 || month == 9 || month == 11) {
  length = 30;
}

if (length == -1) {
  Console.WriteLine("Error: Month \"" + month + "\" is outside of [1,12]");
```

```
} else {
  Console.WriteLine(length);
}
```

Carl Coder, however, recommends:

```
int month = 6;
int length = -1;

length = (month == 1 || month == 3 || month == 5 || month == 7 || month == 8 ||
          month == 10 || month == 12 ? 31 : 0)
       + (month == 2 ? 28 : 0)
       + (month == 4 || month == 6 || month == 9 || month == 11 ? 30 : 0);

if (length == -1) {
  Console.WriteLine("Error: Month \"" + month + "\" is outside of [1,12]");
} else {
  Console.WriteLine(length);
}
```

Solution for exercise 6.5: Holidays

A typical implementation:

```
const int JANUARY = 0;
const int FEBRUARY = 1;
const int MARCH = 2;
const int APRIL = 3;
const int MAY = 4;
const int JUNE = 5;
const int JULY = 6;
const int AUGUST = 7;
const int SEPTEMBER = 8;
const int OCTOBER = 9;
const int NOVEMBER = 10;
const int DECEMBER = 11;

// choose a test case
int month = OCTOBER;
// int month = NOVEMBER;

switch (month)
{
    case OCTOBER:
        Console.WriteLine("Autumn Holiday");
        break;
    case DECEMBER:
        Console.WriteLine("Christmas Holiday");
        break;
```

```
    case APRIL:
        Console.WriteLine("Spring Holiday");
        break;
    case AUGUST:
        Console.WriteLine("Summer Holiday");
        break;
    default:
        Console.WriteLine("Hard work");
        break;
}
```

Carl Coders version:

```
const int JANUARY = 0;
const int FEBRUARY = 1;
const int MARCH = 2;
const int APRIL = 3;
const int MAY = 4;
const int JUNE = 5;
const int JULY = 6;
const int AUGUST = 7;
const int SEPTEMBER = 8;
const int OCTOBER = 9;
const int NOVEMBER = 10;
const int DECEMBER = 11;

// choose a test case
int month = OCTOBER;
// int month = NOVEMBER;

string message = (month == OCTOBER ? "Autumn Holiday"
               : (month == DECEMBER ? "Christmas Holiday"
               : (month == APRIL ? "Spring Holiday"
               : (month == JULY || month == AUGUST ? "Summer Holiday"
               : "Hard work"))));
Console.WriteLine(message);
```

Solution for exercise 6.6: Celcius to Fahrenheit

```
for (double c=-5 ; c<=40 ; c+=0.5) {
  double f = c*9/5 + 32;
  Console.WriteLine("{0,5:F1}°C {1,5:F1}°F", c, f);
}
```

Solution for exercise 6.7: Celcius to Fahrenheit in Reverse

```
for (double c=40 ; c>=-5 ; c-=0.5) {
  double f = c*9/5 + 32;
```

```
  Console.WriteLine("{0,5:F1}°C {1,5:F1}°F", c, f);
}
```

Solution for exercise 6.8: Celcius to Fahrenheit Alternatives

Implementation based on `while`:

```
double c = -5;

while (c <= 40) {
  double f = c*9/5 + 32;

  Console.WriteLine("{0,5:F1}°C {1,5:F1}°F", c, f);

  c += 0.5;
}
```

Implementation based on `do-while`:

```
double c = -5;

do {
  double f = c*9/5 + 32;

  Console.WriteLine("{0,5:F1}°C {1,5:F1}°F", c, f);

  c += 0.5;
} while (c <= 40);
```

Solution for exercise 6.9: Areas of Circles

```
const double pi = 3.14;

for (double r=1 ; r<=5 ; r+=2) {
  Console.Write("r=");
  Console.Write(r);
  Console.Write(" -> area=");
  Console.WriteLine(pi * r * r);
}
```

Solution for exercise 6.10: Primes

```
int last = 1000000;
int max = -1;

// start at 2, the first prime number
for (int i=2 ; i<last ; i++) {
```

```
  bool isPrime = true;

  // skip even numbers greater than 2
  if (i>2 && (i&1) == 0)
    continue;

  // only check up to the square root of i
  for (int j=3 ; j*j<=i ; j+=2) {
    if (i%j == 0) {
      isPrime = false;
      break;
    }
  }

  if (isPrime) {
    // Console.WriteLine(i);
    max = i;
  }
}

Console.WriteLine("Highest prime is " + max);
```

A.5 Basic Datastructures

Solution for exercise 7.1: Definition of Array

An array is a *datastruktur* in which a static number of elements are placed contiguously in memory. Lookups are efficient as all elements have the same size. Behind the scene C# does something like this:

$$A_{\text{element}} = A_{\text{array}} + \text{index} * \textbf{sizeof}(\textbf{typeof}(\text{element})) \tag{A.1}$$

Solution for exercise 7.2: Array Usecase

Arrays solves the problem where all integers under a certain number need to be associated (or mapped) to an individual value.

A common variant of this scenario is that all integers *between* to numbers need to be mapped to an individual value. Here, one often chooses to shift all elements by subtracting the index of the lower bound from all index operations. This way, we are not wasting memory. It will become a bit more expensive (in terms of time) to use the array though. This is a tradeoff.

Solution for exercise 7.3: Array Type

All elements in an array must have the same type. For instance, if one wishes to place data of the type `int` inside an array, then that array must have the type `int[]`. Otherwise, you will get a compile.

Note: Later in this book, we will see how we – to some extend – can work around this rule by using object-oriented constructs (not that this ability is unique to object-oriented languages).

Solution for exercise 7.4: Largest in Array

If we have some sort of guarantee that there only exists a single instance of the largest value, then the following will work:

```
int[] a = {1, 7, 4, 1, 23, 54, 8, 34, 54, 12, 11, 8, 25};

int max  = a[0];
int maxi = 0;

for (int i=0 ; i<a.Length ; i++) {
  int value = a[i];
  if (value > max) {
    max  = value;
    maxi = i;
  }
}

Console.WriteLine(maxi);
```

Otherwise, we should do something like this:

```
int[] a = {1, 7, 4, 1, 23, 54, 8, 34, 54, 12, 11, 8, 25};

// find max
int max = a[0];
foreach (int value in a) {
  if (value > max) {
    max = value;
  }
}

// locate occurrences and output them
for (int i=0; i<a.Length ; i++) {
  int value = a[i];
  if (value==max) {
    Console.WriteLine(i);
  }
}
```

Solution for exercise 7.5: Size of Array Allocation

When we create an array without explicitly defining its contents, we need to tell C#how large this array needs to be. Otherwise, how could it know how much memory to allocate? In C#, this is one using an expression that must evaluate to an integer. This means that the size can be expressed using a constant value, the contents of a variable, or the result of a calculation.

Solution for exercise 7.6: Multiplication Table

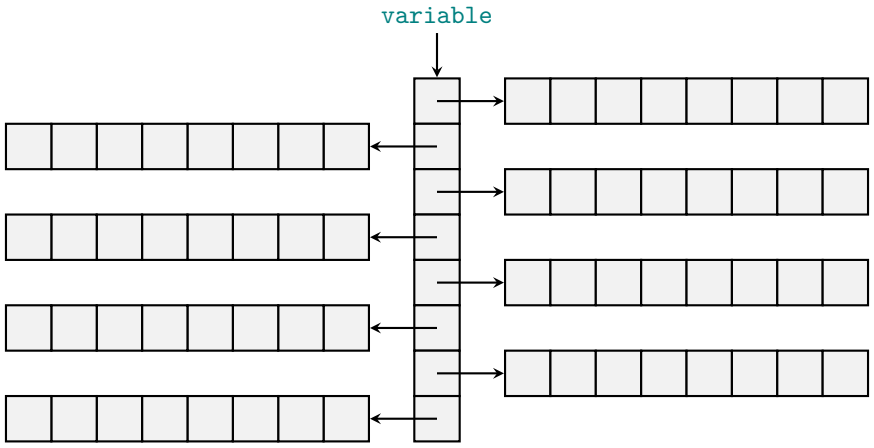
```
const int multiplier = 3;
const int count = 10;

// allocate memory
int[] data = new int[count];

// populate array
for (int i=0 ; i<data.Length ; i++) {
  data[i] = i*multiplier;
}

// perform test
int test1 = 0;
int test2 = count - 1;
Console.WriteLine(test1+" * "+multiplier+" = "+data[test1]);
Console.WriteLine(test2+" * "+multiplier+" = "+data[test2]);
```

Here, we have chosen to print out the first and the last value. If these are correct, then there is a good chance that those in between are also correct.

Solution for exercise 7.7: Sudoku Puzzle

A Sūdoku puzzle is a two-dimensional grid with $9 \cdot 9$ digits. This matches perfectly to a 2D array of integers, which ends up being a single allocation. The more interesting solution is to use an array of arrays (even though the 2D array would be a strictly better solution).

In an array-of-arrays solution, we would have 9 inner arrays (of `int`) embedded in an outer array (of `int[]`). We say that the inner arrays are nested in the outer array. Structurally it looks like this:



As the addresses of the computers memory are laid out in a single dimension, these individual arrays will end up being allocated more or less sequentially. You will have to follow two references

in order to locate the data of a cell, and these references can be redefined. In a 2D array it would be a single allocation, and a single reference to follow, and no possibility of redefining the (missing) inner reference.

Solution for exercise 7.8: Areas of Circles

```
int[] radiuses = {1, 3, 5};
double pi = 3.14;

foreach (int r in radiuses) {
  Console.WriteLine("r=" + r + " -> area=" + (pi * r * r));
}
```

Solution for exercise 7.9: Daily Differences

```
double[] days = {21.5, 23.7, 19.6, 22.5, 25.3, 21.7, 18.9};

for (int i=0 ; i<days.Length-1; i++) {
  Console.WriteLine(days[i + 1] - days[i]);
}
```

Solution for exercise 7.10: Length of Month

```
int[] months = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int month = 6;

if (month < 1 || month > 12) {
  Console.WriteLine("Error: The month \""+month+"\" is not within [1,12]");
} else {
  Console.WriteLine(months[month-1]);
}
```

Solution for exercise 7.11: Primes

```
int last = 1000000;
bool[] sieve = new bool[last];

// initialize sieve
for (int i=1 ; i<sieve.Length ; i++)
  sieve[i] = true;

// fill out sieve
for (int i=2 ; i<Math.Sqrt(sieve.Length) ; i++) {
  if (sieve[i]) {
    for (int j=i*2 ; j<last ; j+=i) {
      sieve[j] = false;
```

```
    }
  }
}

// find and print out largest prime
int max = -1;
for (int i=last-1 ; i>0 ; i--) {
  if (sieve[i]) {
    max = i;
    break;
  }
}
Console.WriteLine("Highest prime is "+max);
```

Solution for exercise 7.12: Calendar

```
int[] monthsNormal = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int[] monthsLeap   = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

for (int year=2000 ; year<=2020 ; year++) {
  bool leap = (year%4==0 && year%100!=0) || (year%400==0);
  int[] month = leap ? monthsLeap : monthsNormal;

  Console.Write(year + ": [ ");
  foreach (int length in month) {
    Console.Write(length + " ");
  }
  Console.WriteLine("]");
}
```

This code is only updating a reference. The contents of the arrays are not being copied.

Solution for exercise 7.13: Calendar Prettyprinting

```
string[] days = {
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
};
string[] months = {
  "January",
  "February",
  "March",
  "April",
```

```
  "May",
  "June",
  "July",
  "August",
  "September",
  "October",
  "November",
  "December"
};
int[] monthsNormal = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int[] monthsLeap = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int year = 2019;
int firstDay = 1; // index for days

// find the correct month lengths
bool leap = (year%4==0 && year%100!=0) || (year%400==0);
int[] monthLengths = leap ? monthsLeap : monthsNormal;

// build base data structure
string[][] calendar = new string[12][];
int dayNumber = firstDay;
for (int month=0 ; month<calendar.Length ; month++) {
  int monthLength = monthLengths[month];
  calendar[month] = new string[monthLength];
  for (int day=0 ; day<monthLength ; day++) {
    calendar[month][day] = days[dayNumber++ % days.Length];
  }
}

for (int month=0 ; month<calendar.Length ; month++) {
  for (int day=0 ; day<calendar[month].Length ; day++) {
    string monthName = months[month];
    string dayName = calendar[month][day];

    Console.Write(year + " " + (day+1) + ". ");
    Console.WriteLine(monthName+": "+dayName);
  }
}
```

Solution for exercise 7.14: Sūdoku Checker

A filled out Sūdoku puzzle has 9 rows and 9 columns. This is in principle a rectangular (square, actually) array. The following solution implements this as an array of arrays (even though a 2d array would have been better).

Every cell has a value between 1 and 9 (both included). This is an integer, and they are small. We can, for instance, use an `int`. We thus end with the type`int[][]`.

Note: We could have used a `byte[][]` and thus saved a bit of memory, but in this case it is very little. By doing this, we would have saved $(4-1)\text{B} \cdot 9 \cdot 9 = 243\text{B}$, but the program would

also have been slower (as 8-bit integers are usually slower that 32-bit integers).

The following is a solution:

```
// test cases
int[][] puzzle = new int[][] {
  new int[] {7, 3, 6, 4, 5, 2, 9, 8, 1},
  new int[] {1, 9, 8, 6, 3, 7, 4, 5, 2},
  new int[] {4, 2, 5, 9, 8, 1, 3, 7, 6},
  new int[] {3, 6, 4, 5, 2, 8, 1, 9, 7},
  new int[] {9, 5, 2, 7, 1, 4, 6, 3, 8},
  new int[] {8, 1, 7, 3, 9, 6, 2, 4, 5},
  new int[] {2, 8, 9, 1, 7, 3, 5, 6, 4},
  new int[] {6, 7, 3, 2, 4, 5, 8, 1, 9},
  new int[] {5, 4, 1, 8, 6, 9, 7, 2, 3}
};
//int[][] puzzle = {
//  new int[] {7, 3, 6, 4, 5, 2, 9, 8, 1},
//  new int[] {1, 9, 8, 6, 3, 7, 4, 5, 2},
//  new int[] {4, 2, 5, 9, 8, 4, 3, 7, 6},
//  new int[] {3, 6, 4, 5, 2, 8, 1, 9, 7},
//  new int[] {9, 5, 2, 7, 1, 4, 6, 3, 8},
//  new int[] {8, 1, 7, 3, 9, 6, 2, 4, 5},
//  new int[] {2, 8, 9, 1, 7, 3, 5, 6, 4},
//  new int[] {6, 7, 3, 2, 4, 5, 8, 1, 9},
//  new int[] {5, 4, 1, 8, 6, 9, 7, 2, 3},
//};

bool correct = true;

// check rows
for (int y=0 ; y<9 ; y++) {
  for (int x=0 ; x<9 ; x++) {
    for (int x2=x+1 ; x2<9 ; x2++) {
      if (puzzle[y][x] == puzzle[y][x2]) {
        correct = false;
      }
    }
  }
}

// check columns
for (int x=0 ; x<9 ; x++) {
  for (int y=0 ; y<9 ; y++) {
    for (int y2=y+1 ; y2<9 ; y2++) {
      if (puzzle[y][x] == puzzle[y2][x]) {
        correct = false;
      }
    }
  }
}
```

```
// check groups
for (int x=0 ; x<3 ; x++) {
  for (int y=0 ; y<3 ; y++) {
    for (int i=0 ; i<9 ; i++) {
      int ix = i % 3;
      int iy = i / 3;
      for (int i2=i+1 ; i2<9 ; i2++) {
        int i2x = i2 % 3;
        int i2y = i2 / 3;
        if (puzzle[y*3+iy][x*3+ix] == puzzle[y*3+i2y][x*3+i2x]) {
          correct = false;
        }
      }
    }
  }
}

Console.WriteLine("Puzzle is " + (correct ? "correct" : "incorrect"));
```

By moving the comment characters to the other `puzzle` definition, one can test the code with both a correct and an incorrect puzzle.

Solution for exercise 7.15: Person

```
Person p = new Person {name="Kurt Ole", age=12};

Console.WriteLine(p.name+" is "+p.age+" years old");

class Person {
  public string? name;
  public int age;
}
```

Solution for exercise 7.16: Direction

Because we write code in English a good name for type would be `Direction`. Convention dictations that the first letter should be capitalized. Even though there are multiple directions, we use the singular form of the noun.

Similar rules apply for the naming of the individual values. One could consider shortening the directions to `N`, `S`, `E`, and `W`. That, however, would also make it harder to interpret the code.

```
Console.WriteLine(Direction.North);

enum Direction {
  North,
  South,
  East,
```

```
  West,
}
```

Solution for exercise 7.17: Units

Subtask 1

The other variable is used to identify a unit, so perhaps we should just call it `unit`? It needs to be of a type that can represent each of the supported types, and that is it's only purpose: to represent on of a number of named values. We normally use an `enum` for this.

Subtask 2

There is a number of ways of mapping a unit type to a factor. We will highlight two such methods:

- ***Switch Statement:*** We switch on the the unit and add a case for each supported unit. In each of these cases, the resulting conversion is performed.
- ***Array Lookup:*** Allocate an array of all the relevant factors, and use the integer value from the `enum` unit type as index.

Subtask 3

The straight-forward solution (that uses a switch statement) looks like this:

```
double value = 1.234; // m
Unit unit = Unit.MilliMeter;

switch (unit) {
  case Unit.Meter:
    value *= 1;
    break;
  case Unit.CentiMeter:
    value *= 100;
    break;
  case Unit.MilliMeter:
    value *= 1000;
    break;
  case Unit.Inches:
    value *= 100 / 2.54;
    break;
  default:
    Console.WriteLine("Oh, no!");
    break;
}

Console.WriteLine("Result is "+value);

enum Unit {
  Meter,
```

```
  CentiMeter,
  MilliMeter,
  Inches,
}
```

In a larger program, that one would have to develop and maintain over time, a better approach would be (to use an array lookup):

```
// init
double[] factors = new double[(int)Unit.Size];
factors[(int)Unit.Meter] = 1;
factors[(int)Unit.CentiMeter] = 100;
factors[(int)Unit.MilliMeter] = 1000;
factors[(int)Unit.Inches] = 100 / 2.54;

double value = 1.234; // m
Unit unit = Unit.MilliMeter;

Console.WriteLine("Result is "+(value*factors[(int)unit]));

enum Unit {
  Meter,
  CentiMeter,
  MilliMeter,
  Inches,
  Size,
}
```

Here, the initialization phase would end up in a separate file.

A.6 Functions

Solution for exercise 8.1: Use

A function is an abstraction that is used to create reusable chunks of named and parameterized code. Functions are used to organize code and to avoid duplicate code.

Solution for exercise 8.2: Lack of Return Value

A functions return type defines the type of the value that it returns. A special type called `void` is used to indicate that no value is returned. When no value is returned, there is no need for a `return`-statement, and if present it doesn't take a parameter.

Solution for exercise 8.3: Sūdoku Prettyprinter

```
int[][] p1 = {
  new int[] {7, 3, 6, 4, 5, 2, 9, 8, 1},
  new int[] {1, 9, 8, 6, 3, 7, 4, 5, 2},
  new int[] {4, 2, 5, 9, 8, 1, 3, 7, 6},
```

```
  new int[] {3, 6, 4, 5, 2, 8, 1, 9, 7},
  new int[] {9, 5, 2, 7, 1, 4, 6, 3, 8},
  new int[] {8, 1, 7, 3, 9, 6, 2, 4, 5},
  new int[] {2, 8, 9, 1, 7, 3, 5, 6, 4},
  new int[] {6, 7, 3, 2, 4, 5, 8, 1, 9},
  new int[] {5, 4, 1, 8, 6, 9, 7, 2, 3},
};

void print(int[][] puzzle) {
  for (int row=0 ; row<9 ; row++) {
    if (row % 3 == 0) {
      Console.WriteLine("+-----+-----+-----+");
    }

    for (int col=0 ; col<9 ; col++) {
      Console.Write((col % 3 == 0 ? "|" : " ") + puzzle[row][col]);
    }

    Console.WriteLine("|");
  }

  Console.WriteLine("+-----+-----+-----+");
}

print(p1);
```

Solution for exercise 8.4: Sum

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

for (int a=0 ; a<=10 ; a++) {
  for (int b=0 ; b<=10 ; b++) {
    Console.Write("{0,3}", Add(a, b));
  }
  Console.WriteLine();
}
```

Solution for exercise 8.5: Own Square Root

```
double square_root(double x) {
  double candidate = 0.0;

  for (double pointer = 1000000000.0 ; pointer > 0.000000001 ; pointer /= 10) {
    candidate += 10 * pointer;
    for (int i = 0 ; i < 10 ; i++) {
      candidate -= pointer;
```

```
      if (candidate * candidate < x) {
        break;
      }
    }
  }

  return candidate;
}

for (double x = 1; x <= 10; x++) {
  Console.WriteLine("sqrt(" + x + ")=" + square_root(x));
}
```

Solution for exercise 8.6: Mathematical Equivalent

Ma mathematical function is a really good match for a function in C#. The big difference is that C#functions – unlike mathematical functions – can have side-effects. They can access state (in memory). Through variables, they read read from and write to the main memory of the computer. This allows them to express behavior that depends on the state of that memory.

Solution for exercise 8.7: Discriminants and Roots

```
double discriminant (double a, double b, double c) {
  return b * b - 4 * a * c;
}

int signum (double value) {
  if (value < 0) {
    return -1;
  } else if (value == 0) {
    return 0;
  } else {
    return 1;
  }
}

double[] calculate_roots (double a, double b, double c) {
  double d = discriminant(a, b, c);
  int rootCount = signum(d);
  double[] roots = null;

  switch (rootCount) {
    case -1:
      roots = new double[0];
      break;
    case 0:
      roots = new double[1];
      roots[0] = (-b) / (2 * a);
      break;
```

```
    case 1:
      roots = new double[2];
      roots[0] = (-b + Math.Sqrt(d)) / (2 * a);
      roots[1] = (-b - Math.Sqrt(d)) / (2 * a);
      break;
  }

  return roots;
}

double[][] tests = {
  new double[] { 1, -4, 1 },
  new double[] { 1, -4, 4 },
  new double[] { 1, -4, 10 },
};

foreach (double[] testcase in tests) {
  double a = testcase[0];
  double b = testcase[1];
  double c = testcase[2];

  double[] rs = calculate_roots(a, b, c);

  Console.WriteLine("y = "+a+"x² + "+b+"x + "+c+" has "+rs.Length+" roots:");
  for (int i = 0; i < rs.Length; i++) {
    Console.WriteLine(" - Root " + i + ": " + rs[i]);
  }
  Console.WriteLine("");
}
```

Solution for exercise 8.8: Factorial function

```
int fac (int i) {
  if (i < 0) {
    Console.WriteLine("That was not very nice of you!");
  }

  if (i > 0) {
    return i * fac(i - 1);
  } else {
    return 1;
  }
}

for (int i = 1 ; i < 10 ; i++) {
  Console.WriteLine(" fac(" + i + ") = " + fac(i));
}
```

Solution for exercise 8.9: Properties of Circles

```
const double pi = 3.14;

double area (double r) {
  return pi * r * r;
}

double circumference (double r) {
  return 2 * pi * r;
}

for (double r = 1 ; r <= 5 ; r += 2) {
  Console.WriteLine("r="+r+" => area="+area(r)+" circ="+circumference(r));
}
```

A.7 Exceptions

SOLUTION FOR EXERCISE 9.1: INDEXING

Delopgave 1

To compile and execute the code:

```
$ dotnet build
...
$ dotnet run
Unhandled exception. System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at Program.<Main>$(String[] args) in .../ArrayIncrementer.cs:line 6
$
```

Delopgave 2

In the printout of this run we can see that a `System.IndexOutOfRangeException` exception is thrown.

Delopgave 3

The exception is thrown on line 6 of `ArrayIncrementer`. On this line `array` is being indexed using `i` which at this point is bound to the value 5. As `array` is initialized to hold 5 values the space allocated can only hold 5 values, and as C# is a 0-indexed language this means that the last valid index is 4. To avoid us trying to access memory outside of the allocation of the process, the C# runtime throws this exception.

Delopgave 4

```
int iterationer = 10;
int[] array = {1, 2, 3, 4, 5};

// increment
for (int i=0 ; i< iterationer ; i++) {
  try {
    array[i]++;
  } catch (IndexOutOfRangeException) {
    Console.WriteLine("Oopsie!");
```

```
  }
}

// print
for (int i=0 ; i<array.Length ; i++) {
  Console.WriteLine(array[i]);
}
```

Delopgave 5

No, this is not the correct solution. This is caused by a bug in the code. It is clear to us that this exception will be thrown every time the code is executed. A non-ugly solution will involve us not getting into a situation where this exception is thrown, and this can be done by *correcting* the code.

Solution for exercise 9.2: Accounts

```
int[] accounts = {903, 716, 67};

int GetAccountNumber ()
{
  Console.WriteLine("Enter an account number: ");
  while (true) {
    try {
      return Convert.ToInt32(Console.ReadLine());
    } catch (FormatException) {
      Console.WriteLine("I'm afraid that I don't understand. Please try again: ");
    }
  }

}

void PrintAccountState (int accountId)
{
  Console.WriteLine("Account " + accountId + " contains " + accounts[accountId]);
}


while (true)
{
  int accountId = GetAccountNumber();
  try {
    PrintAccountState(accountId);
  } catch (IndexOutOfRangeException) {
    Console.WriteLine("Unknown account: " + accountId);
  }
}
```

Solution for exercise 9.3: Average Grade

```
int[] grades = {4, 7, 02, 00, 10, 4, 12};

int GetGrade (int courseId)
{
  int grade = grades[courseId];
  if (grade >= 02) {
    return grade;
  } else {
    throw new Exception(); // this is an exceptional grade!
  }
}

int count = 0;
int sum   = 0;

for (int courseId=0 ; courseId<grades.Length ; courseId++) {
  try {
    sum += GetGrade(courseId);
    count++;
  } catch (Exception) {
  }
}

Console.WriteLine("Average grade is " + ((float)sum / count));
```

A.8 Objects

Solution for exercise 12.1: Customers

```
public class Customer
{
  public string name;
  public int    id;
  public double balance;

  public Customer (string nameValue, int idValue, double balanceValue) {
    name    = nameValue;
    id      = idValue;
    balance = balanceValue;
  }

  public Customer (string nameValue, int idValue) {
    name    = nameValue;
    id      = idValue;
    balance = 0;
  }

  public void Deposit (double amount) {
    balance += amount;
  }
```

```
  public void Withdraw (double amount) {
    balance -= (balance > amount ? amount : 0);
  }

  public double GetBalance () {
    return balance;
  }

  public static void Main (string[] args) {
    Customer aCustomer;

    aCustomer = new Customer("Donald Knuth", 42, 0);
    Console.WriteLine(aCustomer.GetBalance());
    aCustomer.Deposit(1000);
    Console.WriteLine(aCustomer.GetBalance());
    aCustomer.Withdraw(500);
    Console.WriteLine(aCustomer.GetBalance());
  }
}
```

Solution for exercise 12.2: Customer Database

```
class CustomerDatabase
{
  Customer[] customers;

  public CustomerDatabase () {
    customers = new Customer[10];
  }

  public bool Add (Customer customer) {
    for (int i = 0; i < customers.Length; i++) {
      if (customers[i] == null) {
        customers[i] = customer;
        return true;
      }
    }
    return false;
  }

  public void Remove (int id) {
    for (int i = 0; i < customers.Length; i++) {
      if (customers[i] != null && customers[i].id == id) {
        customers[i] = null;
      }
    }
  }

  public Customer[] Get () {
```

```
    return (Customer[])customers.Clone();
  }

  public void Print () {
    foreach (Customer c in customers) {
      if (c == null) continue;
      Console.WriteLine(c.name);
    }
  }

  public static void Main (string[] args) {
    CustomerDatabase db = new CustomerDatabase();

    Console.WriteLine("Result:" + db.Add(new Customer("Alan Turing", 1912)));
    Console.WriteLine("Result:" + db.Add(new Customer("Ada Lovelace", 1815)));
    Console.WriteLine("Result:" + db.Add(new Customer("Charles Babbage", 1791)));
    Console.WriteLine("Result:" + db.Add(new Customer("Donald Knuth", 1938)));
    Console.WriteLine("Result:" + db.Add(new Customer("Grace Hopper", 1906)));
    Console.WriteLine("Result:" + db.Add(new Customer("Edsger Dijkstra", 1930)));
    Console.WriteLine("Result:" + db.Add(new Customer("Tony Hoare", 1934)));
    Console.WriteLine("Result:" + db.Add(new Customer("Hedy Lamarr", 1914)));
    Console.WriteLine("Result:" + db.Add(new Customer("Michael Stonebraker", 1943)));
    Console.WriteLine("Result:" + db.Add(new Customer("Jim Gray", 1944)));
    Console.WriteLine("Result:" + db.Add(new Customer("Douglas Engelbart", 1925))); // Should fai
    Console.WriteLine();

    db.Print();
  }
}
```

Note: The `Get` method returns a copy of the data structure in order to make it impossible for to calling method to name the original data through an old reference.

A.9 Inheritance

Solution for exercise 13.1: Inventory

```
class Item {
  public string name;
  public double price;

  public Item (string nameValue, double priceValue)
  {
    name  = nameValue;
    price = priceValue;
  }

  public string GetName ()
  {
```

```
    return name;
  }

  public double GetPrice ()
  {
    return price;
  }
}

class FoodItem : Item {
  private DateTime expiresAt;

  public FoodItem (string name, double price, DateTime expiresAtValue)
       : base(name, price) {
    expiresAt = expiresAtValue;
  }

  public DateTime GetExpiresAt () {
    return expiresAt;
  }

  public override string ToString () {
    return "FoodItem name='"+GetName()+"'"
                  +" price='"+GetPrice()+"'"
                  +" expiresAt='"+GetExpiresAt()+"'";
  }

  // comment out the Main method to use this class as the entry point of the program

  //public static void Main(string[] args) {
  //  FoodItem[] items = new FoodItem[10];
  //
  //  for (int i=0 ; i<items.Length ; i++) {
  //    items[i] = new FoodItem($"Item {i}", 12.3 * i, DateTime.Now.AddDays(i));
  //  }
  //
  //  foreach (FoodItem item in items) {
  //    Console.WriteLine(item);
  //  }
  //}
}

class NonFoodItem : Item {
  public string[] materials;

  public NonFoodItem (string name, double price, string[] materialsValue)
       : base(name, price) {
      materials = materialsValue;
  }
```

```
  public string[] GetMaterials () {
    return materials;
  }

  public override string ToString () {
    string m = "[";
    for (int i=0 ; i<materials.Length ; i++) {
      m += (i == 0 ? "" : ",") + materials[i];
    }
    m += "]";
    return "NonFoodItem name='" + GetName()
         + "' price='" + GetPrice()
         + "' materials='" + m + "'";
  }

  // comment out the Main method to use this class as the entry point of the program

  //public static void Main (string[] args)
  //{
  //  NonFoodItem[] items = new NonFoodItem[10];
  //
  //  for (int i=0 ; i<items.Length ; i++) {
  //    items[i] = new NonFoodItem("Item " + i, 12.3 * i,
  //                               new string[] { "butter", "cream" });
  //  }
  //
  //  foreach (NonFoodItem item in items) {
  //    Console.WriteLine(item);
  //  }
  //}
}

class Mixed
{
  // comment out the Main method to use this class as the entry point of the program
  static void Main (string[] args) {
    Item[] items = new Item[10];

    for (int i=0 ; i<items.Length ; i++) {
      if (i % 2 == 0) {
        items[i] = new FoodItem("Item " + i, 12.3 * i,
                                new DateTime(i * 1000 * 60 * 60 * 24));
      } else {
        items[i] = new NonFoodItem("Item " + i, 12.3 * i,
                                   new string[] { "butter", "cream" });
      }
    }

    foreach (Item item in items) {
      Console.WriteLine(item);
```

```
    }
  }
}
```

A.10 Naming

Solution for exercise 14.1: Kittens

```
class Kitten{
  double cuteness;

  static int count = 0;

  public Kitten (double cuteness) {
    this.cuteness = cuteness;
    if (cuteness > 9000) {
      Console.WriteLine("The cuteness of this kitten is over 9000!");
    }
    count++;
  }

  public static void Main (string[] args) {
    for (int i = 0; i < 9002; i++) {
      new Kitten(i);
    }
    Console.WriteLine("There are " + count + " kittens.");
  }
}
```

Solution for exercise 14.2: Addition

```
class Adder
{
  static string Solve (int a, int b) {
    return a+" + "+b+" = "+(a+b);
  }

  static string Solve (double a, double b) {
    return a+" + "+b+" = "+(a+b);
  }

  public static void Main (string[] args) {
    Console.WriteLine(Solve(3, 5));
    Console.WriteLine(Solve(3.14, 5.12));
  }
}
```

When we call `Solve` with an `int` and a `double` that will result in the compiler looking for an implementation of this method with a signature that matches `Solve` × `int` × `double`. As this does not exist, then compilation will fail.

Solution for exercise 14.3: Operator

There are two operators; `+` and `-`. The mapping of the five calls to the `Operator` method to operations are:

$$\begin{aligned}(?1?) &\mapsto + \\ (?2?) &\mapsto + \\ (?3?) &\mapsto - \\ (?4?) &\mapsto - \\ (?5?) &\mapsto -\end{aligned}$$

Of these `+` is implemented by the first declaration of the `Operator` method, and `-` is implemented by the last. These declarations obviously have different bodies, but they are also differentiated by their signatures. Here, the type of the third argument is different. The value of the parameter is not used for anything, but the parameter type makes the two signatures unique, and because of that we can name each of them by supplying a value of the right type.

This allows the choice of which `Operator` variant to be called to be determined at compile time. This is fast. But it is also an implementation that is hard to read. Many would likely consider this a misuse of the type system. In reality, most compilers will be able to compile the following implementation to bytecode that is just as efficient:

```
class OperatorTest {
  public static int Operator (int a, int b, bool plus) {
    if (plus) {
      return a + b;
    } else {
      return a - b;
    }
  }

  public static void Main (string[] args) {
    for (int a=0 ; a<5; a++) {
      for (int b=0 ; b<5; b++) {
        Console.WriteLine(a + " (?1?) " + b + " = " + Operator(a, b, true));
        Console.WriteLine(a + " (?2?) " + b + " = " + Operator(a, b, true));
        Console.WriteLine(a + " (?3?) " + b + " = " + Operator(a, b, false));
        Console.WriteLine(a + " (?4?) " + b + " = " + Operator(a, b, false));
        Console.WriteLine(a + " (?5?) " + b + " = " + Operator(a, b, false));
      }
    }
  }
}
```

On top of this, it is debatable whether it is good style to provide an operator in the form of a parameter.

Solution for exercise 14.4: Scope

The following table shows where each to the five variables can and should (marked with underline) be declared.

Lokation	`i`	`d`	`tmp`	`sum`	`bonus`
Location 0	×	×	×	×	<u>×</u>
Location 1	×	×	×	×	
Location 2		<u>×</u>			
Location 3					
Location 4					
Location 5	×	×	×	×	
Location 6	×		×	<u>×</u>	
Location 7	×		×		
Location 8	<u>×</u>		×		
Location 9					
Location 10			×		
Location 11			<u>×</u>		
Location 12					
Location 13					
Location 14					
Location 15					
Location 16	×	×	×	×	

A variable must be declared before the first reference of it, and it should be declared in the narrowest possible scope. If you declare it in a wider scope, you must ensure that it cannot hold an unintentional value across multiple evaluations of the inner scope.

Even though the order of the declarations at class level doesn't matter, it is good practice to declare the variable at the same line as the first assignment. On top of that, convention dictates that field declarations should be at the top of a class definition.

C#does not allow us to redeclare a variable. If you try to do so, then the program won't compile.

Solution for exercise 14.5: Encapsulation

The program constructs a `Circle` object, increases the radius by 37% and adds 0.65 to the x-coordiate of the center. The state of teh the object is printed out – via its attributes – both before and after the changes are made.

To replace the radius attribute `r` with the diameter attribute `d` in `Circle`, we have to:

- Update the declaration of the attribute in `Circle`.
- Update the initialization of the attribute in the constructor for `Circle`. Now, we need to multiply the parameter (which is still a radius) by two.
- The two printouts in the `Main` methods needs to be updated to refer to the new attribute name.
- The line that increases `r` needs to be converted to a line that increases `d`.

The result is:

```
class Circle {
  public double x, y;
  public double d;

  public Circle (double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.d = radius * 2;
  }
}

public class TestCircle {
  public static void Main (string[] args)
  {
     Circle c = new Circle(1.24, 2.83, 12.7);
     Console.WriteLine("X=" + c.x + " Y=" + c.y + " D=" + c.d);
     c.d *= 1.37;
     c.x += 0.65;
     Console.WriteLine("X=" + c.x + " Y=" + c.y + " D=" + c.d);
  }
}
```

And after encapsulation, the code base looks like this:

```
class Circle {
  private double x, y;
  private double d;

  public Circle (double x, double y, double radius)
  {
    this.x = x;
    this.y = y;
    this.d = radius * 2;
  }

  public double GetX () {
    return x;
  }
  public void SetX (double x) {
    this.x = x;
  }

  public double GetY () {
    return y;
  }
  public void SetY (double y) {
    this.y = y;
```

```
  }

  public double GetD () {
    return d;
  }
  public void SetD (double d) {
    this.d = d;
  }
}

public class TestCircle {
  public static void Main (string[] args) {
    Circle c = new Circle(1.24, 2.83, 12.7);
    Console.WriteLine("x=" + c.GetX() + " y=" + c.GetY() + " d=" + c.GetD());
    c.SetD(c.GetD() * 1.37);
    c.SetX(c.GetX() + 0.65);
    Console.WriteLine("x=" + c.GetX() + " y=" + c.GetY() + " d=" + c.GetD());
  }
}
```

To replace the attributes `x` and `y` with an attribute `c` of type `Coordinate` we have to:

- Add the class `Coordinate`.
- Replace the declaration of the attributes `x` and `y` in `Circle` with a single declaration of the coordinate `c`.
- Replace the initialization of the attributes `x` and `y` in the `Circle` constructor with a single initialization of `c` using a call to the `Coordinate` constructor.
- Getters and setters for the attributes `x` og `y` are kept but updated to forward their calls to the corresponding methods in the attribute `c`.

The result is:

```
public class Coordinate {
  private double x, y;

  public Coordinate (double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double GetX () {
    return x;
  }
  public void SetX (double x) {
    this.x = x;
  }
```

```
  public double GetY () {
    return y;
  }
  public void SetY (double y) {
    this.y = y;
  }
}

public class Circle {
  private Coordinate c;
  private double d;

  public Circle (double x, double y, double radius)
  {
    this.c = new Coordinate(x, y);
    this.d = radius * 2;
  }

  public double GetX () {
    return c.GetX();
  }
  public void SetX (double x) {
    c.SetX(x);
  }

  public double GetY () {
    return c.GetY();
  }
  public void SetY (double y) {
    c.SetY(y);
  }

  public double GetD () {
    return d;
  }
  public void SetD (double d) {
    this.d = d;
  }
}

public class TestCircle {
  public static void Main (string[] args) {
     Circle c = new Circle(1.24, 2.83, 12.7);
     Console.WriteLine("x=" + c.GetX() + " y=" + c.GetY() + " d=" + c.GetD());
     c.SetD(c.GetD() * 1.37);
     c.SetX(c.GetX() + 0.65);
     Console.WriteLine("x=" + c.GetX() + " y=" + c.GetY() + " d=" + c.GetD());
  }
}
```

The update to diameter is not directly comparable to the update to `Coordinate`; one is simple, the other complex. But if we look past this difference, then the two changes are very similar inside `Circle` and different where we use the definition of `Circle` (our `Main` method). The change in the encapsulated variant of `Circle` did not affect the code base *outside of* `Circle` at all. Here, we consider `Coordinate` to be inside of `Circle`. We say that encapsulation ensures that we can modify the *internal* representation without affecting the *external* code.

The downside of encapsulation is that we have to go through getters and setters in stead of manipulating the attributes directly. This gives us som clunkier code, but it will usually be an acceptable trade; especially in larger code bases.

A.11 Polymorphism

Solution for exercise 15.1: Inventory

Subtask 1

```
public class Item
{
  private string name;
  private double price;

  public Item (string name, double price) {
    this.name = name;
    this.price = price;
  }

  public string GetName () {
    return name;
  }

  public double GetPrice () {
    return price;
  }
}

class FoodItem : Item
{
  private DateTime expiresAt;

  public FoodItem (string name, double price, DateTime expiresAt) : base(name, price) {
    this.expiresAt = expiresAt;
  }

  public DateTime GetExpiresAt () {
    return expiresAt;
  }

  public override string ToString () {
    return "FoodItem name='"+GetName()+"'"
```

```
                +" price='"+GetPrice()+"'"
                +" expiresAt='"+GetExpiresAt()+"'";
  }
}

class NonFoodItem : Item
{
  private string[] materials;

  public NonFoodItem (string name, double price, string[] materials)
       : base(name, price) {
    this.materials = materials;
  }

  public string[] GetMaterials () {
    return materials;
  }

  public override string ToString () {
    string m = "[";
    for (int i=0 ; i<materials.Length ; i++) {
        m += (i == 0 ? "" : ",") + materials[i];
    }
    m += "]";

    return "NonFoodItem name='"+GetName()+"' price='"+GetPrice()+"' materials='"+m+"'";
  }
}

public class Inventory
{
  private Item[] items;

  public Inventory (Item[] items) {
    this.items = items;
  }

  public Inventory () : this(new Item[0]) {
  }

  public void AddItem (Item item) {
    if (!Contains(items, item)) {
      Item[] newItems = new Item[items.Length+1];

      for (int i=0 ; i<items.Length ; i++) {
        newItems[i] = items[i];
      }
      newItems[items.Length] = item;

      items = newItems;
```

```
    }
  }

  public void RemoveItem (Item item) {
    if (Contains(items, item)) {
      Item[] newItems = new Item[items.Length-1];

      int i = 0;
      for ( ; items[i]!=item ; i++) {
        newItems[i] = items[i];
      }
      for ( i++ ; i<items.Length ; i++) {
        newItems[i-1] = items[i];
      }

      items = newItems;
    }
  }

  public double GetInventory () {
    double total = 0.0;

    foreach (Item item in items) {
      total += item.GetPrice();
    }

    return total;
  }

  public void PrintInventory () {
    Console.WriteLine("Inventory:");
    foreach (Item item in items) {
      Console.WriteLine(" - " + item);
    }
  }

  private bool Contains (Item[] items, Item item) {
    foreach (Item candidateItem in items) {
      if (candidateItem==item) return true;
    }
    return false;
  }
}
```

Subtask 2

```
public class Test
{
  public static void TestFoodItem () {
    FoodItem[] items = new FoodItem[10];
```

```
    for (int i=0; i<items.Length ; i++)
    {
       items[i] = new FoodItem($"Item {i}", 12.3 * i, DateTime.Now.AddDays(i));
    }

    foreach (FoodItem item in items)
    {
       Console.WriteLine(item);
    }
  }

  public static void TestNobFoodItem () {
    NonFoodItem[] items = new NonFoodItem[10];

    for (int i=0 ; i<items.Length ; i++) {
      items[i] = new NonFoodItem("Item " + i,
                                 12.3 * i,
                                 new string[] { "butter", "cream" });
    }

    foreach (NonFoodItem item in items) {
      Console.WriteLine(item);
    }
  }

  public static void PrintStatus (Inventory inventory) {
    inventory.PrintInventory();
    Console.WriteLine("Total: " + inventory.GetInventory());
    Console.WriteLine();
  }

  public static void Main (string[] args) {
    Inventory inventory = new Inventory();
    Item i1 = new Item("chocolate", 19.95);
    Item i2 = new Item("coffee", 24.95);
    Item i3 = new FoodItem("Milk", 12.95, new DateTime());
    Item i4 = new NonFoodItem("USB Charger", 17.45,
                              new string[] { "plastic", "stuff" });
    Item[] items = new Item[] { i1, i2, i3, i4 };

    PrintStatus(inventory);
    foreach (Item item in items) {
      inventory.AddItem(item);
      PrintStatus(inventory);
    }

    inventory.RemoveItem(i1);
    PrintStatus(inventory);
  }
```

```
}
```

Solution for exercise 15.2: A Day at the Races

Subtask 1

```
public class Racer
{
  public    string Name;
  protected double speed;
  protected double distance;

  public Racer (string name, double speed) {
    Name = name;
    this.speed = speed;
  }

  public virtual double Race (double time) {
    distance += speed * time;
    return distance;
  }
}
```

Subtask 2

```
public class RaceCar : Racer
{
  private double fuel;
  private double fuelEfficiency;

  public RaceCar (string name, double speed, double fuel, double fuelEfficiency)
       : base (name, speed) {
    this.fuel = fuel;
    this.fuelEfficiency = fuelEfficiency;
  }

  public override double Race (double time) {
    fuel -= time / fuelEfficiency;
    if (fuel <= 0)
      speed = 0;
    return base.Race(time);
  }
}

public class RaceHorse : Racer
{
  private double stamina;

  public RaceHorse (string name, double speed, double stamina) : base(name, speed) {
```

```
    this.stamina = stamina;
  }

  public override double Race (double time) {
    speed -= time / stamina;
    speed = (speed>2 ? speed : 2);
    return base.Race(time);
  }
}
```

Subtask 3

No reference solution provided.

Subtask 4

```
public class Program
{
  static void Main () {
    for (int i=1 ; i<6 ; i++){
      RaceCar   car   = new RaceCar("McQueen", 12, 5, 1.5);
      RaceHorse horse = new RaceHorse("Secretariat", 8, 5);
      Racer[] racers = [car, horse];
      Race(racers, 20*i, 1);
    }
  }

  private static void Race (Racer[] racers, double raceDistance, double playbackSpeed) {
    foreach (Racer racer in racers) {
      Console.WriteLine(racer.Name);
    }
    Thread.Sleep(1000);

    bool finished = false;
    while (!finished) {
      for (int i=0 ; i<40 ; i++) Console.WriteLine();
      foreach (Racer racer in racers) {
        double distance = racer.Race(playbackSpeed);
        for (int i=0 ; i<distance ; i++){
          Console.Write("-");
        }
        Console.WriteLine(racer.Name);
        if (distance >= raceDistance)
          finished = true;
      }
      Thread.Sleep(200);
      Console.WriteLine();
    }
  }
}
```

Subtask 5

If two racers cross the line in the same simulation step, perhaps we should use their speed to decide who won?

A.12 Abstract Methods

Solution for exercise 16.1: Inventory

Subtask 1

```
public abstract class Item
{
  private string name;
  private double price;

  public Item (string name, double price) {
    this.name = name;
    this.price = price;
  }

  public string GetName () {
    return name;
  }

  public double GetPrice () {
    return price;
  }
}
```

Attempting to call the constructor for `Item` will result in C#giving us a compile-time error. The error happens at compile-time because it is at this point that the rule about not being allowed to instantiate abstract classes is checked.

Subtask 2

```
public interface IExpirable
{
  bool IsExpired ();
}
```

Subtask 3

```
public abstract class Item : IExpirable
{
  private string name;
  private double price;

  public Item (string name, double price) {
```

```
    this.name = name;
    this.price = price;
  }

  public string GetName () {
    return name;
  }

  public double GetPrice () {
    return price;
  }

  public abstract bool IsExpired ();
}
```

Subtask 4

```
class FoodItem : Item
{
  private DateTime expires;

  public FoodItem (string name, double price, DateTime expires) : base(name, price) {
    this.expires = expires;
  }

  public DateTime GetExpires () {
    return expires;
  }

  public override bool IsExpired () {
    return DateTime.Now > expires;
  }

  public override string ToString () {
    return "FoodItem name='"+GetName()+"'"
                  +" price='"+GetPrice()+"'"
                  +" expires='"+GetExpires()+"'";
  }
}
```

Subtask 5

```
public class Inventory
{
  private Item[] items;

  public Inventory (Item[] items) {
    this.items = items;
  }
```

```
public Inventory () : this(new Item[0]) {
}

public void AddItem (Item item) {
  if (!Contains(items, item)) {
    Item[] newItems = new Item[items.Length+1];

    for (int i=0 ; i<items.Length ; i++) {
      newItems[i] = items[i];
    }
    newItems[items.Length] = item;

    items = newItems;
  }
}

public void RemoveItem (Item item) {
  if (Contains(items, item)) {
    Item[] newItems = new Item[items.Length-1];

    int i = 0;
    for ( ; items[i]!=item ; i++) {
      newItems[i] = items[i];
    }
    for ( i++ ; i<items.Length ; i++) {
      newItems[i-1] = items[i];
    }

    items = newItems;
  }
}

public double GetInventory () {
  double total = 0.0;

  foreach (Item item in items) {
    total += item.GetPrice();
  }

  return total;
}

public void PrintInventory () {
  Console.WriteLine("Inventory:");
  foreach (Item item in items) {
    Console.WriteLine(" - " + item);
  }
}
```

```
  private bool Contains (Item[] items, Item item) {
    foreach (Item candidateItem in items) {
      if (candidateItem==item) return true;
    }
    return false;
  }
}
```

A.13 Software Development

Solution for exercise 17.1: Enrollment System

The new program specification with noun-verb analysis:

The enrollment system contains a list of students and courses. It can display the list of students which are enrolled in a particular course. It is possible to enroll students to a course and remove them from a course. It is possible to add and remove, both students and courses, to/from the enrollment system.

The new requirement specification:

- ***Functional requirements:***

ID	*Name*	*Description*
F01	Display a list of students	It must be possible to display a list of students
F02	Display a list of courses	It must be possible to display a list of courses
F03	Add and remove association	It must be possible to associate students to a course, and to remove such associations
F04	Add and remove students	It must be possible to add students to and remove students from the system
F05	Add and remove courses	It must be possible to add students to and remove courses from the system

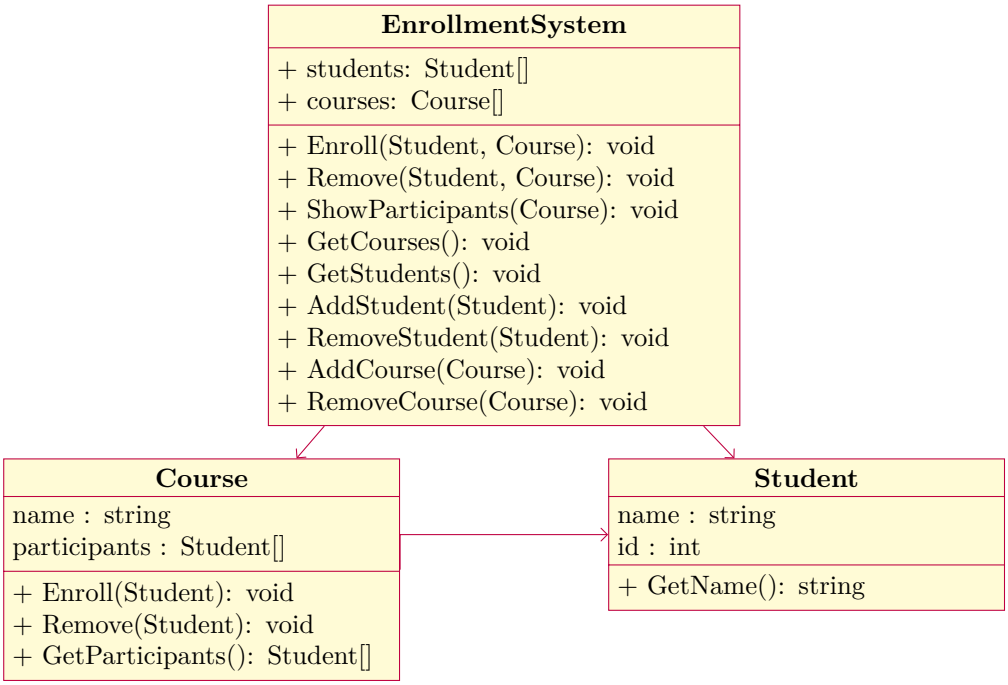
- ***Non-functional requirements:***

ID	*Name*	*Description*
NF01	Capacity	The system must be able to handle 10 courses
NF02	Interoperability	The system must be able to integrate with itslearning

This leads to the new CRC card:

EnrollmentSystem	
Responsibilities	*Collaborators*
• Contains list of `Course`-objects • Contains list of `Student`-objects • Can enroll `Student`-objects in Course-objects • Can remove `Student`-objects from Course-objects • Can display `Student`-objects enrolled in any given `Course`-object • Can add `Student`-objects to the list of `Student`-objects • Can remove `Student`-objects from the list of `Student`-objects • Can add `Course`-objects to the list of `Course`-objects • Can remove `Course`-objects from the list of `Course`-objects	• Student • Course

Based on this we get the following UML class diagram:



In the final code, only the `EnrollmentSystem` class has been modified:

```
public class EnrollmentSystem
{
  public Student[] students;
```

```
public Course[]  courses;

public void Enroll (Student student, Course course) {
  students = new Student[0];
  courses = new Course[0];
  course.Enroll(student);
}

public void Remove (Student student, Course course) {
  course.Remove(student);
}

public void ShowParticipants (Course course) {
  foreach (Student student in course.GetParticipants()) {
    Console.WriteLine(student.GetName());
  }
}

public void GetCourses () {
  for (int i=0 ; i<courses.Length ; i++) {
    Console.WriteLine(courses[i]);
  }
}

public void GetStudents () {
  for (int i=0 ; i<students.Length ; i++) {
    Console.WriteLine(students[i].GetName());
  }
}

public void AddStudent (Student student) {
  // avoid duplicates
  foreach (Student entry in students) {
    if (entry == student) {
      return;
    }
  }

  // create new list
  Student[] newList = new Student[students.Length + 1];
  for (int i=0 ; i<students.Length ; i++) {
    newList[i] = students[i];
  }
  newList[students.Length] = student;

  // override old reference
  students = newList;
}

public void RemoveStudent (Student student) {
```

```
    int i;

    // find index of student
    for (i=0 ; i<students.Length ; i++) {
      if (students[i] == student) {
        break;
      }
    }

    // guard: student not in list
    if (i == students.Length) {
      return;
    }

    // create new list
    Student[] newList = new Student[students.Length - 1];
    int n = 0;
    for (int o=0 ; o<students.Length ; o++) {
      if (o != i) {
        newList[n++] = students[o];
      }
      o++;
    }
    newList[students.Length] = student;

    // override old reference
    students = newList;
  }
}
```

A.14 Parameterized Types

Solution for exercise 18.1: Dynamic Array

```
public class DynArray<T> : IDynArray<T>
{
  T[] data;
  int capacity;
  int fill;

  // constructors

  public DynArray () {
    data = new T[0];
    capacity = 0;
    fill = 0;
  }

  // interface methods
```

```
public void Append (T element) {
  Expand(1);
  data[fill++] = element;
}

public void Insert (int i, T element) {
  Expand(1);
  for (int j=fill ; j>i ; j++) {
    data[j] = data[j-1];
  }
  data[i] = element;
  fill++;
}

public void Remove (int i) {
  for (i++ ; i<fill ; i++) {
    data[i-1] = data[i];
  }
  fill--;
  Shrink();
}

public void Set (int i, T element) {
  data[i] = element;
}

public T Get (int i) {
  return data[i];
}

public int GetFill () {
  return fill;
}

public override string ToString () {
  string s = "DynArray [";
  for (int i=0 ; i<capacity ; i++) {
    s += (i==0 ? "" : "|")+(i>=fill ? "X" : data[i]);
  }
  s += "] utilization=["+fill+"/"+capacity+"]";
  return s;
}

// helper methods

private void Shrink () {
  if (fill<=capacity/2) {
    capacity = (capacity==1 ? 0 : capacity/2);
    T[] newData = new T[capacity];
    for (int i=0 ; i<fill ; i++) {
      newData[i] = data[i];
    }
    data = newData;
  }
}
```

```
  private void Expand (int need) {
    while (fill+need>capacity) {
      capacity = (capacity==0 ? 1 : capacity*2);
      T[] newData = new T[capacity];
      for (int i=0 ; i<fill ; i++) {
        newData[i] = data[i];
      }
      data = newData;
    }
  }
}
```

A.15 Collections

Solution for exercise 19.1: CPR Registry

```
class Person
{
  private string Name { get; set; }
  private int    Age { get; set; }
  private string Cpr { get; set; }

  public Person (string name, int age, string cpr) {
    this.Name = name;
    this.Age  = age;
    this.Cpr  = cpr;
  }

  public override string ToString () {
    return "[Person name='"+Name+"' age='"+Age+"' cpr='"+Cpr+"']";
  }

  public static void Main (string[] args) {
    // subexercise 2
    List<Person> ps = new List<Person>();
    ps.Add(new Person("Emma" , 1, "010101-0101"));
    ps.Add(new Person("Ella" , 2, "020202-0202"));
    ps.Add(new Person("Ida"  , 3, "030303-0303"));
    ps.Add(new Person("Mynte", 4, "040404-0404"));
    ps.Add(new Person("Tinka", 5, "050505-0505"));

    // subexercise 3
    foreach (Person p in ps) {
      if (p.Cpr == "010101-0101") {
        Console.WriteLine(p);
      }
    }

    // subexercise 4
    Dictionary<string, Person> reg = new Dictionary<string, Person>();
    foreach (Person p in ps) {
      reg.Add(p.Cpr, p);
```

```
    }

    // subexercise 5
    Console.WriteLine(reg["010101-0101"]);
  }
}
```

Solution for exercise 19.2: Nondestructive Sets

Subtask 1

```
interface NondestructiveSet<T> : ISet<T>
{
  NondestructiveSet<T> Intersection (NondestructiveSet<T> other);
  NondestructiveSet<T> Union (NondestructiveSet<T> other);
  NondestructiveSet<T> Difference (NondestructiveSet<T> other);
}
```

Subtask 2

```
class NondestructiveHashSet<T> : HashSet<T>, NondestructiveSet<T>
{
  public NondestructiveHashSet () { }

  public NondestructiveHashSet (T[] elements)
  {
    foreach (T element in elements) {
      Add(element);
    }
  }

  public NondestructiveSet<T> Intersection (NondestructiveSet<T> other) {
    NondestructiveSet<T> result = new NondestructiveHashSet<T>();
    result.UnionWith(this);
    result.IntersectWith(other);
    return result;
  }

  public NondestructiveSet<T> Union (NondestructiveSet<T> other) {
    NondestructiveSet<T> result = new NondestructiveHashSet<T>();
    result.UnionWith(this);
    result.UnionWith(other);
    return result;
  }

  public NondestructiveSet<T> Difference (NondestructiveSet<T> other) {
    NondestructiveSet<T> result = new NondestructiveHashSet<T>();
    result.UnionWith(this);
    result.ExceptWith(other);
```

```
    return result;
  }

  public override string ToString () {
    return "{" + string.Join(", ", this) + "}";
  }
}
```

Subtask 3

```
class NondestructiveHashSetTest
{
  static void Main (string[] args) {
    int[] asArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int[] bsArray = { 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 };
    NondestructiveSet<int> a = new NondestructiveHashSet<int>(asArray);
    NondestructiveSet<int> b = new NondestructiveHashSet<int>(bsArray);

    NondestructiveSet<int> a_intersection_b = a.Intersection(b);
    NondestructiveSet<int> b_intersection_a = b.Intersection(a);
    NondestructiveSet<int> a_union_b = a.Union(b);
    NondestructiveSet<int> b_union_a = b.Union(a);
    NondestructiveSet<int> a_difference_b = a.Difference(b);
    NondestructiveSet<int> b_difference_a = b.Difference(a);

    Console.WriteLine("Intersections:");
    Console.WriteLine(" • A   B: " + a_intersection_b);
    Console.WriteLine(" • B   A: " + b_intersection_a);

    Console.WriteLine("");
    Console.WriteLine("Unions:");
    Console.WriteLine(" • A U B: " + a_union_b);
    Console.WriteLine(" • B U A: " + b_union_a);

    Console.WriteLine("");
    Console.WriteLine("Differences:");
    Console.WriteLine(" • A - B: " + a_difference_b);
    Console.WriteLine(" • B - A: " + b_difference_a);
  }
}
```

Subtask 4

Not handed out.

A.16 Input and Output

Solution for exercise 21.1: Command Line Arguments

```
public class CLA
{
  public static void Main (string[] args) {
    // guard: command line arguments
    if (args.Length != 2) {
      Console.WriteLine("Syntax: dotnet run FIRST_INTEGER SECOND_INTEGER");
      Console.WriteLine("        dotnet run 42 56");
      return;
    }

    try {
      int a = int.Parse(args[0]);
      int b = int.Parse(args[1]);

      Console.WriteLine(a + b);
    } catch (FormatException e) {
      Console.WriteLine("Eww, that's nasty!\n" + e.Message);
    }
  }
}
```

Solution for exercise 21.2: Keyboard Input

The following solution hands over the heavy lifting to the standard library of C#. This is done by the call to the static `Reverse` method in `ReverseString`.

```
public class Reverse
{
  static string ReverseString (string? input) {
    if (input==null) return "";
    char[] charArray = input.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
  }

  public static void Main (string[] args) {
    Console.Write("Please enter a text: ");
    string? line = Console.ReadLine();
    string reversed = ReverseString(line);
    Console.WriteLine(reversed);
  }
}
```

Solution for exercise 21.3: Writing to File

```
public class Months
{
  static string[] months = {
```

```
    "Januar",
    "Februar",
    "Marts",
    "April",
    "Maj",
    "Juni",
    "Juli",
    "August",
    "September",
    "Oktober",
    "November",
    "December",
  };

  static bool Write (string filename) {
    try {
      using (StreamWriter writer = new StreamWriter(filename)) {
        for (int i=0 ; i<months.Length ; i++) {
          writer.WriteLine((i + 1) + ":" + months[i]);
        }
      }

      return true;
    } catch (IOException) {
      return false;
    }
  }

  public static void Main (string[] args) {
    bool success = Write("output.txt");
    Console.WriteLine("Operation was" + (success ? " " : " not ") + "successful");
  }
}
```

After execution, the result can be verified by opening the new file `output.txt`.

At compile time, the C# compiler cannot know whether the file we want to write to exists, or for that matter whether we are allowed to write to it. It does not even know which user the program will be executed as or which rights this user has at runtime. Therefore, one of two exceptions can be thrown at runtime: `FileNotFoundException` and `AccessDeniedException`. Both of these are subtypes of `IOException` and – as we want to treat those cases the same – we can therefore just *catch* this one.

In the class variable `months`, the month number is implicitly represented through the index of the array. In this solution, I have chosen to store it explicitly to disk, even though that is not strictly the intention. To get a file format that is easy for humans to understand, I have *offset* the index of the array so that January gets the usual value 1.

Solution for exercise 21.4: Reading from File

```
public class Months
```

```
{
  static string[] months = new string[12];

  static bool Read (string filename) {
    try {
      StreamReader reader = new StreamReader(filename);

      string? line;
      while ((line = reader.ReadLine()) != null) {
        string[] pair = line.Split(':');
        int i;

        // guard: length of pair
        if (pair.Length != 2) {
          Console.WriteLine("Too many elements in line: " + line);
          continue;
        }

        try {
          i = int.Parse(pair[0]);
        } catch (Exception) {
          Console.WriteLine("Unable to parse index: " + pair[0]);
          continue;
        }

        // guard: index out of range
        if (i < 1 || i > 12) {
          Console.WriteLine("Month index out of range: " + i);
          continue;
        }

        months[i-1] = pair[1];
      }

      reader.Close();
      return true;
    } catch (FileNotFoundException) {
      return false;
    }
  }

  public static void Main (string[] args) {
    bool success = Read("output.txt");
    Console.WriteLine("Operation was" + (success ? " " : " not ") + "successful");

    Console.WriteLine("Month table:");
    for (int i=0 ; i<months.Length ; i++) {
      Console.WriteLine(" - "+(i + 1)+": "+months[i]);
    }
  }
```

```
}
```

This code relies on the `output.txt` file produced by the code from section 96.

The C# compiler cannot know at compile time whether the file we want to read from exists, and therefore it chooses to produce code that at runtime may *throw* a `FileNotFoundException` exception. We have to handle this.

The starting point for this task is that the file is encoded in a certain way and we write logic that extracts data based on an assumption that this encoding is respected. This assumption will not always be correct, and it is an attractive quality of our logic to respond appropriately to this situation. We say that the code is robust (with respect to this input).

Solution for exercise 21.5: Highscores

The data structure we want to represent is a sequence of score×player sorted by score. We could introduce a `ScoreEntry` class that contains a score and a player, but to keep the problem simple (e.g., by not having to consider what a player is), the following solution chooses to represent this data as two separate arrays: one for scores and one for player names. These arrays are of equal length and an index designates a combination of score and player.

```
public class Highscores
{
  private int count;
  private string filename;
  private string[] players;
  private int[] scores;

  public Highscores (string filename, int count) {
    this.count = count;
    this.filename = filename;
    this.players = new string[count];
    this.scores = new int[count];
  }

  public bool Save () {
    try {
      StreamWriter writer = new StreamWriter(filename);

      for (int i = 0; i < count; i++) {
        if (players[i] != null) {
          writer.WriteLine(scores[i] + "," + players[i]);
        }
      }

      writer.Close();
      return true;
    }
    catch (IOException) {
      return false;
    }
```

```
}

public bool Load () {
  try {
    StreamReader reader = new StreamReader(filename);

    int i = 0;
    while (!reader.EndOfStream) {
      // guard: filled up all spaces
      if (i >= count) break;

      string? line = reader.ReadLine();
      if (line==null) throw new Exception("Parse error!");
      string[] pair = line.Split(',');

      // guard: length of pair
      if (pair.Length != 2) {
        Console.WriteLine("Too many elements in line: " + line);
        return false;
      }

      // read score
      try {
        scores[i] = int.Parse(pair[0]);
      } catch (Exception) {
        Console.WriteLine("Unable to parse index: " + pair[0]);
        return false;
      }

      // read player
      players[i] = pair[1];

      i++;
    }

    reader.Close();
    return true;
  } catch (FileNotFoundException) {
    return false;
  }
}

public bool NewScore (int score, string player) {
  int i;

  // locate correct place
  for (i=0 ; i<count&&scores[i]>score ; i++) { }

  // guard: not a top score
  if (i == count) return false;
```

```
    // make space
    for (int j=count-2 ; j>=i ; j--) {
      scores[j + 1] = scores[j];
      players[j + 1] = players[j];
    }

    // insert
    scores[i] = score;
    players[i] = player;

    return true;
  }

  public bool ValidIndex (int i) {
    return i >= 0 && i < count && players[i] != null;
  }

  public int GetScore (int i) {
    if (!ValidIndex(i)) return -1;

    return scores[i];
  }

  public string GetPlayer (int i) {
    if (!ValidIndex(i)) throw new Exception("Unknown player "+ i);

    return players[i];
  }

  public static void Main (string[] args) {
    Highscores hs = new Highscores("highscores.csv", 10);
    hs.Load();

    if (hs.ValidIndex(0)) {
      hs.NewScore(hs.GetScore(0) - 1, hs.GetPlayer(0) + "-");
      hs.NewScore(hs.GetScore(0) + 1, hs.GetPlayer(0) + "+");
    } else {
      hs.NewScore(42, "Marvin");
    }

    hs.Save();
  }
}
```

Solution for exercise 21.6: CSV Files

A cell cannot contain the text "`1,2`" because the comma represents a separation of cells. If you try to do so anyway, you end up inserting two cells in the same row; one with the text "`1`" and one with the text "`2`".

```
public class Matrix
{
  double[][] Data { get; set; }

  public Matrix (string filename) {
    Data = new double[0][]; // to fool the compiler into not giving us a wrong warning
    bool success = Load(filename);
    if (!success) {
      Console.WriteLine("Oops, this is not gonna be pretty!");
      throw new Exception("Unable to load "+filename);
    }
  }

  public Matrix (int x, int y) {
    Data = new double[y][];
    for (int i=0; i<y ; i++) {
      Data[i] = new double[x];
    }
  }

  public int GetHeight () {
    return Data.Length;
  }

  public int GetWidth () {
    return Data[0].Length;
  }

  public bool Save (string filename) {
    try {
      StreamWriter writer = new StreamWriter(filename);

      foreach (double[] row in Data) {
        for (int i=0 ; i<row.Length ; i++) {
          double cell = row[i];
          writer.Write((i == 0 ? "" : ",") + cell);
        }
        writer.WriteLine();
      }

      writer.Close();
      return true;
    } catch (IOException) {
      return false;
    }
  }

  private bool Load (string filename) {
    try {
      List<string> lines = new List<string>();
```

```
      // read file
      StreamReader reader = new StreamReader(filename);
      {
        string? line;
        while ((line = reader.ReadLine()) != null) {
          lines.Add(line);
        }
      }
      reader.Close();

      // parse contents of file
      int x = 0;
      int y = 0;
      try {
        Data = new double[lines.Count][];
        for (y=0 ; y<lines.Count ; y++) {
          string line = lines[y];
          string[] cells = line.Split(',');
          Data[y] = new double[cells.Length];
          for (x=0 ; x<cells.Length ; x++) {
            Data[y][x] = double.Parse(cells[x]);
          }
        }
      } catch (FormatException) {
        Console.WriteLine("Unable to parse x=" + x + " y=" + y);
        return false;
      }
    } catch (FileNotFoundException) {
      return false;
    }

    return true;
  }

  public static void Main (string[] args) {
    string m1_filename = "m1.csv";
    string m2_filename = "m2.csv";

    Matrix m1 = new Matrix(4, 6);
    m1.Save(m1_filename);

    Matrix m2 = new Matrix(m1_filename);
    for (int i=0 ; i<Math.Min(m2.GetHeight(), m2.GetWidth()) ; i++) {
      m2.Data[i][i] += 1;
    }
    m2.Save(m2_filename);
  }
}
```

This solution has a number of weaknesses:

1. The `Main` method is not informed of a possible lack of success during loading of data in the constructor that takes a `String` as argument. Ideally, one would make sure that the constructor returns a `null` object, but this is not possible in C#. One could throw an exception, but then we would have to handle this and that is not an attractive solution either. Instead, the code should be refactored so that the `Load` method is separate from the constructor. This is considered good style.
2. A matrix is by definition rectangular. It is possible to construct a CSV file that the constructor for `Matrix` will interpret into a data structure that is not rectangular. For example, try deleting the last cell in the top row. The method `Load` will happily load this data structure, but the data structure will not be consistent with the model the class is designed to represent. This means that the class is not robust to input data of the wrong structure.
3. Similarly, `Load` will not react sensibly to a cell that does not contain anything it can parse to a `double`. This means that the class is not robust to input data of the wrong format.
4. There is no check in either `GetValue` or `SetValue` for whether you are trying to index into the data structure outside the dimensions of the matrix.

Solution for exercise 21.7: Filesystem Traversal

```
public class FSTraverse
{
  static void Traverse (List<string> result, DirectoryInfo directory, string extension) {
    foreach (FileInfo file in directory.GetFiles()) {
      if (file.Extension == extension) {
        result.Add(file.FullName);
      }
    }

    foreach (DirectoryInfo subDirectory in directory.GetDirectories()) {
      Traverse(result, subDirectory, extension);
    }
  }

  public static void Main (string[] args) {
    // guard: command line arguments
    if (args.Length != 1) {
      Console.WriteLine("Syntax: dotnet run PATH");
      Console.WriteLine("        dotnet run .");
      return;
    }

    string path = args[0];

    Console.WriteLine("Searching "+path);
    List<string> matches = new List<string>();
```

```
    Traverse(matches, new DirectoryInfo(path), ".csv");

    Console.WriteLine("Matches:");
    foreach (string match in matches) {
      Console.WriteLine(" - " + match);
    }
  }
}
```

Solution for exercise 21.8: Shell

```
public class Shell
{
  public static void Main (string[] args) {
    bool done = false;

    while (!done) {
      Console.Write("$ ");

      string? line = Console.ReadLine();
      if (line==null) continue;
      string[] commandArgs = line.Split(' ');
      string command = commandArgs[0];

      switch (command) {
        case "exit":
          done = true;
          break;
        case "echo":
          Console.WriteLine(line);
          break;
        case "time":
          Console.WriteLine(DateTime.Now);
          break;
        default:
          Console.WriteLine("Sorry, I don't quite know what to do with '"+line+"' :-(");
          break;
      }
    }
  }
}
```

Test of funktionality:

```
$ help
Sorry, I don't quite know what to do with 'help' :-(
$
Sorry, I don't quite know what to do with '' :-(
```

```
$ echo
echo
$ echo a b
echo a b
$ time
Sun Sep 13 20:16:11 CEST 2020
$ exit
```

Solution for exercise 21.9: Turtle Interpreter

The following solution is based on a very simple language. There is a command to take one step in each direction (north, north-east, east, south-east, south, south-west, west, north-west), a command to enable drawing and another to disable drawing. The sequence of commands needed to draw the image is thus relatively long:

```
public class Canvas
{
  int width;
  int height;
  bool[][] data;

  public Canvas (int width, int height) {
    this.width = width;
    this.height = height;
    this.data = new bool[height][];
    for (int i=0 ; i<height ; i++) {
      this.data[i] = new bool[width];
    }
  }

  public bool GetValue (int x, int y) {
    return data[y][x];
  }

  public void SetValue (int x, int y, bool v) {
    data[y][x] = v;
  }

  public void Print () {
    Console.Write("+");
    for (int x=0 ; x<width ; x++) Console.Write("-");
    Console.WriteLine("+");

    for (int y=0 ; y<height ; y++) {
      Console.Write("|");
      for (int x=0 ; x<width ; x++)
        Console.Write((GetValue(x, y) ? "X" : " "));
      Console.WriteLine("|");
    }
```

```
    Console.Write("+");
    for (int x=0 ; x<width ; x++) Console.Write("-");
    Console.WriteLine("+");
  }
}
```

```
using System;
using System.IO;

public class Turtle
{
  int x = 0;
  int y = 0;
  bool draw = false;
  Canvas canvas;

  public Turtle (Canvas canvas) {
    this.canvas = canvas;
  }

  public void Raise () {
    draw = false;
    Draw();
  }

  public void Lower () {
    draw = true;
    Draw();
  }

  public void GoN () {
    y -= 1;
    Draw();
  }

  public void GoS () {
    y += 1;
    Draw();
  }

  public void GoW () {
    x -= 1;
    Draw();
  }

  public void GoE () {
    x += 1;
    Draw();
  }
```

```
public void GoNE () {
  x += 1;
  y -= 1;
  Draw();
}

public void GoSE () {
  x += 1;
  y += 1;
  Draw();
}

public void GoSW () {
  x -= 1;
  y -= 1;
  Draw();
}

public void GoNW () {
  x -= 1;
  y -= 1;
  Draw();
}

public void Draw () {
  canvas.SetValue(x, y, canvas.GetValue(x, y) || draw);
}

public static void Main (string[] args)
{
  Canvas canvas;
  Turtle turtle;

  // guard: command line arguments
  if (args.Length != 3) {
    Console.WriteLine("Syntax: dotnet run WIDTH HEIGHT FILENAME");
    Console.WriteLine("        dotnet run 11 12 ../house.turtle");
    return;
  }
  int width  = int.Parse(args[0]);
  int height = int.Parse(args[1]);
  string filename = args[2];

  canvas = new Canvas(width, height);
  turtle = new Turtle(canvas);

  // open file
  StreamReader file = new StreamReader(filename);
  string? line;
  while ((line = file.ReadLine()) != null) {
```

```
      switch (line) {
        case "go north":
          turtle.GoN();
          break;
        case "go south":
          turtle.GoS();
          break;
        case "go east":
          turtle.GoE();
          break;
        case "go west":
          turtle.GoW();
          break;
        case "go north-east":
          turtle.GoNE();
          break;
        case "go south-east":
          turtle.GoSE();
          break;
        case "go south-west":
          turtle.GoSW();
          break;
        case "go north-west":
          turtle.GoNW();
          break;
        case "pen up":
          turtle.Raise();
          break;
        case "pen down":
          turtle.Lower();
          break;
        default:
          Console.WriteLine("Warning. Unrecognized command '"+line+"'. Skipping ...");
          break;
      }
    }
    file.Close();

    // print result
    canvas.Print();
  }
}
```

Solution for exercise 21.10: Sūdoku Visualization

So solution provided.

A.17 Tooling

Solution for exercise 22.1: Timing

Subtask 1:

```
class Timing
{
  static double initialValue = 1.0000001;

  static double Fun (double x, double y) {
    if (y <= 1) {
      return x;
    } else {
      return Fun(x, y - 1) * Fun(x, y - 1);
    }
  }

  public static void Main (string[] args) {
    for (double i=1 ; i<=32 ; i++) {
      long t0 = DateTimeOffset.Now.ToUnixTimeMilliseconds();
      double f = Fun(initialValue, i);
      long t1 = DateTimeOffset.Now.ToUnixTimeMilliseconds();

      Console.WriteLine("fun(" + initialValue + "," + i + ") = " + f +
                        " (calculation took " + (t1 - t0) + "ms)");
    }
  }
}
```

Subtask 2:

Yes, when `Fun(x,-1)` is called in this implementation, the value that `x` is bound to is returned.

If the condition in `Fun` had tested for equality, a negative `y` value would have ended up in the `else` branch and it would not calculate `fun(x,y)` correctly. Then, when `fun(1.0000001,-1)` is calculated, it does so by first calculating `fun(1.0000001, -2)`. This would be done by first calculating `fun(1.0000001,-3)` and so on. This sequence only stops when the `y` value reaches `1`, which (in practice) does not happen. This implementation is not correct for negative numbers (or large positive numbers for that matter).

Subtask 3:

But this is not the only reason why it is a bad algorithm. If we look a little closer at what happens if we have to calculate `fun(5,4)`, then `fun(5,3)` is calculated twice. For each of these, `fun(5,2)` is calculated twice (that is 4 times in total), and for each of these, `fun(5,1)` is "calculated" twice (that is 8 times in total). This would be even worse with a larger exponent. This would have been fine if the result of these calculations depended on something external that could not be predicted, but in this case it is obvious that the calculation of `fun(5,3)` results in the same value every time. One would therefore see a huge improvement in performance by calculating `fun(x,y-1)` once and multiplying this value by itself.

Solution for exercise 22.2: Date

```
class DateTest {
  static void Main(string[] args) {
    for (long elapsed=1000 ; elapsed<=1000000 ; elapsed*=10) {
      DateTime d = System.DateTimeOffset.FromUnixTimeMilliseconds(elapsed).DateTime;
      Console.WriteLine(d.ToString());
    }
  }
}
```

Executing the code yields the following printout:

```
01/01/1970 00.00.01
01/01/1970 00.00.10
01/01/1970 00.01.40
01/01/1970 00.16.40
```

Measured in seconds, there is a factor of 10 between each of these lines. This means that C#'s timing is done in milliseconds and starts on January 1, 1970 at 01:00 in the night.

Solution for exercise 22.3: Terminal Use

Subtask 1

```
while (true) {
  Console.WriteLine(Console.WindowWidth+"x"+Console.WindowHeight);
  Thread.Sleep(1000);
}
```

Subtask 2

```
char character;

while (true) {
  int width  = Console.WindowWidth;
  int height = Console.WindowHeight;

  for (int y=0 ; y<height ; y++) {
    for (int x=0 ; x<width ; x++) {
      if (y==height/2 && x==width/2) {
        character = '+';
      } else if (y==height/2) {
        character = '-';
      } else if (x==width/2) {
        character = '|';
      } else {
        character = ' ';
```

```
      }

      Console.Write(character);
    }
    Console.WriteLine("");
  }

  Thread.Sleep(1000);
}
```

Subtask 3

```
World world = new World();
Ball ball = new Ball(world, 1, 1);
world.Run();

public class World {
  List<Ball> balls;
  public int width;
  public int height;
  char[,] cells;

  public World () {
    this.balls = new List<Ball>();
    this.width = Console.WindowWidth;
    this.height = Console.WindowHeight;
    this.cells = new char[height, width];
  }

  public void AddBall (Ball ball) {
    balls.Add(ball);
  }

  void Clear () {
    for (int y=0 ; y<height ; y++) {
      for (int x=0 ; x<width ; x++) {
        cells[y, x] = ' ';
      }
    }
  }

  public void PlaceBall (int x, int y) {
    cells[y, x] = 'O';
  }

  void Render () {
    for (int y=0 ; y<height ; y++) {
      for (int x=0 ; x<width ; x++) {
        Console.Write(cells[y, x]);
      }
```

```
      Console.WriteLine("");
    }
  }

  void Step () {
    foreach (Ball ball in balls) {
      ball.Step();
    }
  }

  public void Run () {
    while (true) {
      Render();
      Clear();
      Step();
      Thread.Sleep(100);
    }
  }
}

public class Ball {
  World world;
  int x, y;
  int dx, dy;

  public Ball (World world, int dx, int dy) {
    this.world = world;
    this.x = 0;
    this.y = 0;
    this.dx = dx;
    this.dy = dy;
    world.AddBall(this);
  }

  public void Step () {
    if ((dx>0 && x+dx>=world.width)  || (dx<0 && x+dx<0)) dx *= -1;
    if ((dy>0 && y+dy>=world.height) || (dy<0 && y+dy<0)) dy *= -1;

    x += dx;
    y += dy;

    world.PlaceBall(x,y);
  }
}
```

Note: This implementation gracefully extends to multiple balls.

Subtask 4

No reference solution provided

A.18 Debugging

Solution for exercise 23.1: Locate the Mistake

The error occurs when `points` has the value 50. Then both conditions will evaluate to `true` and this will cause both branches to be evaluated with the text "`You PASSEDFAILED!`" as a result.

The error is due to an ambiguity about whether 50 points is passed or failed. Once this ambiguity is resolved, one of the conditions can be fixed. Although this will result in correct code, it still has a significant weakness: two independent branches are used in the code for an either-or situation. This means that correctness over time depends on a programmer updating both conditions correctly when the limit changes from 50 to something else (e.g., 40). A better solution would be to have a single condition in an *if-then-else* construction. This effectively defines the second block to be evaluated in every case where the first one is not evaluated. Thus, it is guaranteed that exactly one of the blocks is evaluated.

Solution for exercise 23.2: Correctness

When we subtract one random number from another, we get a random number. The average of a series of random numbers approaches the mean of the generator's distribution. So when we take the average of 20 random numbers 10 times in a row, the results should be close to each other, relative to the width of the distribution.

This is not fit the case that we are observing. In addition to the fluctuating averages, the standard deviation seems to be relatively small. This suggests that something is wrong. It turns out that the test *is* wrong. The problem comes from the calculation of `d[i]` where it is always the first element of `a` that is subtracted from the `i`th element of `b`. This means that in practice, the elements from `b` are offset by the value of `d[1]` (which is constant for all `i`) instead of offset by the random number in `a[i]`.

After correcting this, we get a smaller variation in the mean, and larger standard deviations:

```
[  101    6   43   21   10  -60    3  -94  -74   61  -80   81   40  -49 -124 -179   95 -101  -51 -127    4   85   98  -75
    51 -118 -110   26   33   33  136    1  -56  -56    6   93  -12   44  131    8  -64  114   43 -148   94   14   95  105 ] avg=   2 stdev=80.76
[   20 -195   43  -32   -2 -128   58  102    1  -96   77  -62  -91  -33   33  -78  -40   23  -36 -113 -122  -79  -99  -89
  -104  -40  -19    5   29  -66  -86 -107  -13   27  -36  -44  110   19  -39  103   43   57   -7   16   -1  -26   -4   42 ] avg= -22 stdev=66.33
[  -99  143  -96  -12    7  -54   77   98  176   35   11 -118   67   95  -31   59   64  102  -73   48 -191   26  -11  -55
   -54  -87    1  -96    8   24  -51   76   41   -3  -84    8   74  -22  -67   19   11 -108   -6  105   45  138  -20  156 ] avg=   7 stdev=78.90
[  -48 -139 -116  -64   20   89  -16  -62  -45   30  -44   39   29 -166   54  -84   84  114 -135   34    4    3  -93   -1
    87   10 -139 -109  128  -38   11  158  140  -57    5  -57  -53   44   20   87   10  -32 -100   45   55  -55  -72  124 ] avg=  -6 stdev=79.45
[   87  -76   25 -129   45   -9  -17   98  -97   14   31    3   35   74   41   -3   84  -45  140   64  -73 -156   72    1
   -11   99  -20  -11   40   64  157  -18   10  129  -71  -89  -68  -85  -49 -101  -18  -45  -70  -54   13  -31  -95  -47 ] avg=  -3 stdev=71.66
[  -83 -118  -47   39 -114   30 -137 -142   50   33  -44  100   13  146  -18   18  -19  -60  -77   -7  -10  -95   77   99
    30  -23  -12   43 -107  -43   39    1   68  148   34   15  -19    2  110   55  -63  191   48 -114  107   61   85  -78 ] avg=   4 stdev=78.36
[    3   33  -66   30   79   33   54  105   65   -7  -26   64 -101   -6   26  -69   82  111    4  -14  -11   66   86   26
   135   22  -32    5  -76   47   53  -52 -129  107  138  110   42  -47  -55   59   48   -6   81  127    1   64    8  -85 ] avg=  23 stdev=64.04
[  -50 -130   55   -2  132 -116   97  -22  -55 -128   23   56  -79  -67  -14   32  106   28  -56  148   68   30   39 -117
    65   51   35  -43   20   18   -8   35  -71  116  -97  -58    6 -169  112 -185   -4  -47  143 -132   31   20  -51 -112 ] avg=  -7 stdev=82.75
[ -108  -75  -69   53 -115   -6  -61   -2  -44   -1    1  -23   -1   -6   10  -47  -29  -44  -66  -55  -14  143   64   49
   -14 -104  139   55 -135  -33   42  -26   24  -38  -29   57   20   58  -40   43    3  -24    4  -60   28 -100  -95  -75 ] avg= -15 stdev=59.59
[   11  -37  -80 -135  -10  -15  -85 -138    8  139   99   54  -24  -43  -14   26  -57   41   81 -117   29  -44 -102 -133
   -88  -10  -33  -74  112  -31  -16  129  -26  101   24  -67  -99 -107   98  -28  -50   55   54  -39  -13  -24  -55  -93 ] avg= -17 stdev=71.40
```

Solution for exercise 23.3: Sum

The `sum` method contains three bugs:

1. The method operates on indices instead of the elements that are at the index positions in `numbers`.

2. The `for`-loop condition is off-by-one such that the method goes one step too far. This results in an `IndexOutOfRangeException`.
3. The loop variable `i` starts with the value 1, which means that the first element in `numbers` is skipped. The calculated result is therefore the value of `numbers[0]` too low.

These errors are fixed in the following code:

```
public class Adder
{
  public static int Sum (int[] numbers) {
    int result = 0;
    for (int i=0 ; i<numbers.Length ; i++) {
      result += numbers[i];
    }
    return result;
  }

  public static void Main (String[] args) {
    int[] testcase1 = {1, 3, 5, 2, 3, 7};
    int[] testcase2 = {5, 6, 2, 3};

    Console.WriteLine(Sum(testcase1));
    Console.WriteLine(Sum(testcase2));
  }
}
```

Solution for exercise 23.4: Happiness

The trick here is that C# – like almost all other programming languages – will evaluate boolean expressions lazily and short-circuit. This means that with the boolean operators AND and OR, you only evaluate the right-hand side if it has the possibility to affect the result of the full expression. For example, the value of the expression `false && SomeMethod()` will be false regardless of what `SomeMethod` evaluates to, and C# will therefore choose *not* to call `SomeMethod`. Similarly, `true || SomeMethod()` will always evaluate to true and C# will thus not call `SomeMethod`.

This is often used for some *smart* constructions. For instance, you can guard against problematic situations such as method calls on `null` objects:

```
if (person!=null && person.IsAlive()) {
    Console.WriteLine(person.GetName()+" is alive!");
}
```

A.19 Testing

Solution for exercise 24.1: Pin

Subtask 1

A counter `time2pin` keeps track of how many transactions there are before we *have to* ask for a password. If this limit is zero, or the limit amount `CRITICAL_AMOUNT` has been exceeded, then this variable is reset to the starting point (`MAX_TRANSACTIONS`).

We then check whether the critical amount has been exceeded and if this is not the case, we count `time2pin` down by one and check whether the result is zero. If one of these conditions applies, then a PIN code must be requested and the method returns `true`.

Subtask 2

The amount `amount` has 350 and 351 as limit values because the rule is whether the amount is *greater than* 350. 350 is the largest amount where this is not the case and 351 is the smallest amount where it is the case.

The variable `time2pin` is a little more complicated. In the code we can see that the value of `MAX_TRANSACTIONS` is assigned. This makes this value a limit value. We can also see that `time2pin` decreases with time, is integer and is compared to zero. Therefore -1, 0 and 1 (zero and neighbors) are also limit values.

Subtask 3

```
public class Pin
{
  const int CRITICAL_AMOUNT = 350;
  const int MAX_TRANSACTIONS = 4;
  private int time2pin = MAX_TRANSACTIONS;

  bool Expend (int amount) {
    if (time2pin==0 || amount>CRITICAL_AMOUNT) {
      time2pin = MAX_TRANSACTIONS;
    }
    return amount > CRITICAL_AMOUNT || --time2pin == 0;
  }

  public static void Main(string[] args) {
    int[] amount_edges = { 350, 351 };
    int[] transact_edges = { -1, 0, 1 };

    foreach (int amount in amount_edges) {
      foreach (int transact_count in transact_edges) {
        Pin pin = new Pin();
        bool result = true;
        for (int i=0; i<MAX_TRANSACTIONS-transact_count ; i++) {
          result = pin.Expend(amount);
        }
        Console.WriteLine("a=" + amount + " t=" + transact_count + " r=" + result);
      }
    }
  }
}
```

Executing this program prints the following:

```
a=350 t=-1 r=false
a=350 t=0 r=true
a=350 t=1 r=false
a=351 t=-1 r=true
a=351 t=0 r=true
a=351 t=1 r=true
```

From this we can see that all test cases with a `amount` of 351 give us a `true` value, and that the test case where `amount` is 0 and `time2pin` is 0 gives a `true` value. This represents correct behavior.

These combinations of boundary values represent the most obvious potential for problems. We cannot conclude from this analysis whether `Expend` works under all combinations of input (`amount`) and state (`time2pin`), but it is a good start.

Subtask 4

If a random number generator had been involved, we would have had to run our test cases enough times to be able to argue that we have statistical evidence to draw a conclusion.

A.20 Software Architecture

Solution for exercise 25.1: Inventory

No solution provided.

A.21 Paradigms

Solution for exercise 26.1: Livebook Installation

No solution provided.

Solution for exercise 26.2: Elixir Introduction

No solution provided.

Solution for exercise 26.3: Demo Repository

No solution provided.

Solution for exercise 26.4: Filtering of Even Numbers

Solution with guard:

```elixir
defmodule Filter do
  def filter([]) do
    []
  end
  def filter([head|tail]) when is_integer(head) do
    if rem(head,2)==0 do
      [head|filter(tail)]
    else
```

```
      filter(tail)
    end
  end
end

[1,5,-12,-845,23,-1,14,16,2,35,-53,243,4,2,6,-1,-42]
|> Filter.filter()
```

Solution for exercise 26.5: Elementary Equality

```
defmodule ElemEq do
  def compare([], []) do
    []
  end
  def compare([a_head|a_tail], [b_head|b_tail]) do
    [a_head==b_head|compare(a_tail,b_tail)]
  end
end

test_suite = [
  {[], []},
  {[1,2,3], [1,2,3]},
  {[1,2,3], [2,3,1]},
  {[1.0], [1]},
  {["hello", "world"], ["goodbye", "world"]},
  {[1, "a", 2, "b"], [1, "a", 2, "B"]},
]

Enum.map(test_suite, fn {a, b} -> ElemEq.compare(a, b) end)
```

Using higher-order functions it is a bit easier:

```
test_suite = [
  {[], []},
  {[1,2,3], [1,2,3]},
  {[1,2,3], [2,3,1]},
  {[1.0], [1]},
  {["hello", "world"], ["goodbye", "world"]},
  {[1, "a", 2, "b"], [1, "a", 2, "B"]},
]

Enum.map(test_suite, fn {a, b} -> Enum.zip(a, b) |> Enum.map(fn {a,b} -> a==b end) end)
```

But why does this work?

Solution for exercise 26.6: The Fibonacci Sequence

Subtask 1

```
defmodule FibPrimitive do

  def calc(0) do
    0
  end

  def calc(1) do
    1
  end

  def calc(n) when n>0 do
    calc(n-1) + calc(n-2)
  end

end
```

Subtask 2

To calculate fib(n) where $n = 1$, fib(1) must be evaluated exactly once. If $n = 3$, fib(1) must be evaluated exactly twice: fib(3) is indeed equal to fib(2) + fib(1), and each of these requires that fib(1) be evaluated once. Each time n is increased by one, the number of times fib(1) must be evaluated doubles. In other words, we have a *complexity* that follows 2^n. It quickly becomes a problem.

Subtask 3

```
defmodule FibThoughtfull do

  def calc(n) when n>=0 do
    calc(0, 1, n)
  end

  defp calc(last, _last_last, 0) do
    last
  end

  defp calc(last, last_last, ttl) when ttl>0 do
    calc(last+last_last, last, ttl-1)
  end

end
```

The difference here is that nothing is calculated twice.

Subtask 4

According to the resulting plot, it is quite clear that the solution from subtask 3 is massively superior. This is something you often see within algorithms.

Solution for exercise 26.7: Pipelines

Subtask 1

```
defmodule Algorithm do
  def solve(humans) do
    solve(humans, None)
  end
```

```
  defp solve([human], best) do
    incorporate(best, human)
  end
  defp solve([human|rest], best) do
    new_best = incorporate(best, human)
    solve(rest, new_best)
  end

  defp incorporate(best, contender) do
    if contender.sex==:male and contender.age>=18 and (best==None or contender.age<best) do
      contender.age
    else
      best
    end
  end
end
```

Subtask 2

```
humans
|> Enum.filter(fn human -> human.sex == :male and human.age >= 18 end)
|> Enum.map(fn human -> human.age end)
|> Enum.min()
```

The higher order functions are actually implemented via recursion, but we don't see that at this level of abstraction.

Subtask 3

In subtask 1, the recursion is separated from the update of the best value. This is a good thing as it makes the code faster to navigate.

In subtask 2, it is clear what role each line has, and there are very few of them. This is Aslak's favorite.

Solution for exercise 26.8: Higher-Order Functions

```
defmodule HigherOrder do
  def map([], _fun) do
    []
  end
  def map([head|tail], fun) do
    [
      fun.(head)
    |
      map(tail, fun)
    ]
  end

  def filter([], _fun) do
    []
  end
```

```
  def filter([head|tail], fun) do
    if fun.(head) do
      [head|filter(tail, fun)]
    else
      filter(tail, fun)
    end
  end
end

test = [1,5,-12,-845,23,-1,14,16,2,35,-53,243,4,2,6,-1,-42]

test
|> HigherOrder.filter(fn value -> value>0 end)
|> HigherOrder.map(fn value -> value+1 end)
```

Solution for exercise 26.9: C♭

Subtask 1

The abstract syntax tree represents the structure of our code. It is constructed from the language's grammar, which is recursively defined[1] with a number of rules for what constitutes a statement, what constitutes an expression, and so on.

Subtask 2

A `while` loop is evaluated by first evaluating a condition. If this evaluates to `false`, our current state is returned. If it evaluates to `true`, the body is evaluated (resulting in a new state) and then we repeat this procedure by recursion.

Solution for exercise 26.10: Airports

Subtask 1

There is no reference solution for this task.

Subtask 2

The function `Req.get!` takes a parameter which must be a string representing a valid URL. It then performs an HTTP GET operation (which means that it downloads the file) and returns a data structure through which (via the key `:body`) the contents of this file can be accessed.

Subtask 3

`index` is a data structure of type `Map` which can be used to look up a column in our CSV data file based on a name. It thus enables us to name columns so that we can refer to the column "latitude" instead of its index.

Subtasks 4 and 5

```
alias MapLibre, as: Ml

# List of countries of various categories
```

[1] https://github.com/aslakjohansen/cflat/blob/main/src/parser.yrl

```
# source: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Country_codes
countries_eu =
  "BE,BG,CZ,DK,DE,EE,IE,EL,ES,FR,GR,HR,IT,CY,LV,LT,LU,HU,MT,NL,AT,PL,PT,RO,DI,DK,FI,SE"
  |> String.split(",")
countries_efta =
  "IS,LI,NO,CH"
  |> String.split(",")
countries_eu_cand =
  "BA,ME,MD,MK,GE,AL,RS,TR,UA"
  |> String.split(",")
countries_enp =
  "AM,BY,AZ,DZ,EG,IL,JO,LB,LY,MA,PS,SY,TN"
  |> String.split(",")
countries_other =
  "AR,AU,BR,CA,CN,CM_X_HK,HK,IN,JP,MX,NG,NZ,RU,SG,ZA,KR,TW,UK,US"
  |> String.split(",")
countries_interest =
  [
    countries_eu,
    countries_efta,
    countries_eu_cand,
    countries_enp,
    countries_other
  ]
  |> List.flatten()

# data source
url = "https://raw.githubusercontent.com/ip2location/ip2location-iata-icao/master/iata-icao.csv"

# fetch data and split it into lines
file =
  url
  |> Req.get!()
  |> Map.get(:body)
  |> String.replace("\",\"", "\"|\"", global: true)
  |> String.replace("\"", "", global: true)
  |> String.split("\r\n")

# split lines into header and data lines
[header_line|contents_lines] = file

# parse the header into a name-to-index map
index =
  header_line
  |> String.split("|")
  |> Enum.with_index()
  |> Map.new()

# parse data lines according to header map to obtain a list of airports
airports =
  contents_lines
  |> Enum.filter(fn line -> line != "" end)
  |> Enum.map(
    fn line ->
      elements =
        line
        |> String.split("|")
      index
      |> Enum.reduce(%{},
        fn {key, i}, acc ->
          Map.put(acc, key,
            case {key, Enum.at(elements, i)} do
              {"latitude", value} -> Float.parse(value) |> elem(0)
```

```
            {"longitude", value} -> Float.parse(value) |> elem(0)
            {_, value} -> value
          end
        )
      end
    )
  end
)

# define list of markers to place on map
markers =
  airports
  |> Enum.filter(fn %{"country_code" => country} -> Enum.member?(countries_interest, country) end)
  |> Enum.map(
    fn %{"latitude" => lat, "longitude" => long, "airport" => _name, "country_code" => country} ->
      [
        {long, lat},
        scale: 0.5,
        color:
          cond do
            # String.contains?(name, "International") -> "red"
            country=="DK" -> "red"
            Enum.member?(countries_eu, country) -> "blue"
            Enum.member?(countries_efta, country) -> "cyan"
            Enum.member?(countries_eu_cand, country) -> "green"
            Enum.member?(countries_enp, country) -> "orange"
            Enum.member?(countries_other, country) -> "white"
            true -> "black"
          end
      ]
    end
  )

# produce map
Ml.new()
|> Kino.MapLibre.add_markers(markers)
```

Subtask 6

"Heathrow Airport" appears on the list of airports, but it is listed as belonging to GB (Great Britain), and GB is not mentioned on these country lists from the EU. Here, UK (United Kingdom) is mentioned. This means that it is filtered out as a result of partial task 4.

Solution for exercise 26.11: World of Zuul

The is no reference solution to this exercise.

Appendix B

Concepts to Avoid

It is natural to supplement a book like this with material from the Internet and other sources. It might even be a good idea, and as you progress through your education it will be critical. However, it is also necessary to be critical of such information. Some of it is helpful, some is not, and some is straight up counterproductive. The online information on C#has a tendency to fall into the latter two categories for novice developers. This is not a matter of correctness, but about limiting your exposure to unnecessary complexity and other ways of shorting the learning process. The following is a non-exclusive list of concepts to avoid:

- ***Threads*** Threads allow parts of your code to happen concurrently. That is, they can be expressed in such way that the execution order is flexible. On a system with multiple hardware threads, this will usually result is some degree of parallelism. Concurrency is hard to reason about, and threads is one of the harder ways of achieving concurrency. If you think that any of the topics of this book are difficult, adding threads to the mix will likely make them impossible.
- ***`var` keyword*** The `var` keyword is used to instruct the compiler to automatically infer types. While that comes with advantages for an experienced developer, it also makes it also removes the visual cues to types. For a novice programmer that is often detrimental to the understanding of the type systems role.

Bibliography

[1] Lisanne Bainbridge. "Ironies of automation". In: *Automatica* 19.6 (1983), pp. 775–779. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(83)90046-8. URL: https://www.sciencedirect.com/science/article/pii/0005109883900468.

[2] Scott Chacon and Ben Straub. *Pro git: Everything you need to know about Git*. English. Second. Apress, 2014. URL: https://git-scm.com/book/en/v2.

[3] Ralph Waldo Emerson. *Self-Reliance: An Excerpt from Collected Essays, First Series*. 1841.

[4] Richard Feynman. "Cargo Cult Science". In: (1974). Caltech's 1974 commencement address. URL: https://calteches.library.caltech.edu/51/2/CargoCult.htm.

[5] Alexander C. Karp and Nicholas W. Zamiska. *The Technological Republic: Hard Power, Soft Belief, and the Future of the West*. London: Penguin Random House, 2025.

[6] Nataliya Kosmyna et al. *Your Brain on ChatGPT: Accumulation of Cognitive Debt when Using an AI Assistant for Essay Writing Task*. 2025. arXiv: 2506.08872 [cs.AI]. URL: https://arxiv.org/abs/2506.08872.

[7] Robert Love. *Presentation: Optimal GNOME Programming*. May 2005. URL: https://2005.guadec.org/schedule/gnometalks.html#speedylove.

[8] J.E. Lovelock. "Gaia as seen through the atmosphere". In: *Atmospheric Environment (1967)* 6.8 (1972), pp. 579–580. ISSN: 0004-6981. DOI: https://doi.org/10.1016/0004-6981(72)90076-5. URL: https://www.sciencedirect.com/science/article/pii/0004698172900765.

[9] Kaveh Madani. *Global Water Bankruptcy: Living Beyond Our Hydrological Means in the Post Crisis Era*. Jan. 2026. DOI: 10.53328/inr26kam001. URL: http://dx.doi.org/10.53328/INR26KAM001.

[10] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[11] Peter Jay Salzman. *The Linux Kernel Module Programming Guide (ver 2.6.4)*. URL: https://tldp.org/LDP/lkmpg/2.6/html/index.html.

[12] Larry Wall. *Request for Regular Expression Feature -why not{?foo}*. URL: news:1994Jun15.074039.2654@netlabs.com.

[13] Joss Whedon, Ashley Gable, and Thomas A. Swyden. *Buffy the Vampire Slayer – season 1, episode 8: "I, Robot… You, Jane"*. Apr. 1997.

Index