Programming Essentials: A Beginner's Guide With A Holistic Approach

Paperback ISBN: 9798319237163

Hardcover ISBN: 9798280102392

Cover Design by Anze Ban Virant - ABV Atelier Design

First Edition: May 2025

This book is intended to provide general information and guidance regarding programming concepts and techniques. The author and publisher make no representations or warranties with respect to the accuracy or completeness of the contents and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. The reader is encouraged to consult additional resources or professional advice as needed.

# Table of Contents

## Preface

I wanted to write this book because formal coding courses can be challenging for learners. The main difficulty is that learners are often taught the implementation or syntax rules first, without a proper understanding of the underlying concepts. Therefore, this book introduces the essential concepts that every developer should learn upfront. In this regard, it stands out from other books.

The book begins by introducing all the Linux applications and commands you will use when writing code. Before diving into their usage, it also explains how these applications are executed within the Linux operating system (OS). This approach will help you become familiar with the Linux system and the fundamental concepts that you will use in your applications.

Additionally, the book aims to show you that you don't need any complex tools or setups to start coding — an average computer is sufficient. Learning to code in this way will also help you understand how more complex tools used in professional environments work under the hood and what they do behind the scenes.

## Who should read this book?

The book is aimed at absolute beginners with no prior programming experience. It is not suitable for those who do not plan to write and run all the sample code as described.

Some concepts included are considered advanced topics even for experienced learners. They are incorporated based on the organization of the material, presented at appropriate points in the learning process. For example, instructions on cross-compilation are provided, but this is a technique that is usually not needed for everyday development. Cross-compilation is included as a reference for those who may need it later and to demonstrate the underlying concept.

Additionally, certain concepts are introduced to familiarize readers with the Linux-based system. For instance, the **kill** command is explained to illustrate that almost any action can be performed using Linux commands.

## Acknowledgments

I would like to extend my heartfelt gratitude to Shasank Dave, Eray Ates, and Denis Anisimov for your support, encouragement, and contributions.

## Teaching Style

The book presents each concept as a case study to illustrate why it is needed and what problem it solves. After a concept is introduced, related statements will be highlighted in red, such as:

**Learn how to access to arguments**

**The red statements represent items on your to-do list when learning a new programming language**. After these red statements, the concept will be implemented through sample code in Go. This teaching style is called "**abstract teaching**" a term derived from "**abstraction**" in programming, which will be explained later in the book.

All red statements in the book are compiled into the "**Essential Language Learning Checklist**" section of the Appendix for future reference.

## More about chapters

A typical programming course or training often begins with a chapter about variables. However, learners should also understand more about programming languages, how to execute applications, and how operating systems run these applications. Learning the concepts first provides a stronger foundation for coding and helps you communicate with others using the correct terminology.

Therefore, this book starts with the key concepts you should become familiar with first. That is why the title emphasizes a **holistic approach**.

The first chapter will explore a concept from Japanese martial arts related to the stages of learning. In the IT industry, learning is a continuous process—you will always encounter new tools, languages, or technologies. Understanding these stages is essential, as it helps you determine which stage you want to reach or consider sufficient for each technology you use.

**The second chapter focuses on how to run applications. It explains what an application is and how it operates within the OS. But why is this important? Throughout your software development career, you will use various applications such as editors, compilers, and interpreters. These tools help you develop, build, and deliver your code. To effectively use these tools, it's essential to understand how they are executed. This understanding will also assist you in running the applications you develop in the subsequent chapters.**

**Additionally, when an application runs in an OS, it follows a set of common steps regardless of the application or programming languages. Several concepts relate to these steps, and understanding them will help you access and use the concepts introduced in this chapter. As you learn about application execution, you will gradually become familiar with Linux commands, which will be used throughout the book.**

**The chapter also aims to help overcome any fear of working with a command-line interface or terminal. Furthermore, it demonstrates how developers transform code into executable applications. Since this process varies across programming languages, your code will go through different procedures depending on the language you use.**

**The third chapter will focus on creating a "Hello World" application from scratch. You will learn how to create files and directories, as well as how to write your code into these files. Finally, building on chapter two, you will learn how to run your code, similar to how applications are executed earlier. Use this chapter as a reference point while running code samples in the subsequent chapters.**

All subsequent chapters will focus on various technical programming concepts.

Finally, the book concludes with a roadmap to guide your continued learning. One of the most critical skills you need to develop is **debugging**. Improving debugging skills is an essential aspect of programming, yet it is often underrepresented in programming education. While this book does not cover debugging in detail, your development tools will provide the necessary information to learn more about this skill. For additional guidance, refer to the debugging section in the last chapter.

## What do you need throughout the book?

You can access everything you need with the help of Docker (https://www.docker.com). Docker is a helpful containerization tool. At this point, you do not need to know anything about Docker or containerization. Docker will let all readers perform the same tasks, regardless of their OS or computer.

There are two ways to use Docker. Firstly, you can install it on your machine. However, you may need to change some BIOS settings to enable virtualization. You can find information about how to install it on the internet.

Alternatively, use the online Docker playground (https://labs.play-with-docker.com/). It lets you create containers for a limited duration of three hours. Do the following to access the playground:

1. Go to https://labs.play-with-docker.com/).
2. Click login.
3. Select docker and log in after creating an account.
4. Click **Start**, and your session starts.
5. On the left menu, click **Add New Instance**.
6. Now, you can use the terminal on the right panel.

### What is a container?

You do not need to know the detailed workings of containerization, but understanding what a container does and why to use it can be helpful. Think of a container as a brand-new, isolated computer that runs inside your existing system, with preinstalled software and files. The container provides access to all the necessary software and files for the book from your computer. The containers you create will include the following contents:

- Linux OS
- Go toolchain
- All files and software you need to practice

## Creating the container

In the examples in this book, you will need to create a container from an image using Docker. Afterward, you can follow the instructions for each example.

You can create a container with the following command. Open the command prompt or terminal with superuser privileges. Then, run: **docker run -it --rm denizgursoy/dev-lab**

```
user@local:~$ docker run -it --rm denizgursoy/dev-lab
Unable to find image 'denizgursoy/dev-lab:latest' locally
latest: Pulling from denizgursoy/dev-lab
d25f557d7f31: Already exists
8098e2c93c77: Pull complete
59f7b7f4c914: Pull complete
4f4fb700ef54: Pull complete
Digest:
sha256:442b359be2f88cab9ee1167b15b2aea74d3e59d65d8852d8650370bea58
0c9ed
Status: Downloaded newer image for denizgursoy/dev-lab:latest
/projects/src #
```

If you haven’t downloaded the image (denizgursoy/dev-lab), Docker will download it automatically. The message “Status: Downloaded newer image for denizgursoy/dev-lab:latest” means the system downloaded the image. After that, Docker will start a container. Now you have a terminal from the container created. If you want to exit the container, you need to write:

```
exit
```

And then press Enter. You will be back on your command prompt/terminal.

Note:

When you exit the container, Docker will destroy the container because of the --rm argument. The files/directories you created will be destroyed with the container. To continue from the same container without losing files, use this command to create a named container:

```
docker run -it --name dev-lab denizgursoy/dev-lab
```

After you exit the container, you can resume it with this command:

```
docker start -ai dev-lab
```

That was all the setup you needed to do; if it is successful, you are now ready.

# Chapter 1

## Stages of learning

You will learn:

- Shu Ha Ri concept

Shu Ha Ri ( 守破離 ) is a Japanese martial arts concept that describes stages of learning.

Marting Fowler (2014) defines Shu Ha Ri as follows:

> Shu – In this beginning stage the student follows the teachings of one master precisely. He concentrates on how to do the task, without worrying too much about the underlying theory. If there are multiple variations on how to do the task, he concentrates on just the one way his master teaches him.
>
> Ha – At this point the student begins to branch out. With the basic practices working he now starts to learn the underlying principles and theory behind the technique. He also starts learning from other masters and integrates that learning into his practice.
>
> Ri – Now the student isn't learning from other people, but from his own practice. He creates his own approaches and adapts what he's learned to his own particular circumstances.

## Baking Pizza

Let's apply the concept to pizza baking.

### 1. Shu (守) - Follow the Rules

In this stage, the learner follows the recipe strictly to understand the basic pizza-making process.

Activities:

Choose a Simple Recipe: Select a basic pizza dough recipe and a simple tomato sauce.

Follow Instructions: Learn how to accurately measure ingredients, knead the dough for the right amount of time, and cook at the appropriate temperature.

Focus on Techniques: Demonstrate how to stretch the dough, spread the sauce evenly, and sprinkle toppings.

Key Takeaway: In this stage, it's important to follow established methods exactly to develop foundational skills and knowledge.

### 2. Ha (破) - Break the Rules

In this stage, the learner begins to experiment with variations and adapt the learned techniques to understand variations and personal touches in pizza-making.

Activities:

Experiment with Ingredients: Encourage learners to try different types of flour (like whole wheat or gluten-free) or sauces (like pesto or BBQ sauce).

Creative Toppings: Let them explore unconventional toppings or combinations, such as different cheeses, vegetables, or proteins.

Techniques: Learn how to make their own pizza base (like a stuffed crust or thin crust) and try new cooking methods (such as grilling or using a pizza stone).

Key Takeaway: This stage is about understanding the fundamentals deeply enough to break away from them and make personal adjustments.

### 3. Ri (離) - Transcend the Rules

In this stage, the learner can innovate and create their own pizza style, integrating everything they've learned to develop personal style and insights based on experience.

Activities:

Create a Signature Pizza: Learners can design their own unique pizza, combining techniques and flavors they've developed.

Presentation and Flavor Pairing: Learn how to pair flavors and presentation techniques (like how to garnish or serve the pizza).

Community Feedback: Have learners present their pizzas to a group for feedback, encouraging them to explain their thought process and choices behind their pizza creation.

Key Takeaway: This stage is about mastery—being able to create new forms and ideas of pizza-making that reflects personal style and insights.

## Shu Ha Ri in Learning Software Development

**Shu Ha Ri** is also an important concept for developers. Throughout your career, you will work with many different technologies. Mastering all of them and reaching the Ri stage in each is nearly impossible.

Instead, focus more on understanding the underlying principles of the technologies you prioritize, aiming to reach the Ha stage or even Ri in those areas. For other technologies, the Shu stage may suffice initially.

In this book, you will learn many concepts and will be at the Shu stage of learning these. Achieving higher levels requires practicing the concepts and further reading their documentation.

Starting from the next chapter, as you learn new concepts, take your time to find more information or watch videos online. Experiment with your ideas in containers, since it's safe—you cannot break anything easily. If you do encounter issues, just create a new container and start fresh.

Another effective way to retain knowledge is through repetition. Try typing commands or coding manually, even if you believe you have mastered them. Avoid copying and pasting; instead, practice to become comfortable with the terminal.

Finally, keep in mind that many concepts presented here are independent of any specific programming language. Every developer should first learn the basics of Linux, as you will likely use or deploy to Linux-based systems, regardless of the programming language you choose.

# Chapter 2

## Executing Applications

You will learn:

- The compilation, compile-time, and cross compilation
- Interpreters
- Executing applications
- LS command
- Arguments
- Logs
- Runtime
- Pwd command
- Which command
- Ps command
- Kill command
- Environment variables
- Stdin, Stdout, Stderr

## What is an Application?

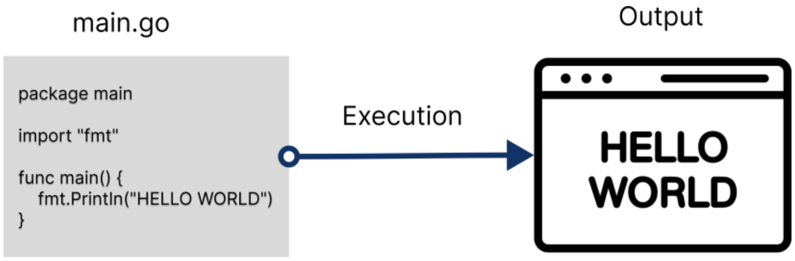
Applications are software that operate computers. This chapter is dedicated to exploring the applications you will use most often that come with the Linux OS. Understanding how an application is executed and what happens behind the scenes in the operating system is crucial, as you will be using many of these applications during and after your development process. For example, you will use a text editor application to write your code, along with various Linux commands to create files and directories. You will also use the Go application to build and run your programs from the source code.

Finally, the terms introduced in this chapter are independent of any specific programming language. The more you understand these terms and concepts, the better your overall comprehension of programming languages will become.

## From text to output

Firstly, programming essentially comes down to writing text. However, there are many rules you must follow when writing code. These rules are called **syntax**, and they vary from one language to another. You will spend much of your time learning the syntax of the language you choose. This book will cover some of the syntax rules of the Go language in the upcoming chapters.

Before learning the syntax rules, it is important to understand how applications are created and executed. Let's start with the simplest example: the Hello World application.



```
package main

import "fmt"

func main() {
   fmt.Println("HELLO WORLD")
}
```

```
HELLO
WORLD
```

*Figure 2-1. Source code to output*

How is the source code written in a file executed, and how does it print the output? The answer may change from language to language. Firstly, some languages are **compiled languages**. In a compiled language, source code is transformed into an **executable(application)** generated by another application called a **compiler**. This generated executable can then be run like any other application within the operating system.

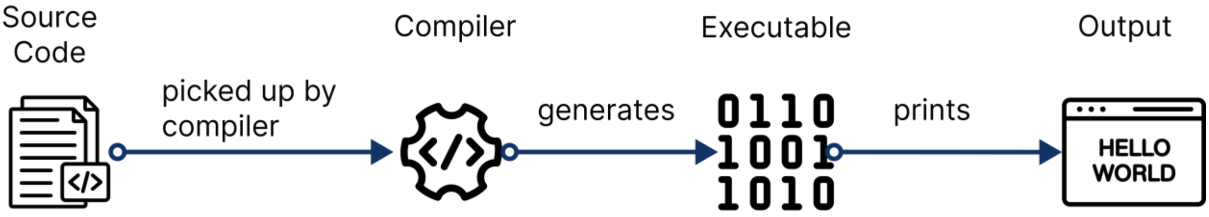


*Figure 2-2. From source code to output in a compiled language*

Converting source code into an executable is called **compilation**. A key point about executables created this way is that they can only run on computers with the same operating system and processor architecture as the one used for compilation. For example, if you compile your source code on a Linux computer with an amd64 processor architecture, the resulting executable will not work on a Windows computer with the same architecture.

In short, applications often need to be compiled separately for different OS and architecture combinations. Generally, compilers support **cross-compilation**, which allows you to compile code for different OS and architecture pairs on the same machine.

In a compiled language, the compiler examines the source code to ensure it adheres to all syntax rules and generates an executable if no syntax errors are found. The duration spent during this process is called **compile time**. Compile time is important because some features and capabilities of the software are determined at this stage and cannot be modified afterward. Any error during compile time will result in a **compilation error**.

Not all programming languages are compiled. In some languages, the source code is read and executed directly without generating artifacts like executables. These are called **interpreted languages.** In such cases, the code is provided to an interpreter installed on the operating system, which then executes the code directly.

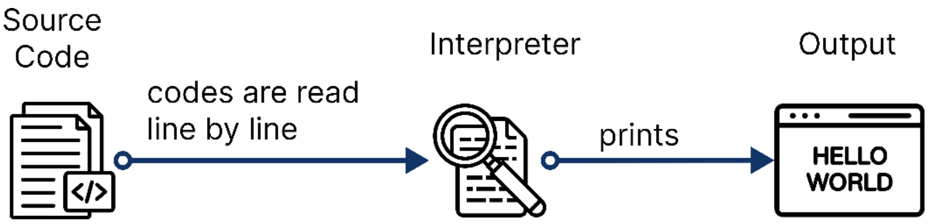


*Figure 2-3. From source code to output in an interpreted language*

In interpreted languages, the source code is ready to be used as-is. How is the code executed on different operating systems and processor architectures? A suitable interpreter must be installed on the platform to execute code written in interpreted languages.

Finally, some languages utilize both compiled and interpreted approaches and are known as **hybrid languages**.

**Learn whether your language is compiled, interpreted, or a hybrid language**

Go is a compiled language that generates different executables depending on operating systems and processor architecture.

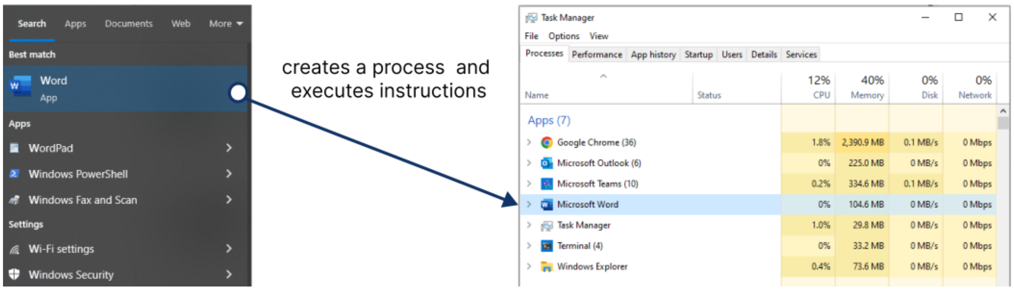
**Learn how to cross compile**

The **GOOS** and **GOARCH** environment variables are used to build for different OS and architectures. Cross compilation will be presented in the next chapter.

## Installing and executing applications

We always installed applications via installer before we used them. What does an installer do, then? In the case of compiled languages, the installer moves the compiled executable suitable for your computer into a directory in your OS, along with the files the executable needs during execution.

In Windows OS, applications are run by double-clicking on an executable or a shortcut to it. When an executable is launched, the operating system loads the program into memory, creates a **process** (which can be seen in the Task Manager), and begins executing its instructions.



creates a process and executes instructions

| Name | Status | 12%<br>CPU | 40%<br>Memory | 0%<br>Disk | 0%<br>Network |
|---|---|---|---|---|---|
| Apps (7) | | | | | |
| Google Chrome (36) | | 1.8% | 2,390.9 MB | 0.1 MB/s | 0 Mbps |
| Microsoft Outlook (6) | | 0% | 225.0 MB | 0 MB/s | 0 Mbps |
| Microsoft Teams (10) | | 0.2% | 334.6 MB | 0.1 MB/s | 0 Mbps |
| Microsoft Word | | 0% | 104.6 MB | 0 MB/s | 0 Mbps |
| Task Manager | | 1.0% | 29.8 MB | 0 MB/s | 0 Mbps |
| Terminal (4) | | 0% | 33.2 MB | 0 MB/s | 0 Mbps |
| Windows Explorer | | 0.4% | 73.6 MB | 0 MB/s | 0 Mbps |

*Figure 2-4. Applications and processes in Windows OS*

## Runtime

The time span between an application's start and termination is called **runtime**. Runtime is a crucial concept because it is during this stage that you can read and write files, receive user inputs, utilize networks, allocate memory, and perform many other operations.

Finally, when an application terminates, its process also ends, returning an **exit code** between 0 and 255. An exit code of 0 indicates that the process completed successfully, while any other code suggests that the application terminated with an error.

**Learn how to set exit code**

In Go, **os.Exit** function is used to set the exit code.

```
os.Exit(1)
```

## Commands

Many applications are installed within the container. Most of these applications come with the Linux OS and are primarily written in C, a compiled language. They are compiled using a compiler and then distributed to the OS inside the container, ready to be used. These small applications are also called **commands**. You can perform nearly every task by using Linux commands.

Let's discover **ls** command together.

### Ls command

The **ls** command is used to list the content of directories. To execute a command, you must write the application's path where it is installed and press enter. The **ls** command is located under the **/bin** directory, so we need to write **/bin/ls** and press enter:

```
user@local:~$ docker run -it --rm denizgursoy/dev-lab
/projects/src # /bin/ls
argument        management-app
/projects/src #
```

## Arguments

When you execute the **ls** command, it lists the contents of the current directory (**the working directory**). However, there is a hidden file named **.hidden.txt** (its name starts with a dot), which the **ls** command does not display by default. To show hidden files, you need to configure the **ls** command to include them.

To modify the behavior of a command, you add parameters (also called **arguments**) after the command name, separated by spaces. For example, to display hidden files with **ls**, you can use the **-a** argument. Now, execute the **ls** command with **-a** argument.

```
/projects/src # /bin/ls
argument        management-app
/projects/src # /bin/ls -a
.               ..              .hidden.txt     argument
management-app
/projects/src #
```

If you pay attention, the **ls** command also showed two other items: `.` and `..`. In Linux, `.` and `..` are special directory entries that make navigating the filesystem more efficient. Here's a simple explanation:

**`.` (Single Dot):** Represents the current directory. It's a shortcut to refer to the directory you are currently in (**working directory**).

**`..` (Double Dot):** Represents the parent directory. It's a shortcut to move up one level in the directory tree.

**Note:**

Remember that **arguments** are passed during the startup of an application. Arguments are convenient for short-term applications, such as commands, that run for a brief period. However, some applications run continuously to provide services, such as databases and web servers.

If you develop long-running applications, you can still use arguments to configure the application at startup. However, if you need to pass data to the application during its runtime without restarting it, you should avoid using arguments. Instead, consider other methods such as receiving user input through the console, using graphical user interfaces, or making web requests. These approaches allow the application to operate continuously without interruption.

**Note:**

It is important to understand the folder structure in Linux systems along with the use of the **ls** command. The entire Linux system resides within the / directory, known as the **root directory**. This is similar to the C:\ directory in Windows-based systems. The root directory contains many subdirectories, each serving specific functions within the system.

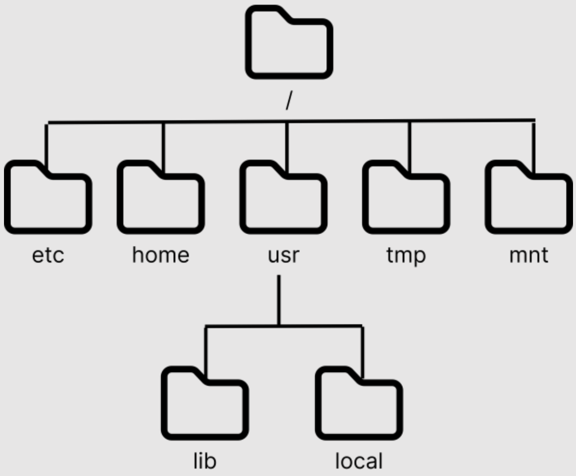


*Figure 2-5. Linux folder structure*

For example, the **/home** directory contains all users' personal folders, while the **/tmp** directory stores temporary files. If you need to locate specific files or directories, you can specify their full paths starting from the root (/) directory. For instance, the **lib** directory located under **usr** can be addressed as **/usr/lib**.

The complete path from the root directory to a specific file or folder is called the **absolute path** or **full path**.

### Learn how to access arguments

In Go, arguments are stored in the **Args** variable inside the **os** package and can be accessed as **os.Args**.

## Environment variables and the PATH variable

So far, we have written the absolute path of the **ls** binary whenever we want to execute it. Do we have to memorize the paths of all the commands in the OS? Of course not. You can also execute by simply typing the command.

```
/projects/src # ls
argument        management-app
/projects/src #
```

So, how does Linux locate and execute a program? Linux maintains a list of directories where it searches for executable files. This list is stored in a system-wide variable called **PATH**. The directories in **PATH** are combined into a single text, separated by colons (**:**).

When you type a command, the system searches through each directory listed in the **PATH** variable and executes the first executable it finds that matches the command name. If no matching executable is found in any of these directories, the system displays a **command not found** error.

```
/projects/src # tool
bash: tool: command not found
/projects/src #
```

Values stored in the computer, like the **PATH** variable, are called **environment variables**. Environment variables are dynamic key-value pairs. Applications can access the environment variables during the runtime. Therefore, they allow applications to behave differently without any code change. See following as an example:

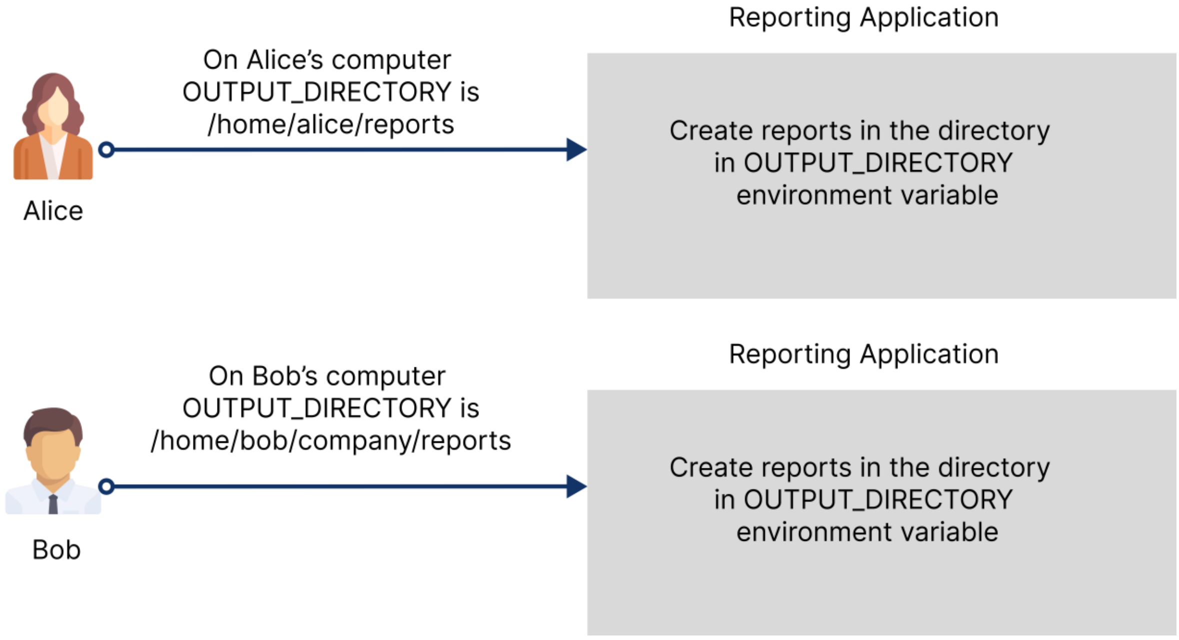


*Figure 2-6. The same environment variable with different values in different computers*

Suppose you develop an application that generates reports in specified directories. Users of the application want to choose where the reports are stored. To achieve this, the application reads the value of the **OUTPUT_DIRECTORY** environment variable and saves the reports to that location.

For example, on Alice’s computer, reports will be saved in **/home/alice/reports**, while on Bob’s computer, they will be stored in **/home/bob/company/reports**. This happens without any change to the application's code—only by setting the **OUTPUT_DIRECTORY** environment variable differently on each system.

The **env** command displays all the environment variables defined in the system.

```
/projects/src # env
HOSTNAME=ee2afe6fd156
PWD=/projects/src
HOME=/root
TERM=xterm
SHLVL=1
PATH=/go/bin:/usr/local/go/bin:/usr/local/sbin:/usr/local/bin:/usr
/sbin:/usr/bin:/sbin:/bin
GOPATH=/go
_=/usr/bin/env
/projects/src #
```

If you inspect the **PATH** environment variable, you'll notice that the last directory is the parent directory of the **ls** binary. This is why you can execute the **ls** command without typing its full absolute path.

Besides, if you want to learn where a command is located in the OS, you can execute the **which** command.

```
/projects/src # which ls
/bin/ls
/projects/src #
```

## Running an application you developed

So far, we have executed commands that come with the Linux OS**.** There is an **argument-app** application inside the container under the argument directory. Let’s see the content of the argument directory by executing the **ls argument/** command to list the content of the argument directory under the working directory.

```
/projects/src # ls argument/
argument-app  main.go
/projects/src #
```

To execute the **argument-app**, we need to find its absolute path. The **argument-app** is under the argument directory in the working directory, but how can we find which directory we are in? You can locate the working directory using the **pwd** command, which stands for "print working directory".

```
/projects/src # pwd
/projects/src
/projects/src #
```

**Note:**

The working directory is also displayed before the **#** sign by the used terminal.

```
/projects/src #
```

The working directory is **/projects/src**. We can combine directories with the **"/"**character. So, the absolute path of the **argument-app** resolves to **/projects/src/argument/argument-app**. Previously, it was said that **"."** also refers to the working directory, so we can also use the **./argument/argument-app**. If you reference subdirectories in the working directory, you can omit the leading **"./"** so **argument/argument-app** resolves to the same path. Let's run the **argument-app** with different arguments.

```
/projects/src # /projects/src/argument/argument-app arg1 -arg2 --
arg3
Argument at position 1 is /projects/src/argument/argument-app
Argument at position 2 is arg1
Argument at position 3 is -arg2
Argument at position 4 is --arg3
/projects/src # ./argument/argument-app arg1 -arg2 --arg3
Argument at position 1 is ./argument/argument-app
Argument at position 2 is arg1
Argument at position 3 is -arg2
Argument at position 4 is --arg3
/projects/src # argument/argument-app
Argument at position 1 is argument/argument-app
/projects/src #
```

**Note:**

Notice that first argument is always the location of the executable.

## Logs

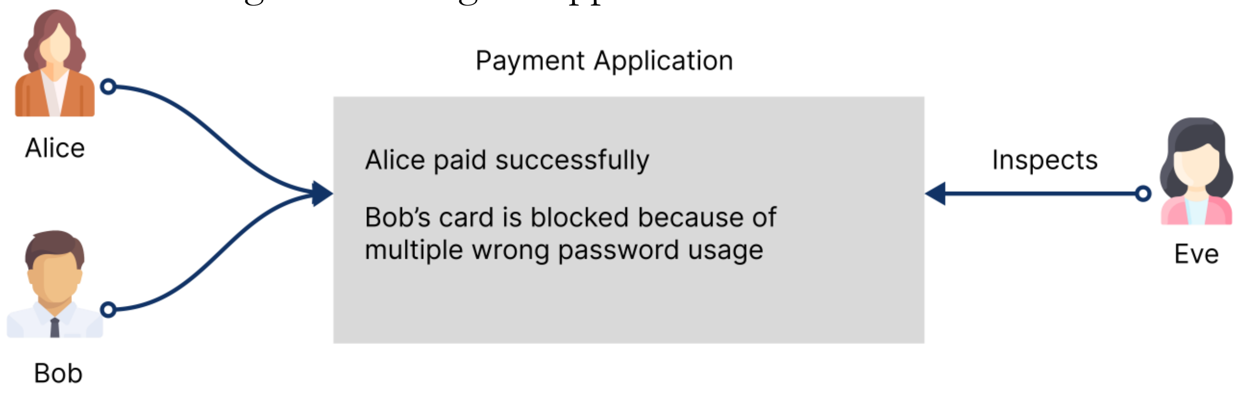
An application generally prints some lines to the console. These lines are called **logs**. Let’s execute the **ls** command with another argument **-l,** allowing us to see the content in an extended format.

Logs

```
/projects/src # /bin/ls -a -l
total 20
drwxr-xr-x    4 root     root          4096 Jun  2 17:51 .
drwxr-xr-x    3 root     root          4096 Jun  2 17:52 ..
-rw-r--r--    1 root     root            26 Jun  2 17:51
.hidden.txt
drwxr-xr-x    2 root     root          4096 Jun  2 17:51 argument
drwxr-xr-x    3 root     root          4096 Jun  2 17:51
management-app
/projects/src #
```

Logs are useful for monitoring applications and their current activities. Depending on the application's purpose, log entries should be detailed and descriptive.

Suppose you developed a payment application for an e-commerce company. Alice and Bob use your application to purchase items on your website. One day, Bob contacts your company's call center, saying he couldn't complete a purchase. Eve, another developer in your organization, needs to investigate why the payment process failed for Bob. Since your application handles data, maintaining logs with helpful information about its operations would assist other developers in troubleshooting and tracking the application's behavior.



Alice paid successfully

Bob’s card is blocked because of multiple wrong password usage

*Figure 2-7. Use of application logs*

In this scenario, Eve can see that Alice's payment was successful, while Bob's payment failed. By analyzing the logs, Eve can easily deduce that Bob's payment failed because he entered the password incorrectly multiple times.

How does the **ls** command log the content of the directory into the terminal in which we run the application? It achieves that via the three steams attached to every application before execution. These three streams are **stdin**, **stdout**, **stderr**.

**stdin (Standard Input)**
**stdin** is the standard input stream. It is used to read input provided by the user or another program. By default, **stdin** is associated with the keyboard.

**stdout (Standard Output)**
**stdout** is the standard output stream. It is used for writing the output of a program. By default, **stdout** is associated with the terminal.

**stderr (Standard Error)**
**stderr** is the standard error stream. It is used for writing error messages and diagnostics. By default, **stderr** is also associated with the terminal.

### Learn how to access to the three streams

In Go, **stdin**, **stdout**, and **stderr** streams are under the **os** package. They can be accessed using **os.Stdin**, **os.Stdout** and **os.Stderr** respectively.

### Learn how to read input from stdin

To read a line from **stdin**, **Scanln** function in **fmt** package can be used as:

```
var input string
fmt.Scanln(&input)
```

### Learn how to write to stdout and stderr

To output messages to the streams **fmt.Fprintln** can be used as:

```
fmt.Fprintln(os.Stdout, "message")
fmt.Fprintln(os.Stderr, "message")
```

In general, print functions in the **fmt** package print to **os.Stdout**.

```
fmt.Println(“message")
```

## More about commands

Sometimes, you may want to learn more about the commands available in the Linux operating system. This includes exploring different arguments that can be used with these commands and understanding how to utilize them. For example, you can run the **ls** command with the **--help** argument to learn more about its options and usage.

```
/projects/src # ls --help
BusyBox v1.36.1 (2024-06-10 07:11:47 UTC) multi-call binary.

Usage: ls [-1AaCxdLHRFplinshrSXvctu] [-w WIDTH] [FILE]...

List directory contents

        -1      One column output
        -a      Include names starting with .
        -A      Like -a, but exclude . and ..
        -x      List by lines
        -d      List directory names, not contents
        -L      Follow symlinks
        -H      Follow symlinks on command line
        -R      Recurse
        -p      Append / to directory names
        -F      Append indicator (one of */=@|) to names
        -l      Long format
        --full-time     List full date/time
        -h      Human readable sizes (1K 243M 2G)
        --group-directories-first
        -S      Sort by size
        -X      Sort by extension
        -v      Sort by version
        -t      Sort by mtime
        -tc     Sort by ctime
        -tu     Sort by atime
        -r      Reverse sort order
        -w N    Format N columns wide
        --color[={always,never,auto}]
/projects/src #
```

**Note:**

Not all commands have a **--help** option. You can also find detailed information and manual pages for these applications on the internet.

## More about processes

Every application creates a process in the operating system when it is executed. In a Windows-based system, you can view the list of running processes using the Task Manager. How can you see the list of running processes in a Linux operating system?

To view running processes, you need to execute the **ps** command.

```
/projects/src # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   29 root      0:00 ps
/projects/src #
```

Every process has a unique identifier called a PID, which can be seen in the PID column. PIDs can be used to reference specific processes. So, how can we terminate a running application? The **kill** command is used to terminate a process. To do so, you should pass the process's PID as an argument to the **kill** command.

```
/projects/src # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   32 root      0:00 ps
/projects/src # kill 1
/projects/src #
```

## In A Nutshell

In a nutshell, the following steps are done to execute the code based on the different types of languages.

In a compiled language:

1. Code is compiled by the compiler
2. The compiler creates an executable
3. The executable is run to start the application

In an interpreted language:

1. Code is given to the interpreter
2. The interpreter starts the application

In a hybrid language:

1. Code is given to the compiler
2. The compiler creates an artifact to be used later by an interpreter
3. Artifact is given to the interpreter
4. The interpreter starts the application

Here is a high-level overview of the application's execution:

1. You execute an application with the arguments.
2. The application is loaded to memory.
3. A process is created in the operating system.
4. The stdin, stdout, and stderr are attached to the application.
5. The code of the applications is executed.
6. The application is terminated with an exit code.
7. The process is terminated, and allocated memories are released.

## Summary

- Generating executables from the source code is **compiling**.
- When an application is compiled, it works only in the same OS and processor architecture type. Compiling applications for different OS and processor types is called **cross-compilation**.
- Codes can be executed directly in some languages without compiling via interpreters installed on OSs. These types of languages are called **interpreted languages**. In a language, if you need to compile the application to execute it, it is called a **compiled language**. Some languages are both compiled and interpreted languages. They are called **hybrid languages**.
- The time spent in compilation is called **compile time**. Any error during compile time will result in a **compilation error**.
- Some applications come within the OS, and they are called **commands**.
- The **ls** command is used to list files and directories in a particular directory.
- Paths of files or directories from the root directory are called **absolute** or **full paths.**
- If you need to execute an application, you need to write the **absolute path** of the executable.
- The period between the application is run and terminated is called **runtime**.
- When an application is executed, it creates a **process** in the OS. All running processes can be listed with the **ps** command. A process can be terminated with the **kill** command. A process ends with an **exit code** to inform the user whether it ran successfully or not. Exit code can be returned with **os.Exit** function in Go.
- Arguments allow you to configure applications. The -a argument in the **ls** command includes hidden files in the listing. The -l argument shows items in a long format.
- The current directory you work in is called the **working directory**.

- In the Linux system, if you want to address files or directories in your working directory, you need to use the ./ prefix. If you need to refer to the parent directory, you should use the ../ prefix.
- Linux system is stored in the / directory, known as the **root directory**.
- More information about a command can be displayed by using the help argument.
- When an application is executed, it prints information to the terminal. These texts are called **logs**. Logs are used to track the applications.
- **Environment variables** are key and value pairs stored in the OS. Users can personalize environment variables. An application can read or change env variables.
- The **env** command lists all the environment variables defined in the system.
- **PATH** is a special environment variable that includes all the directories where executables are located. Directories are separated by a colon.
- The **which** command locates the executable by searching it inside the directories in the **PATH** variable.
- If you want to execute an application that is not in the **PATH** variable, you should write its absolute path. If you want to run an executable in the working directory, you should use the prefix ./ before the executable name.

## Learn more

- Learn more about the Linux folder structure and the directories under the root directory.
- Learn whether languages like Java and JavaScript are compiled, interpreted, or hybrid programming languages.
- Navigate among directories.
- To learn other commands and to see you can perform a lot of operations with the commands, try to do the followings:
  - Learn how to check disk space

- Learn who is the current user
- Learn how to see all of the users defined in the system
- Learn how to find a file
- Learn how to get CPU information

# Chapter 3

## Create Hello World Application

You will learn:

- Mkdir command
- Cd command
- Touch command
- Nano command
- Go build and go run
- Cat command

## Hello World Application

"Hello World" is a simple application that only prints some text to standard output (stdout). Creating and running a "Hello World" program is important because it demonstrates that all the necessary development tools are installed and ready to use. For the sample codes in the upcoming chapters, you will create new applications in a container, and you should follow the steps below whenever you need to create a new application.

1. Create a container by executing **docker run -it --rm denizgursoy/dev-lab**
2. Create a new directory for every application to store its source files.

To create a new directory, you can use the **mkdir** command. You should pass the path you want to create the folder as an argument, and it will create the folder there. In our case, you can execute **mkdir first-application** command, and it will create the first-application directory in the working directory.

```
/projects/src # mkdir first-application
/projects/src #
```

Now, you need to change your working directory to the new one you created. We will create source files in the **first-application** directory and execute some commands easily inside the same folder.

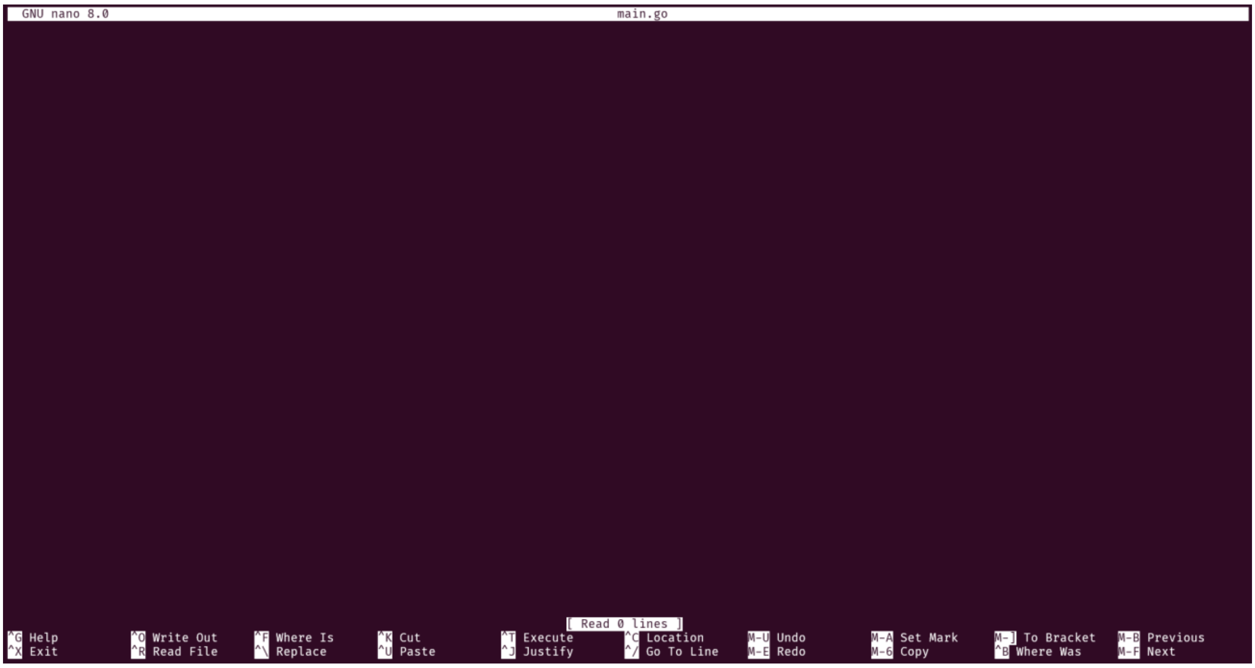
To change the directory, you need to use the **cd** command, which stands for **change directory**. You should provide the path of the directory as an argument. For example, to switch to the **first-application** directory, you would execute:

```
/projects/src # cd first-application/
/projects/src/first-application #
```

Now, we need to create a file to write our code. Generally, people name it **main.go**. A file can be created easily with the **touch** command. You should give the path of the target file as an argument to the **touch** command. In our case, you should execute **touch main.go**. It will create main.go file in the **working directory**.

```
/projects/src/first-application # touch main.go
/projects/src/first-application # ls
main.go
/projects/src/first-application #
```

It is time to start writing our first code. We will use the installed nano application in the container, allowing you to edit text files on the terminal. You should execute **nano main.go**.

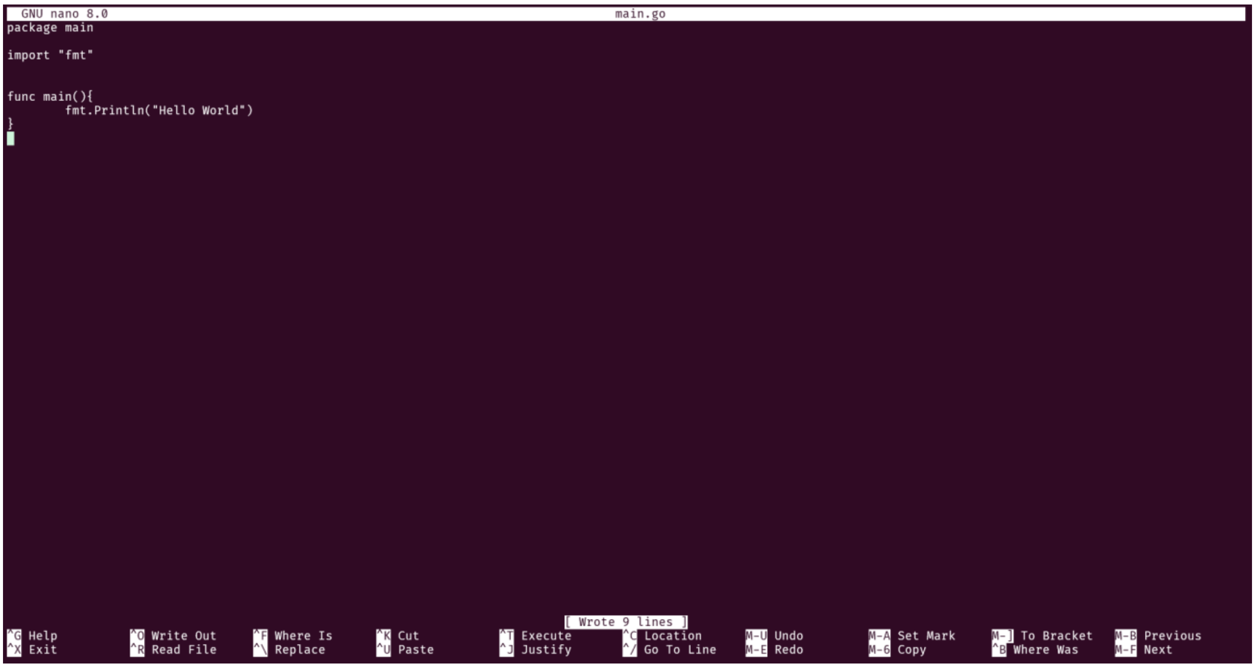


```
  GNU nano 8.0                                   main.go

                                             [ Read 0 lines ]
```

*Figure 3-1. Nano application interface*

Write the following code into the editor:

```
package main


import "fmt"


func main() {
      fmt.Println("Hello World")
}
```



```
package main

import "fmt"


func main(){
        fmt.Println("Hello World")
}
```

*Figure 3-2. Nano application with code written*

Press Ctrl + O to save the file, and press Enter. After saving, press Ctrl + X to exit. We are back in our terminal. Now we can compile our source code and get an executable. We will use the go command installed in the container. To compile the source code, execute **go build** command.

```
/projects/src/first-application # nano main.go
/projects/src/first-application # go build
```

> **Note:**
>
> The **build** argument of the **go** command calls the go compiler to build source code into an executable **with the working director's name**. Note that you need to execute the command on where your main file is.

If you execute the **ls** command, you will see an executable having the working directory's name. To execute it, you should run **./first-application**.

```
/projects/src/first-application # ls
first-application main.go
/projects/src/first-application # ./first-application
Hello World
/projects/src/first-application #
```

Regarding cross-compilation, the Go command builds executables using the **GOOS** and **GOARCH** environment variables. If these variables are not set, it uses current system information. You can set these variables before executing the **go build** command to compile for different systems.

All possible OS and processor architectures Go can build for can be seen with the **go tool dist list** command.

```
/projects/src/first-application # go tool dist list
…
plan9/arm
solaris/amd64
wasip1/wasm
windows/386
windows/amd64
windows/arm
windows/arm64
/projects/src/first-application #
```

Let's build for **windows/amd64**. To do that, we need to set the environment variables before the build command as: **GOOS=windows GOARCH=amd64 go build**

```
/projects/src/first-application # GOOS=windows GOARCH=amd64 go
build
/projects/src/first-application # ls
first-application first-application.exe  main.go
/projects/src/first-application #
```

Whenever you modify the code, creating the binary and executing it to verify its functionality can be time-consuming. Instead, you can run the code directly from the source by executing **go run .** command. When you run an application with the **go run** command, Go compiles the code, creates a temporary executable, and runs it automatically, all behind the scenes.

```
/projects/src/first-application # go run .
Hello World
/projects/src/first-application #
```

**Note:**

The go run command expects **the file path containing the main function or the directory of the main file** as an argument.

```
go run main.go
go run .
```

By executing the above command, we tell the compiler that the main is in the working directory. However, make sure that your package name is main and that the main function is also defined. Otherwise, the application will not compile.

You can still pass your arguments after the reference to the main as follows:

```
go run . arg1 arg2
go run main.go arg1 arg2
```

Let's see the code in the main.go and dissect it to understand deeper. To see the code, you can execute **nano main.go**, or to inspect it; you can use **cat** command to print the content of a file to the terminal. We can do it by executing **cat main.go**.

```
/projects/src/first-application # cat main.go
package main

import "fmt"

func main(){
        fmt.Println("Hello World")
}
/projects/src/first-application #
```

Go source codes are grouped under packages. Hence every go file must belong to a package, and that package should be stated in the first line as:

```
package main
```

Every directory is a package in Go, and you can have many Go files inside a directory/package. All the go files inside the same directory must have the same package name. Otherwise, you will get a compilation error.

**You must import a package first if you use a resource belonging to that package**. Since we use the **Println()** function from the **fmt** package that comes built-in with Go, we write:

```
import "fmt"
```

In Go, as in most compiled languages, the **main** is a specific function. The **main** function is the start point of the application, where it executes your code line by line. **The main function in Go must be in the main package.** That is why the package name has to be main. You cannot have multiple main functions in the main package because the compiler does not know which one to execute.

The compiler will raise a **compilation error** when you execute **go build** or **go run** command with a source code file without a main function or package. Hence, we had to define a function like:

```
func main() {
      fmt.Println("Hello World")
}
```

> **Note:**
>
> However, this does not mean that your source code file, including the **main function,** should have the name `**main.go**`. It can have a different name, but by convention, people use **main.go**.

You can use the **Println** function in the **fmt** package to print to **stdout**. In Go, whenever you want to use a resource from another package, you must always combine it with the package name. Therefore, we write the following **statement** in the main function:

```
fmt.Println("Hello World")
```

Applications consist of **statements** performing specific actions. They are executed from top to bottom, and every statement is written in a new line. Lastly, statements in most programming languages must end with the `;` semicolon. However, you do not have to use it in Go. See the following example:

```
func main() {
      fmt.Println("First statement")
      fmt.Println("Second statement")
}
```

**Note:**

In the following chapters, we will also use the **Printf** function. The function takes format as the first argument. The format may have many verbs starting with the **%** symbol. The function replaces the verbs with values in the following arguments, respectively.

```
fmt.Printf("%s book has %d pages\n”, "hamlet", 216)
```

It will print:

```
hamlet book has 216 pages
```

The print function shows the value in one line. To go to the next line, **\n (Escape Sequence)** must be used.

See the Formatting Verbs section in the appendix to learn about other verbs.

Now, you can modify the **main .go** file, change the package and function names, and try to run it to practice what you learned.

**Note**:

Your source code may also include comments describing the code's purpose. Comments help developers explain the logic. The compilers and interpreters ignore the comments. Single-line comments start with a double slash //. See:

```
func main() {
      // my first application
      fmt.Println("hello world") // change here
}
```

**In the following chapters, you should create a new project for every chapter and execute the code samples as described in the chapter.**

## Summary

- The **mkdir** command creates a directory.
- The **cd** command changes the working directory.
- The **touch** command creates files.
- The **cat** command displays the content of files.
- File content can be viewed or changed with the **nano** command.
- Go application starts from the **main function** in the **main package**.
- The **Println** function in the **fmt** package can be used to print to **stdout**.
- An executable can be created with the **go build** command where the main function is in the main.go. It will create the executable in the working directory.
- To cross compile, **GOOS** and **GOARCH** env variables should be set before the build command.
- A Go application can be run with **go run .** command which builds the executable and run it.
- Comments give information about the code. A comment starts with //.

## Learn more

- Learn how to remove files and directories.
- Try to install another text editor to the container via the terminal.
- Learn how to copy files to another directory.
- Learn how to move files to another directory.
- Learn how to rename files and directories.

# Chapter 4

## Memory, Bits, Bytes, Variables and Constants

You will learn:

- Bits and bytes
- Memory
- Variables
- Constants
- Enumeration

## Memory, Bits and Bytes

Applications need to store data and process it during runtime. They store their data in the computer's Random Access Memory (RAM), which is ephemeral and acts like short-term memory. If the computer is restarted, all stored values will be lost.

Memory consists of blocks that store the data you want.

*Figure 4-1. Memory blocks*

Each memory block has an address. You can access the value you stored in a block by its address.

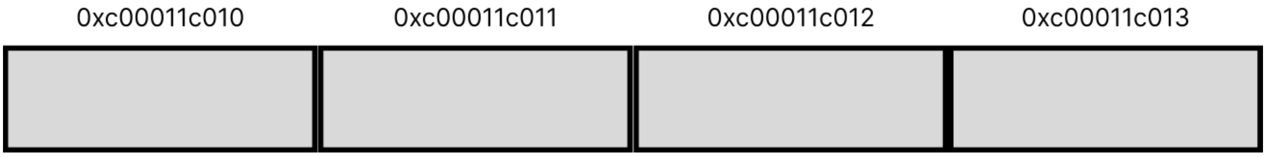


*Figure 4-2. Memory blocks with addresses*

There are certain rules for storing data in a memory block. **First, only integer values can be stored directly in memory**. If you want to store your name in memory, it is stored as a sequence of integers representing each character. We will discuss later how characters are stored as integers in memory.

Secondly, the integer value must be stored as **bits** in **binary** format. A bit is a computer's smallest data measurement unit, 0 or 1. With a single bit, we can represent, at max, two different values. If we need to represent more values, we must use more bits. Let's say we have 2 bits; the following combinations are all values we can represent with 2 bits:

00

01

10

11

There are 4 combinations created with 2 bits. Therefore, we conclude that the total number of values that can be represented with n bits is $2^n$.

Lastly, A combination of 8 bits has a special name called a **byte.** A memory block can store **exactly** 8 bits (1 byte). Therefore, a byte can have $2^8$=256 values ranging between 0 and 255. **If one block is insufficient for the data you want to store, you cannot steal another bit from the next block. You must allocate consecutive memory blocks to store the integer value you have**. The table below shows the maximum value you can store by the bytes count.

| Byte count (Block count) | Bits | Minimum value | Maximum Value |
|---:|---:|---:|---:|
| 1 | 8 | 0 | 255 |
| 2 | 16 | 0 | 65535 |
| 3 | 24 | 0 | 16777215 |
| 4 | 32 | 0 | 4294967295 |

*Table 4-1. Byte sizes and their range of values*

How is an integer converted to binary? To convert an integer (in decimal base) to binary, we must divide the integer repeatedly by 2 until the quotient is zero and write the reminders in the reverse order. Let's convert 26 to binary:

26 ÷ 2 = 13 remainder 0

13 ÷ 2 = 6 remainder 1

6 ÷ 2 = 3 remainder 0

3 ÷ 2 = 1 remainder 1

1 ÷ 2 = 0 remainder 1

So 26 is 11010 in binary format. When it is stored in the byte, it is padded with the leading zeros if it is not 8 bits. How can we convert a binary value back to integer? We need to multiple digits starting from the rightmost digit by the power of 2. The first exponent starts from zero and increases until the last digit. Let's convert 11010 back to integer.

$$\begin{array}{cccccc} 1 & 1 & 0 & 1 & 0 & \\ 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\ 16 + & 8 + & 0 + & 2 + & 0 & = 26 \end{array}$$

If we convert 503 to binary, it is 00000001 11110111. So, we need two bytes to store 503 in memory. Let’s calculate the integer value of each byte. Binary 00000001 is 1, and 11110111 is 247.

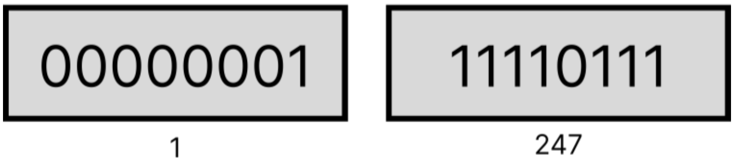


*Figure 4-3. One integer stored as two bytes in the memory*

In short, integer 503 requires two bytes, and its bytes values are 1 and 247 respectively.

So far, all the integer values we've discussed have been positive. Now, it's important to distinguish between different types of integers. Some values, such as the number of employees in a company, logically cannot be negative. In these cases, we use **unsigned integers**, which do not store the sign of the number.

However, in other contexts—like storing temperature values—integers can be both positive and negative (**signed integers**). So, how are negative values stored in memory? How is the sign encoded? Negative integers are typically stored using **Two’s complement** representation.

Two’s complements method involves the following steps:

1. Ignore the sign and convert the number to binary.
2. Invert the bits. Make 0’s 1 and 1’s 0.
3. Add 1

Let’s do it for -26 together.

1. 26 is 11010 in the binary, and we pad it until it is 8 bits. So the result is 00011010
2. We invert 0’s and 1’s so bits are 11100101
3. We add 1 to bits, so 11100101 + 1 = 11100110

Therefore, -26 is stored in the memory as 11100110.

**When negative numbers are written in binary, the leftmost bit is the sign bit: 0 for positive and 1 for negative**. So, the first bit always stores the sign. This, therefore, means that we have 7 bits left for the actual number. With 7 bits, the maximum positive value that can be stored is $2^7$ -1 = 127 (excluding zero), and the minimum negative value is -$2^7$ =-128. Therefore, in a byte, negative values range from -128 to -1, and positive values range from 0 to 127.

## Variables

So far we learned the theorical part how we store integer values in the memory. Let's say we write an application that process number of employees in a company, we have the following questions:

- How many blocks do we need to allocate?
- How do we allocate those blocks? How should we remember the address of the memory blocks so we can read the value stored later?
- How can we store a value in the block/s? Should we convert every value to binary when we want to store value in the blocks?

Let's say the employee count is 26.

How many blocks we must allocate depends on the expected maximum values you want to store as employee count. Block counts are associated with predefined **types** in programming languages, which will be covered in the next chapter. For now, we will use the **uint8** type, which stands for **unsigned 8 bits integer**. It allocates one block/byte that only stores positive values up to 255.

We decided to use the **uint8** type. How can we allocate the memory for the type and reference it in our code? We can allocate the memory by creating **variables**. When we allocate the memory, it has an address like **0xc00011c010**, so do we have to memorize the address for every variable? The address will also change in every allocation because the computer will return the address of an **available memory block**. Besides, the address is not human-readable and impossible to remember. Therefore, we need to give an alias to our reserved memory block. It is called a **variable name**.

As to variable names, you must abide by the following rules:

- Variable name cannot start with a number
- Variable names cannot have symbols like $!
- Variable name cannot have space
- Variable name can have underscore but not hyphen
- Variable name cannot be any keywords in the language

When you declare variables, there are some naming cases that you should follow. Languages do not force you to use them; you should use them to write readable code. See the Naming Convention section in the appendixes for full list of the naming cases. In Go, by convention, variable names are written in **camel case**.

Let's select a variable name, then. It should be descriptive enough about the value it stores so we can use **countOfEmployees** as an alias for that memory block.

## Learn how to declare variables

In Go, variables are **declared** with the **var** keyword followed by the variable name and its type separated by space.

```
package main

func main(){
     var countOfEmployees uint8
}
```

Now let's run the code. You will get the following error:

```
./main.go:5:5: countOfEmployees declared but not used
```

The Go compiler does not allow you to create variables you do not use because it allocates unnecessary memory. Go compiler forces you to use it or delete it. For now, we can print it to the **stdout**. We will import the **fmt** package just like we did in the Hello World application. Try:

```
package main

import "fmt"

func main(){
    var countOfEmployees uint8
    fmt.Println(countOfEmployees)
}
```

You will see no error if you run it. But what is it going to print? We have not assigned any value yet. Let's run and see the output:

```
0
```

It prints zero, but why? When memory blocks are allocated, all bits are zeros by default. That initial value of a type is also called **zero value**. So the **countOfEmployees** variable looks like the following in the memory:

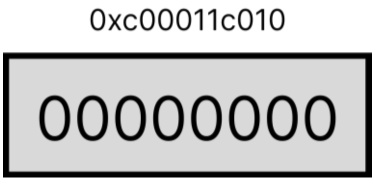


*Figure 4-4. Memory allocation and zero value*

Additionally, note that you do not see the value in binary format in the output; the programming language has already converted it to an integer for you. **In your code, you do not need to manually convert between binary and integer representations.**

**Note:**

Like most other languages, Go allows you to create multiple variables in a single line. The following declaration is also valid.

```
package main


import "fmt"


func main(){
      var countOfEmployees,countOfManagers uint8
      fmt.Println(countOfEmployees,countOfManagers)
}
```

Prefer declaring one variable in a line for more readable code.

How can we see the memory address, then? The memory address allocated by the variable can be accessed with the **&** operator. So, **&countOfEmployees** will give you the address of that variable.

```
package main

import "fmt"

func main(){
      var countOfEmployees uint8
      fmt.Println(&countOfEmployees)
}
```

It will print:

```
0xc00011c010
```

**Note:**

If you run the application repeatedly on your computer, you may see different addresses because, depending on the state of memory at the time of execution, the Operating System will allocate different blocks for the variables.

Lastly, we want to store the value in our variable. The = operator is used to store value in a variable. It is the most important operator, and it is crucial to understand how it works. The = operator calculates the value on the **right-hand side** and **copies** the result to the address of the variable on the **left-hand side.**

```
package main

import "fmt"

func main(){
      var countOfEmployees uint8
      countOfEmployees=26 // assign new value
      fmt.Println(countOfEmployees)
}
```

Now the **countOfEmployees** variable looks like the following in the memory:

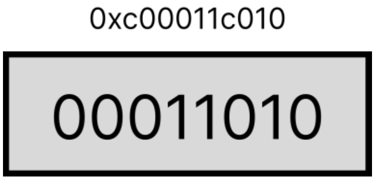


*Figure 4-5. Value stored as bits in the memory block*

**So, you do not need to convert the value to binary in the code.**

For every type, the textual representation of its value, as written in the source code, is called **literal**. For every type you learn, you must also learn its literal; in other words, how to represent a value of that type.

For the **uint8** type, we can directly use the **integer** values as they are.

The value of a variable can be also assigned during its declaration:

```
package main

import "fmt"

func main(){
      var countOfEmployees uint8 = 26
      fmt.Println(countOfEmployees)
}
```

In Go, Integer literals can be written using the following numeral systems.

| **Numeral system** | **Base** | **Starts with** | **Literal** |
|---|---|---|---|
| Decimal | 10 | - | 26 |
| Binary | 2 | 0b or 0B | 0b11010 |
| Octal | 8 | 0 | 032 |
| Hexadecimal | 16 | 0x | 0x1A |

*Table 4-2. Integer literals in different bases*

See how 26 can be written in different numeral systems in the following code:

```
package main

import "fmt"

func main() {
	var decimal uint8 = 26       // Decimal
	var binary uint8 = 0b11010   // Binary
	var octal uint8 = 032        // Octal
	var hexadecimal uint8 = 0x1A // Hexadecimal

	fmt.Println(decimal)     // Output: 26
	fmt.Println(binary)      // Output: 26
	fmt.Println(octal)       // Output: 26
	fmt.Println(hexadecimal) // Output: 26
}
```

**Choose decimal values over another numeral system**. Remember that your code should be readable; others might also read your code. In the examples of the book, decimal values will be used as integer literals.

As you noticed, its type must be stated when a variable is declared. These kinds of languages are **statically typed languages,** meaning that the variable type is checked during compile time, and you are not allowed to assign a value of a different type to that variable.

There are also **dynamically typed languages**. Type checking takes place in the runtime in dynamically typed languages. When you declare a variable, you do not write its type. How does the language treat the variable, then? It is pretty simple: you can assign values of different types to a variable, but the variable's type is the last value's type you assigned.

## Learn whether the language is a statically typed language or dynamically typed language

Go is a statically typed language. Violation of type constraints causes **compilation errors**.

## Copy by value

As said before, the **=** operator copies value in Go as in most languages. It is important to understand the behavior of the operator in order to avoid bugs in your software.

Let's execute the following code:

```
package main

import "fmt"

func main() {
	var myAge uint8
	var votingAge uint8
	votingAge = 18
	myAge = votingAge
	votingAge = 20
	fmt.Println(myAge, votingAge)
}
```

It will print:

```
18 20
```

Even though we wrote `myAge = votingAge`, changing the value of the **votingAge** variable did not change the value of the **myAge** variable. Let's dissect the code.

We created two variables, and they allocated the following memory blocks. **For readability, the following example will display integer values instead of binary values.**

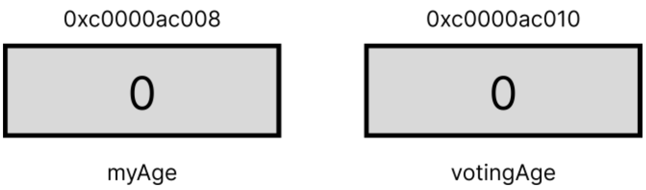


*Figure 4-6. Variables with allocated addresses and zero values*

When we execute `votingAge = 18`, value 18 is written to address of **votingAge** variable.

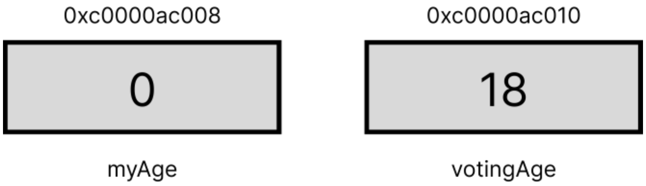


*Figure 4-7. Value stored the address via = operator*

When we execute `myAge = votingAge`, the address of the **myAge** variable does not change. It does not become the address of the **votingAge** variable: **0xc0000ac010**. Addresses of the variables stay as they are. The = operator calculates the value on the right-hand side and copies the value to the address of the **myAge** variable: **0xc0000ac008**.

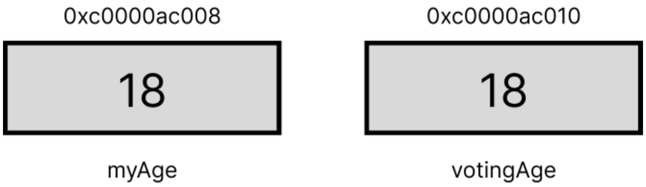


*Figure 4-8. The = operator copies value*

When we execute `votingAge = 20`, the value of the **myAge** variable did not change, and the application printed 18 and 20, respectively.

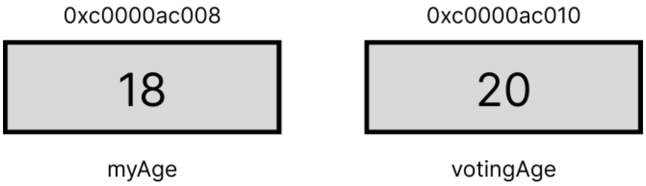


*Figure 4-9. Address remains same after new value is assigned*

## Constants

Some values are not meant to change during runtime, or you may simply prefer to keep them fixed. For example, the number of states in the United States does not change during program execution, and it would be incorrect to modify this value. To represent such unchangeable values, we use **constants**. The main difference between a variable and a constant is that the constant's value must be initialized at the time of declaration, and it cannot be changed afterward. In Go, constants are declared using the **const** keyword instead of **var**, and by convention, their names are written in **PascalCase**.

```
package main

import "fmt"

func main(){
      // must be initialized with the declaration
     const NumberOfStates uint8 = 50
     fmt.Println(NumberOfStates)
}
```

If you try to assign a new value to the constant, you will get a compilation error.

Change the code to the following and run it.

```
package main

import "fmt"

func main(){
     const NumberOfStates uint8 = 50
     NumberOfStates=51
     fmt.Println(NumberOfStates)
}
```

You will get the following error:

```
./main.go:7:16: cannot assign to NumberOfStates
```

**Note:**

**Grouping declarations:** In Go, declarations can be grouped with brackets to avoid repeating keywords. For example:

```
var var1 uint8

var var2 uint8

var var3 uint8

var var4 uint8
```

Declarations above can be grouped as:

```
var (

      var1 uint8

      var2 uint8

      var3 uint8

)
```

The same grouping can be done for the **import**, **const**, and **type** keywords.

## Enumeration

Enumeration is a data type that contains a set of predefined constants. The best example of enumeration can be the seasons in a year. There are four seasons, and the number of seasons will not change. Therefore, it is wise to use enumeration in case of a related set of constants.

**Learn how to create enumeration**

Go does not support enumeration. In Go, you need to define four different constants to simulate enumeration for seasons.

## In A Nutshell

- Variables can be created with the **var name type = literal** syntax.

```
var countOfEmployees uint8 = 26
```

- Choose declaring variables if the value will change in the runtime.
- Chose declaring constants if the value does not change in the runtime.

## Summary

- Memory consists of blocks. Every block stores exactly 8 bits, which is also called **byte**.
- Only integer values are stored in memory, so every value stored in memory must be represented by a number.
- In the source code, we need to declare **variables** to allocate memory. The **type** of the variable determines the number of blocks that will be allocated.
- A value to a variable can be assigned with the = operator.
- The = operator calculates the values on the right-hand side and copies the value to the address of the variable on the left-hand side.
- Languages in which you must specify a type for a variable are called **statically typed languages**; on the other hand, languages that do not require type in the variable declaration are called **dynamically typed languages**.
- **Constants** are the variables for which you can only assign value once in the declaration. Assigning value to a constant variable apart from declaration causes a **compilation error**.
- **Enumerations** are set of related constants. Go does not support enumeration.

## Learn More

- Learn if JavaScript is a statically typed language or dynamically typed language. See how a variable is declared in JavaScript.

# Chapter 5

## Types

You will learn:

- Integers
- Chars
- Strings
- Arrays
- Lists
- Maps
- Functions
- Arithmetic Operations
- Order Of Operations
- String Concatenation
- Overflow

We have learned how to declare variables and constants. Now, it is time to explore other data types, along with their literals and default (zero) values. Most programming languages have similar types, which will be described shortly. However, their names may vary between languages.

### Learn all the types with their zero values and their literals

In the upcoming section on the types, you will be asked questions related to real-life situations to help illustrate which types are used in programming languages to store their answers. Each type will be explained alongside its description.

**Question**: How many siblings do you have?

**Answer**: 2

**Explanation**: The answer cannot be a floating-point number like 3.6; it must be an integer. Therefore, programming languages include a type specifically designed to represent integers.

Integer types in programming languages cannot store all values from minus infinity to plus infinity, as they have defined minimum and maximum limits.

In Go, there are four different integers types that can store both negative and positive integers: **int8**, **int16**, **int32**, and **int64**. Integers type names also end with the number of bits they allocate.

| Type | Bit Size | Minimum Value | Maximum Value |
|---|---|---:|---:|
| uint8 | 8-bit | 0 | 255 |
| uint16 | 16-bit | 0 | 65535 |
| uint32 | 32-bit | 0 | 4294967295 |
| uint64 | 64-bit | 0 | 18446744073709551615 |
| int8 | 8-bit | -128 | 127 |
| int16 | 16-bit | -32768 | 32767 |
| int32 | 32-bit | -2147483648 | 2147483647 |
| int64 | 64-bit | -9223372036854775808 | 9223372036854775807 |

*Table 5-1. Range of integer types*

**Note:**

When you start coding in Go, you will encounter other integer types such as **uint** and **int**. Their sizes depend on the processor architecture of the computer. For example, the **int** type is **int32** on a 32-bit processor and **int64** on a 64-bit processor.

Finally, the names of types may vary between programming languages. For example, Java does not support unsigned integers, and it uses different names for 16-bit and 64-bit integers: **short** and **long**, respectively.

**Zero value**: 0

**Integer Literal**: Any value between the range of the type can be used as a value.

Try the following code:

```
package main

import "fmt"

func main(){
     var zeroValueInteger int
     var siblingCount int = 2
     fmt.Println(zeroValueInteger, siblingCount)
}
```

It will print:

```
0 2
```

**Note:**

In Go, **if you assign a value to a variable during the declaration, you do not have to write the variable type explicitly**. The compiler is smart enough to infer the type from the value on the right-hand side. Therefore, the following declarations are both valid.

```
    var siblingCount int = 2
    var siblingCount = 2
```

Besides, the **:=** operator can be used to declare and initialize a variable concisely without typing the **var** keyword and the variable **type**. We can define the `siblingCount` variable as:

```
    siblingCount := 2
```

**Question**: What is the ratio of a kilometer to a mile?

**Answer**: 0.62137119

**Explanation**: The answer to this question is always a floating number. So, we need a type to represent floating numbers. You can store floating values in **float32** and **float64** types. Float types also use one bit to store the sign, so they are signed types. Just like integers, they have minimum and maximum values. You can see ranges of all float types in Go in the following table. Notice that there are no unsigned types for floats.

| **Type** | **Bit Length** | **Minimum Value** | **Maximum Value** |
|---|---|---:|---:|
| float32 | 32-bit | -3.4e+38 | +3.4e+38 |
| float64 | 64-bit | -1.7e+308 | +1.7e+308 |

*Table 5-2. Range of float types*

Finally, the types' names may change from language to language. In Java, for example, there is a **float** type corresponding to the **float32** type and a **double** type corresponding to the **float64** type in Go.

**Zero value:** 0.0

**Float Literal:** Any floating values between the range can be used as float values.

Try the following code:

```
package main

import "fmt"

func main(){
     var zeroValueFloat float32
     var kilometerMileRatio = 0.62137119
     fmt.Println(zeroValueFloat, kilometerMileRatio)
}
```

It will print:

```
0 0.62137119
```

**Questions**: What is the first letter of your name?

**Answer**: D

**Explanations**: The answer is a single character. Therefore, we need a type to store individual characters. A character does not necessarily mean just English letters from a to z; it can also be numbers, punctuation marks, or other symbols.

Every language might have different ways of dealing with characters. In Go, you have the **rune** type. It is an alias to **int32** type. In most other languages, it is called **char**.

How can we represent a character with an integer? Firstly, we have a set of characters that we want to represent. This set is called a **character set**. We also know that computers only deal with numbers. Therefore, we need to map every character in the set to an integer called a **code point** and use the same code point across computers/languages to get the same character. Representing the characters' code points in bits is called **character encoding**.

Yet, first, there should be a standard agreement as to such a mapping. The ASCII (American Standard Code for Information Interchange) character encoding arrived to meet the expectations in the earlier dates. ASCII assigned numbers to the characters starting from 0 to 127. As you can see from the following table, 65 is assigned to uppercase `A`, and 97 is assigned to lowercase `a`.

| Dec | Char | Dec | Char | Dec | Char | Dec | Char |
|---|---|---|---|---|---|---|---|
| 32 | SPACE | 56 | 8 | 80 | P | 104 | h |
| 33 | ! | 57 | 9 | 81 | Q | 105 | I |
| 34 | " | 58 | : | 82 | R | 106 | j |
| 35 | # | 59 | ; | 83 | S | 107 | k |
| 36 | $ | 60 | < | 84 | T | 108 | l |
| 37 | % | 61 | = | 85 | U | 109 | m |
| 38 | & | 62 | > | 86 | V | 110 | n |
| 39 | ' | 63 | ? | 87 | W | 111 | o |
| 40 | ( | 64 | @ | 88 | X | 112 | p |
| 41 | ) | 65 | A | 89 | Y | 113 | q |
| 42 | * | 66 | B | 90 | Z | 114 | r |
| 43 | + | 67 | C | 91 | [ | 115 | s |
| 44 | , | 68 | D | 92 | \ | 116 | t |
| 45 | - | 69 | E | 93 | ] | 117 | u |
| 46 | . | 70 | F | 94 | ^ | 118 | v |
| 47 | / | 71 | G | 95 | _ | 119 | w |
| 48 | 0 | 72 | H | 96 | ` | 120 | x |
| 49 | 1 | 73 | I | 97 | a | 121 | y |
| 50 | 2 | 74 | J | 98 | b | 122 | z |
| 51 | 3 | 75 | K | 99 | c | 123 | { |
| 52 | 4 | 76 | L | 100 | d | 124 | \| |
| 53 | 5 | 77 | M | 101 | e | 125 | } |
| 54 | 6 | 78 | N | 102 | f | 126 | ~ |
| 55 | 7 | 79 | O | 103 | g |  |  |

*Table 5-3. Some values from ASCII table*

Over time, some characters could no longer be used because they were not included in the ASCII character set. As a result, ASCII was extended to include additional characters, increasing the character encoding to support up to 255 characters.

This expansion was still insufficient because it could not represent many characters from different languages, such as Arabic, Chinese, and others. Today, the Unicode standard provides a unique code point for every character. To be more precise, Unicode version 15 contains over 140,000 code points.

There are also different character encodings, such as UTF-8, UTF-16, and UTF-32. Go uses UTF-8 encoding, which encodes characters using varying numbers of bytes (from 1 to 4 bytes). This is why the **rune** type in Go is based on **int32**, as it can represent any Unicode code point.

To summarize, a **rune** in Go stores a Unicode code point, an **int32**, of a character. If you try to print a rune to the console, you will only see the code point of that character in the Unicode. That also means you can assign a number to a rune if you know which character it corresponds to.

**Zero value:** 0

**Rune Literal**: Values of rune can be represented in two different ways:

- You can directly use the codepoint
- You can write the character in single quotes, and the compiler will replace it with its code point for you.

Try the following code:

```
package main

import "fmt"

func main() {
     var zeroValueRune rune
     var upperCaseDWithCodePoint rune = 68
     var upperCaseD rune = 'D'
     fmt.Println(zeroValueRune, upperCaseDWithCodePoint,
     upperCaseD)
}
```

It will print:

```
0 68 68
```

**Question**: What is the most significant danger to humanity?

**Answer**: Climate change is the most significant danger to humanity because the sea level will rise 0.35 cm in 2 years!!!

**Explanation**: The answers to these questions can be multiple characters, including symbols and numbers. This type is called **string.**

**String Literal:** Strings values are written between **double quotes**.

Try the following code:

```
package main
import "fmt"


func main() {

      var zeroValueString string

      var answerToBiggestDanger = "Climate change is the most
      significant danger to humanity because the sea level will
      rise 0.35 cm in 2 years!!!"

      fmt.Println(zeroValueString)

      fmt.Println(answerToBiggestDanger)

}
```

It will print:

```
Climate change is the most significant danger to humanity because
the sea level will rise 0.35 cm in 2 years!!!
```

**Zero value**: The zero value for the string type is an empty string. It means there is no character. So it zero value is `""`.

```
var zeroValueString string = ""
```

One common problem with string values is to have double quotes inside your strings. Let's store the following sentence in a string variable:

He said "I know how to escape"

Try the following code:

```
package main

import "fmt"

func main() {
     sentence := "He said "I know how to escape""
     fmt.Println(sentence)
}
```

It will print:

```
./main.go:6:24: syntax error: unexpected I at end of statement
```

We got the error because the **double quote is an illegal character in strings**. Because the languages evaluate characters between double quotes as a string ("He said ") and consider the rest (I know how to escape" as part of the code. To use illegal characters in strings, you must **escape** it with a preceding **\** character.

Try the following code:

```
package main

import "fmt"

func main() {
     sentence := "He said \"I know how to escape\""
     fmt.Println(sentence)
}
```

It will print:

```
He said "I know how to escape"
```

**Question**: Are you eligible to vote?

**Answer**: Yes

**Explanation**: The answer to this question can be either yes or no, so we need a type that can store these two options. In Go, this type is called **bool**. In most other languages, it is referred to as **boolean**. The size required to store a boolean value in memory is typically 1 byte.

**Zero value**: Bool variables are **false** by default.

**Bool Literal**: Boolean values can be either **true** or **false**.

Try the following code:

```
package main

import "fmt"

func main() {
	var zeroValueBoolean bool
	var isEligible = true
	fmt.Println(zeroValueBoolean, isEligible)
}
```

It will print:

```
false true
```

All the types covered up to now are called **primitive types**, and they exist in almost every language. Upcoming types are created by using primitive types.

**Question**: Which countries make up the United Kingdom?

**Answer**:

- England
- Scotland
- Wales
- Northern Ireland

The answer to the question is a fixed size (4) of the same type (string). We know that the **size will not change in the runtime**. Languages, including Go, have the **array** type to store fixed size of the same type values.

Arrays are fundamental and one of the most commonly used types in programming languages. Understanding how arrays work is essential for learning programming. Arrays can be likened to buildings in real life. For example, consider a four-story building, where each floor contains one apartment.

*Figure 5-1. Building without floor numbering*

You enumerate the floors starting from 0.

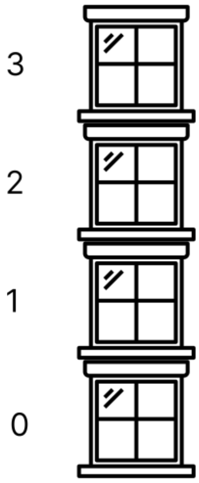


*Figure 5-2. Building with floor numbering*

In the building, the first floor is enumerated 0, and the last floor is enumerated 3. You can access the floors with the number associated with them. This number in the array type is called the **index**.

What happens if someone attempts to access floors that do not exist, such as floor -1 or floor 7? It is not possible in a real building. Similarly, in an array, trying to access an index outside its valid range results in an **index out of bounds** error.

There might be many apartments on a floor.

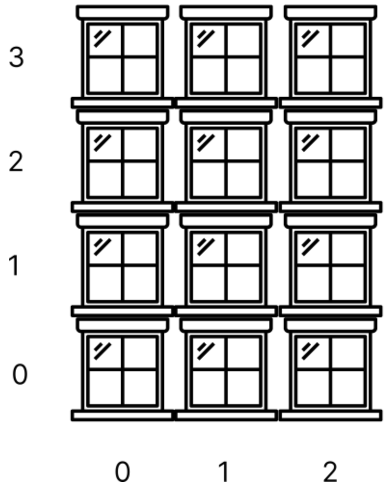


*Figure 5-3. Building with multiple apartments in each floor*

The above image shows four-story buildings with three apartments on each floor. In programming languages, you can create an array of arrays called a **two-dimensional array**. If you want to access the apartment at the top right corner, first, you should go to the third floor and find the second apartment. Two-dimensional array is out of scope of the book.

What if you need more floors? Can you add more floors to the top? Of course not. In such cases, you would need to demolish the existing building and rebuild it with additional floors. The same principle applies to arrays: you cannot increase the length of an existing array. Instead, you must create a new array with the desired, larger size.

To summarize, when you create an array of a certain type with length **n**, you allocate **n consecutive memory blocks**, each of the size of that type. The index of the first element is 0, and the index of the last element is **n - 1**. For example, in the answer to the question, you need to create a string array with a length of 4. You can access the first country using index 0, and the fourth country using index 3.

**Type syntax of array**: In Go, the array type is represented with length written between square brackets followed by the type of values.

Length of array

Type of values
to be stored

```
var countriesInTheUK [4]string
```

**Array Literal:** The value of an array type variable can be initialized with array syntax followed by opening and closing curly brackets. Initial values can be written between square brackets separated by a comma.

```
var countriesInTheUK = [4]string{
      "England",
      "Scotland",
      "Wales",
      "Northern Ireland",
}
```

Values separated
by comma

**Zero value**: In Go, the zero value of an array is an array where each element is set to its respective zero value.

**Using and manipulating array variable:** Value in an array variable can be accessed with the **index** number written between square brackets.

Index of the value

```
var firstCountry string = countriesInTheUK[0]
```

Suppose you need to assign value to a particular index of an array variable. In that case, you should write the index after the variable name between square brackets and use the **=** operator.

Index you want to change

```
countriesInUK[0] = "Englaland"
```

Try the following code:

```
package main

import "fmt"

func main() {
     var ages [4]int
     ages[1] = 32
     fmt.Println(ages)
}
```

It will print:

```
[0 32 0 0]
```

To answer the question, try the following code:

```
package main

import "fmt"

func main(){
     var zeroValueIntegerArray [4]int
     var countriesInUK = [4]string{
            "England",
            "Scotland",
            "Wales",
            "Northern Ireland",
     }
     fmt.Println(zeroValueIntegerArray)
     fmt.Println(countriesInUK)
}
```

It will print:

```
[0 0 0 0]

[England Scotland Wales Northern Ireland]
```

**Note:**

Strings are also an array of characters under the hood. When a new string is created with the following syntax:

```
name := "Alice"
```

It can be considered as:

```
name:= [5]rune{'A', 'l', 'i', 'c', 'e'}
```

Thus, you can treat a string as an array.  For example, you can access the second character by its index.

```
name := "Alice"

fmt.Println(name[1])
```

Sometimes, the number of elements you need to store is not fixed and can change at runtime. For example, consider an application that stores the names of visitors entering a hospital. To do this effectively, you need to add names when visitors arrive and remove them when they leave. Using a fixed-size array is difficult to manage in such cases. Every time you need to store more names than the current array size, you would have to create a larger array, copy the existing data into it, and add new names. Similarly, when visitors leave, you need to delete their names, clear the corresponding array entries, and possibly shrink the array if it becomes much larger than the actual data. Managing arrays this way is cumbersome and prone to errors. Fortunately, many languages offer dynamic, resizable array-like types. These are often called **lists**, though their names may vary across languages. Typically, lists internally use arrays, but they automatically manage the array's size and length, making it easier for developers.

## Learn how to create, add to, or remove from lists

In Go, the **slice** type can be used as a list.

**Type syntax of slice**: In Go, slice syntax is almost similar to array syntax. Go creates a slice when you omit size from the array syntax. Thus, slice type is represented with square brackets followed by the type of values.

Type of values to be stored

```
      var visitors  []string
```

**Slice Literal:** The value of a slice type variable can be initialized with slice syntax followed by opening and closing curly brackets. Initial values should be written between square brackets separated by a comma.

```
      var visitors =  []string{"Alice", "Bob"}
```

**Accessing and manipulating slice variable**: Value in a slice type variable can be accessed with the index number written between square brackets.

Index of the value

```
      var secondVisitor string = visitors[1]
```

If you need to assign value to a certain index of a slice variable, you should write the index after the variable name between square brackets and use the **=** operator.

Index you want to change

```
visitors[1]= "Eve"
```

**Zero value:** The zero value of a slice is **nil**.

**Note:**

**Nil** is a special value in Go. **Nil** value is used to indicate the absence of value. If you try to operate on a value that does not exist, your application will **panic** in Go. Panic will cause the unsafe termination of your application.

In the following example, we have a slice variable, but we have not assigned it a value. We try to access the element at index 0.

```
package main

import "fmt"

func main() {

      var employeesIn []string

      fmt.Println(employeesIn[0])

 }
```

It will print:

```
panic: runtime error: index out of range [0] with length 0
```

Most languages use **null** instead of **nil,** and they throw **NullPointerException** when you try to operate on null variables.

You should operate on slices variables if their values are not **nil** to avoid **panic** or a **NullPointerException** in your application.

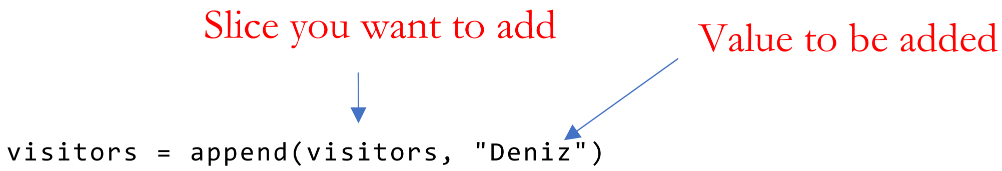
Try the following code:

```
package main

import "fmt"

func main() {
     var zeroValueSlice []string
     visitors:= []string{"Alice", "Bob",  "Eve"}
     fmt.Println(zeroValueSlice)
     fmt.Println(visitors)
}
```

It will print:

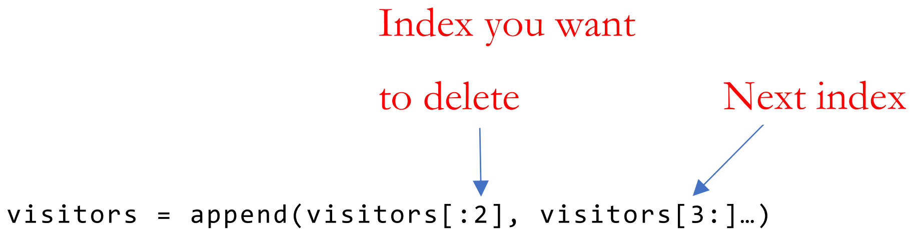
```
[]
[Alice Bob Eve]
```

Adding and removing from a slice can be done with the built-in **append** method in Go. The **append** method takes the slice as the first argument and the values you want to add as the following arguments. It creates a new slice that contains old values and new values. Therefore, you need to assign the new slice to the old slice.



```
visitors = append(visitors, "Deniz")
```

Deleting from the slice looks strange, but for now, you should use the following statement and know it by heart.



```
visitors = append(visitors[:2], visitors[3:]…)
```

Try the following code:

```
package main
import "fmt"
func main() {
	var visitors = []string{"Alice", "Bob", "Eve"}
	// add to slice
	visitors = append(visitors, "Deniz")
	fmt.Println(visitors)
	// delete value at the second index
	visitors = append(visitors[:2], visitors[3:]...)
	fmt.Println(visitors)
}
```

It will print:

```
[Alice Bob Eve Deniz]
[Alice Bob Deniz]
```

**Question**: Suppose you are a teacher, and your students have taken their final exam. After grading the papers, the scores are as follows:

Alice: 85

Bob: 90

Eve:95

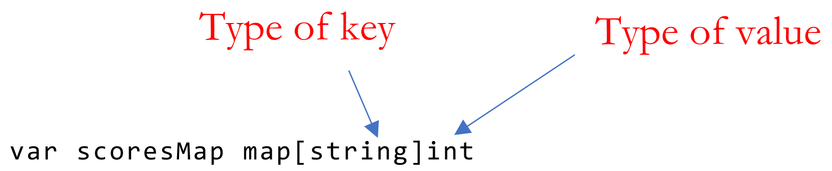
What is Alice’s score in the final?

**Answer**: 85

**Explanation**: We have all the scores. When you are asked a specific student’s score, you can say the score directly. In this case, the name is called key, and the score is called value. This kind of structure is called a **map**. Go also has the type **map**. To create a map, you need two types: one for the **key** and one for the **value**.

You should also be aware that you cannot have two identical keys. When you try to add a key-value pair to the map with an existing key, the map replaces the existing key's value.

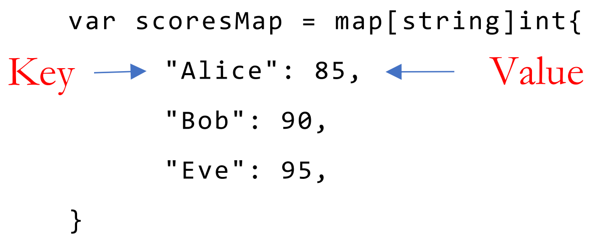
**Type syntax of map**: In Go, the map type starts with the **map** keyword followed by the key type written inside square brackets and value type.



```
var scoresMap map[string]int
```

**Zero value:** The zero value of a map is **nil.**

**Map literal:** The value of the map type variable can be initialized with map syntax followed by opening and closing curly brackets. A colon separates key-value pairs.



```
var scoresMap = map[string]int{
    "Alice": 85,
    "Bob": 90,
    "Eve": 95,
}
```

**Using and manipulating map variable:** Values in a map type variables can be accessed by writing the **key** between square brackets.

Key

```
var scoreOfAlice int = scoresMap["Alice"]
```

If you need to assign value to a specific key of a map variable, you should write the key after the variable name between square brackets and use the = operator.

Key New value

```
scoresMap["Alice"] = 95
```

You can use the built-in delete function to delete key-value pairs from a map.

Map Key

```
delete(scoresMap, "Alice")
```

Try the following code:

```
package main

import "fmt"

func main() {
     var zeroValueMap map[string]int
     var scoresMap = map[string]int{
         "Alice": 85,
         "Bob": 90,
         "Eve": 95,
     }
    fmt.Println(zeroValueMap)
    fmt.Println(scoresMap)
    fmt.Println(scoresMap["Alice"])
}
```

It will print:

```
map[]

map[Alice:85 Bob:90 Eve:95]

85
```

**Note:**

New slices and maps can be created with the built-in **make** function:

```
scoreMap := make(map[string]int)
visitors := make([]string, 0)
```

When slices are created with the built-in **make** function, you must pass an initial length as the second argument.

**Question**: When and where were you born?

**Answers**: 1991 Arapgir

**Explanation**: The question is somewhat tricky because the answer involves two different types we have learned: integer and string. In programming languages, there are composite types that can store multiple variables within a single entity. In Go, this type is called a **struct**; in other languages, it is known as a **class**. In short, structs are types that contain **multiple variables** inside, allowing you to group related data together.

We will create a struct called **Birth Information** to store the answer to the question.

**Type syntax of struct**: In Go, the struct type starts with the **struct** keyword, followed by opening and closing brackets. Inside the brackets, you write the **fields** by name and type, separated by a single space, and every field is written in a new line.

```
         var birthInformation struct {
Field Name  →  Place string  ←  Field Type
               Year int
         }
```

**Zero value:** When you create a variable of a struct type, all the fields of the struct will have zero values of their types.

**Struct Literal:** The value of a struct type (**instance**) variable can be initialized with struct syntax, followed by opening and closing curly brackets. A colon separates fields and values, and a comma separates value pairs.

In programming, **a struct is like a blank form or template that defines the fields and structure needed to organize data**, but it doesn't contain any specific information itself. **An instance of that struct is like a filled-out form, where each field has been given actual values**—such as place and year—representing a specific example of the template. **Essentially, the struct provides the blueprint for data, while the instance is a particular set of data filled into that blueprint.**

```
      var birthInformation = struct {
            Place string
            Year int
      }{
Field Name → Place: "Arapgir", ← Field Value
                  Year: 1991,
      }
```

Try the following code:

```
package main
import "fmt"
func main() {
      var zeroValueStruct struct {
            Place string
            Year int
      }

      var birthInformation = struct {
          Place string
          Year int
      }{
          Place: "Arapgir",
          Year: 1991,
      }

      fmt.Println(zeroValueStruct)
      fmt.Println(birthInformation)
}
```

It will print:

```
{ 0}
{Arapgir 1991}
```

## Creating New Type

As shown in the code above, we need to specify all the field names explicitly whenever we work with a struct type. If the struct has ten fields, this can lead to repeated code, which can become cumbersome. Wouldn't it be much better to declare a custom type and use it just like other basic types such as **string** or **int64**?

In Go, new types can be created with the **type** keyword.

```
type BirthInformation struct {

     Place string

     Year int

}
```

After the type is created, we can use it in a variable declaration, like the other types.

Try the following code:

```
package main

import "fmt"

func main() {
    type BirthInformation struct {
        Place string
        Year int
    }
    var birthInformation = BirthInformation{
        Place: "Arapgir",
        Year: 1991,
    }
   fmt.Println(birthInformation)
}
```

New type

It will print:

```
{Arapgir 1991}
```

**Question**: How should I notify the customers via their phone number about the package delivery?

**Answer**: You should send an SMS to the phone number.

The answer to the question is different from any previous answer. In this case, we need to define a behavior or operation. Programming languages have **function** types that perform some operations.

Functions may need inputs called **parameters** to perform their operations. Parameters are basically **variables** created in the function under the hood. Functions may also send back some values calculated during operation, called **return values**.

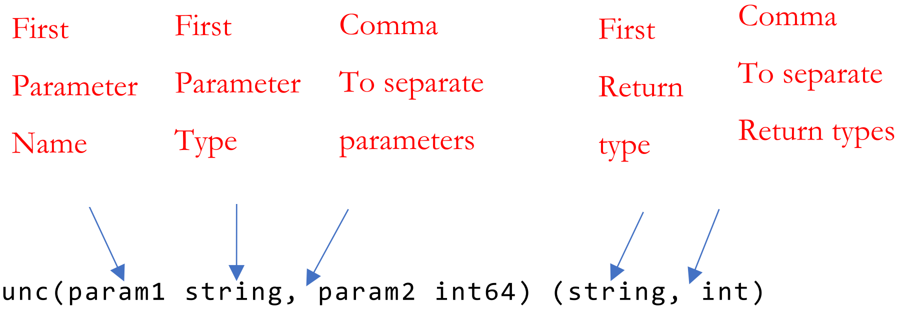
Some languages allow you to treat functions like any other primitive type, which means they can be stored in variables and passed as values. Programming languages are then said to have **first-class functions**.

Some languages heavily use functions and are designed based on functions. These languages are said to be **functional languages**.

## Learn whether your language is functional and has first-class functions

Go is not a fully functional language. However, it has some functional properties. One of them is that functions in Go are **first-class functions**.

**Type syntax:** Function type syntax starts with the **func** keyword and two opening and closing parentheses pairs. **Parameters** are written by their name and type format in the first parentheses, separated by a comma. Inside the second parentheses, **return types** are written and separated by commas. There is no limit to the number of parameters and return types.



```
func(param1 string, param2 int64) (string, int)
```

Functions do not have to return any value. In this case, you do not have to write second parentheses.

```
func(param1 string, param2 int64) // does not return anything
```

If your function returns only one type, you may omit second brackets. By conventions, they are not written.

```
func(param1 string, param2 int64) string // returns single value
```

**Note:**

In most programming languages, a function must return **at most one** value. However, in Go, it is possible to return multiple values from a single function.

**Function Literal:** The value of a function type variable can be initialized with function syntax followed by opening and closing curly brackets.

If the function has return types, you must return **values** in the function body using the **return** keyword. A function must return the same number of values in the same order defined in the function's return types.

```
var calculateAreaOfSquare = func(sideLength int) int {
     var area = sideLength * sideLength
     return area
}
```

Function Body

**Note:**

Function names start with a verb by convention and describe what it does.

**Execution of functions:** You can execute a function and retrieve its return values within the function body. This process is called "**calling**" a function. To call a function, you write the function's name followed by parentheses, and inside the parentheses, you include values in the order defined by the function's parameters. The values you pass to the function are called **arguments**.

Arguments can be:

- Value
- Variable
- Value returned from a function call

For example, **calculateAreaOfSquare** can be called as follows:

Value

```
calculateAreaOfSquare (2)
```

Variable

```
var length = 5
squareAreaCalculator(length)
```

Function returning value

```
var getValue = func() int {
      return 5
}
squareAreaCalculator(getValue())
```

Arguments **cannot** be statements or declarations.

```
squareAreaCalculator(var a = 5) // causes compilation error
```

You can get the returned values from the function call by assigning them to the variables.

```
var a int
a = squareAreaCalculator(2)
```

If the function you call returns more than one value, you can assign it to two variables separated by a comma.

```
var functionWithTwoReturns = func(firstParam int, secondParam
bool)(int, bool) {
      return 5, false
}
var firstValue int
var secondValue bool
firstValue, secondValue = functionWithTwoReturns(2, false)
```

Variables to store returned values can also be created with the **:=** operator.

```
firstValue, secondValue := functionWithTwoReturns(2, false)
```

**Note:**

If you do not want to use a function's returned value, you can replace it with _ to ignore it in Go.

```
firstValue, _ := functionWithTwoReturns(2, false)
```

In the above code, secondValue is ignored.

**Zero value:** Zero values of a function is **nil**.

Try the following code:

```
package main

import "fmt"

func main() {
     var zeroValueFunction func(phoneNumber string)
     // send message function
     var sendMessageTo = func(phoneNumber string) {
           fmt.Println("sending sms to " + phoneNumber)
     }

      // make call function
     var makeCallTo = func(phoneNumber string) {
            fmt.Println("calling " + phoneNumber)
     }

     fmt.Println(zeroValueFunction)
     phoneNumber := "11111111"
     sendMessageTo(phoneNumber)
     makeCallTo(phoneNumber)
}
```

It will print:

```
<nil>

sending sms to 11111111

calling 11111111
```

## Casting

Sometimes, you may need to convert a variable or value from one type to another. In the following example, we have a function that calculates the area of a circle. The function accepts the radius as a **float64**. However, we have an integer variable as radius.

```
package main

import "fmt"

func main() {

      var calculateAreaOfACircle = func(radius float64) float64 {

            var area = 3.14 *radius * radius

            return area

      }

      var radiusAsInteger =2

      area :=calculateAreaOfACircle(radiusAsInteger)

      fmt.Println(area)

}
```

It will print:

```
./main.go:11:32: cannot use radiusAsInteger (variable of type int)
as float64 value in argument to calculateAreaOfACircle
```

The integer value 2 is also valid for the types: int32, int64, float32, and float64. Therefore, we need to convert the integer value to a float. Conversion from one type to another is called **casting**. In Go, type casting is performed explicitly using the syntax **type(value)**. See the following:

```
package main
import "fmt"
func main() {
    var foo int = 2
    var bar float64 = float64(foo) // Casting int to float64
    fmt.Println(bar) // Output: 2.0
}
```

Now, we can pass an integer variable to the **calculateAreaOfACircle** function by casting it into float64. See the following code:

```
package main
import "fmt"
func main() {
      var calculateAreaOfACircle = func(radius float64) float64 {
            var area = 3.14 *radius * radius
            return area
      }
      var radiusAsInteger =2
      area :=calculateAreaOfACircle(float64(radiusAsInteger))
      fmt.Println(area)
}
```

It will print:

```
12.56
```

**Note:**

If you attempt to cast incompatible types to each other, you will encounter a **compilation error**.

```
bool(1)
int("foo")
```

**Note:**

In some languages, you do not have to explicitly cast types in your code, as the language handles the conversion automatically. This feature is called **type coercion**.

## Arithmetic Operations

Arithmetic operators allow you to perform mathematical operations on numerical types such as integers and floats.

| Operator | Description |
|---|---|
| + | Addition or Plus Sign |
| - | Subtraction or Minus Sign |
| * | Multiplication |
| / | Division |
| % | Modules |
| ++ | Increment by one |
| -- | Decrement |

*Table 5-4. Arithmetic operators*

Try the following code:

```
package main
import "fmt"

func main() {
      a := 15
      b := 8
      fmt.Println("Addition: ", a+b)
      fmt.Println("Subtraction: ", a-b)
      fmt.Println("Multiplication: ", a*b)
      fmt.Println("Division: ", a/b)
      fmt.Println("Modulus: ", a%b)
      a++ // is equal to writing a=a+1
      b-- // is equal to writing b=b-1
      fmt.Println("Increment: ", a)
      fmt.Println("Decrement: ", b)
}
```

It will print:

```
Addition:  23
Subtraction:  7
Multiplication:  120
Division:  1
Modulus:  7
Increment:  16
Decrement:  7
```

Note that 15/8 should result in 1.875, but it prints as 1**. The reason is that dividing two integers yields an integer result as well—specifically, the fractional part is discarded**. For example, 18/5 evaluates to 3, not 3.6. You will see the fractional part if you use float types instead of integers. See the following example:

```
package main
import "fmt"

func main() {
      a := 15.0
      b := 8.0
      fmt.Println("Division: ", a/b)
}
```

It will print:

```
Division:  1.875
```

You can use the + and – operators as the signs of the numerical values. See the following example:

```
package main
import "fmt"

func main() {
      a := +15.0
      b := -8.0
      fmt.Println("Division: ", a/b)
}
```

It will print:

```
Division:  -1.875
```

## Order Of Operations

The same rules from mathematics regarding the precedence of operations also apply in programming languages. See the following example:

```
package main
import "fmt"
func main() {
      result := 6 / 2 * (1 + 2)
      fmt.Println(result)
}
```

It will print:

```
9
```

## String Concatenation

The + operator also concatenates strings to create new strings. See the following example:

```
package main
import "fmt"

func main() {
    str1 := "Hello"
    str2 := "World"
    result := str1 + " " + str2 // Concatenating strings
    fmt.Println(result) // output will be Hello World
}
```

It will print:

```
Hello World
```

## Overflow

What happens if we assign a value to a numeric type larger than it can store? This behavior is called **overflow**. The program continues to work when an overflow happens. In case of an overflow,

- If the value reaches more than the type’s maximum values, then the value is set to the type’s minimum value.
- If the value reaches less than the type’s minimum values, then the value is set to the type’s maximum value.

See the following example:

```
package main
import "fmt"

func main() {
    // Let's use a small int8 type having range of -128 to 127
    var maxInt8 int8 = 127
    var minInt8 int8 = -128
    maxInt8++ // This will cause an overflow maxInt8 is now -128
    minInt8-- // This will cause an overflow, minInt8 is now 127
    fmt.Println(maxInt8,minInt8)
}
```

It will print:

```
-128 127
```

**It is the developer's responsibility to check for overflow**. Make sure the chosen type is large enough to store the value you intend to assign to the variable. For example, if you want to store an integer value larger than the maximum value of **int64**, you can use big types, which are implemented separately in almost every programming language. However, these big types are outside the scope of this book.

## In A Nutshell

You can decide which type to choose for the values you want to store based on the following criteria:

- Use one of the **primitive types** if you have **one value.**
- Use an **array** if you have **fixed number of values** and it does not change during the runtime.
- Use **slice** if you do not know how many values you want to store, and the size might change in the runtime.
- Use **struct** if you need to store multiple related variables in a single unit.
- Use **map** if you have key-value pairs and you want to access a value with its key.
- Use **function** if you need to store **behavior** or  you want to reuse the same behavior in your code.

## Summary

- **Float** types are used to store floating numbers.
- **Rune** is a type that stores only one character. In the language's encoding system, every character has an integer value called a **code point**.
- **Strings** can contain many characters. It should be considered as a list of characters. To use illegal characters like double quotes inside a string, you need to escape it with \ in your code.
- **Boolean** stores only **true** and **false**.
- **Array** type stores multiple values of the same type with a fixed size called **length**. The location of an item in an array is called an **index**. Indexes range between 0 and length-1.
- An array storing another array as values is called a **two-dimensional array**.
- The **slice** type stores multiple values of the same type without any fixed size. The built-in **append** function inserts or removes a item.
- **Nil** indicates the absence of value. Operating on nil values causes **panic** and terminates the application.
- The **map** type stores key and value pairs. To remove an item from the map, use the built-in delete function.
- The built-in **make** function is used to create maps and slices.
- **Struct** is the type that can store many types inside. Struct items are called fields.
- Value of a **struct** is called an **instance**. The **struct** is like the **blank form**, defining what information is needed while the **instance** is like the **filled form**, containing the actual values for a specific case.
- The **type** keyword is used to create a new type.
- A **function** executes some statements. Functions get values called **parameters** from the caller and might **return** some values back to the caller.
- Some languages treat functions like any other variable. Those functions are called **first-class functions**.

- Executing a function is called **calling**. When a function is called, values must be passed to every parameter. Those values are called **arguments**.
- Converting a variable or value from one type to another is called **casting**.
- **Arithmetic operators** perform mathematical operations on numeric data types.
- The **order of operations** in programming languages **is the same as in mathematics.**
- The **+** operator **concatenates** strings.
- If the value of a numeric data type is set to a value larger than its maximum value, **overflow** happens.

## Learn More

- Learn more about escaping in strings.
- Learn how floating numbers are stored in the memory.
- Read more about the order of operations in the programming languages.
- Read about functional languages.
- Read about first-class functions.

# Chapter 6

## Scopes

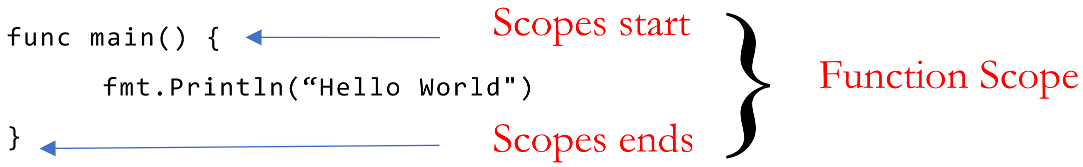
You will learn:

- Scopes
- Block Scope
- Shadowing
- Garbage Collection
- Global Scope
- Anonymous Functions
- Named Functions
- Function Signatures
- Methods
- Packages
- Encapsulation

In programming languages, scope refers to the areas where variables, types, and functions can be accessed. The definition and rules of scope may vary between languages. Additionally, not all languages implement scope in exactly the same way. However, they share important similarities in how scope is handled.

## Learn the scopes in the language

You should know that **every matching curly bracket creates a new scope** in most programming languages. For example, every function creates a new scope.



```
func main() {
      fmt.Println(“Hello World")
}
```

Scopes is the context where you can declare types and variables and access them. **Inside the same scope, you cannot create variables and types with the same name.**

```
package main
import "fmt"
func main() {
      var firstVariable = 1
      var firstVariable = 2
      fmt.Println(firstVariable)
}
```

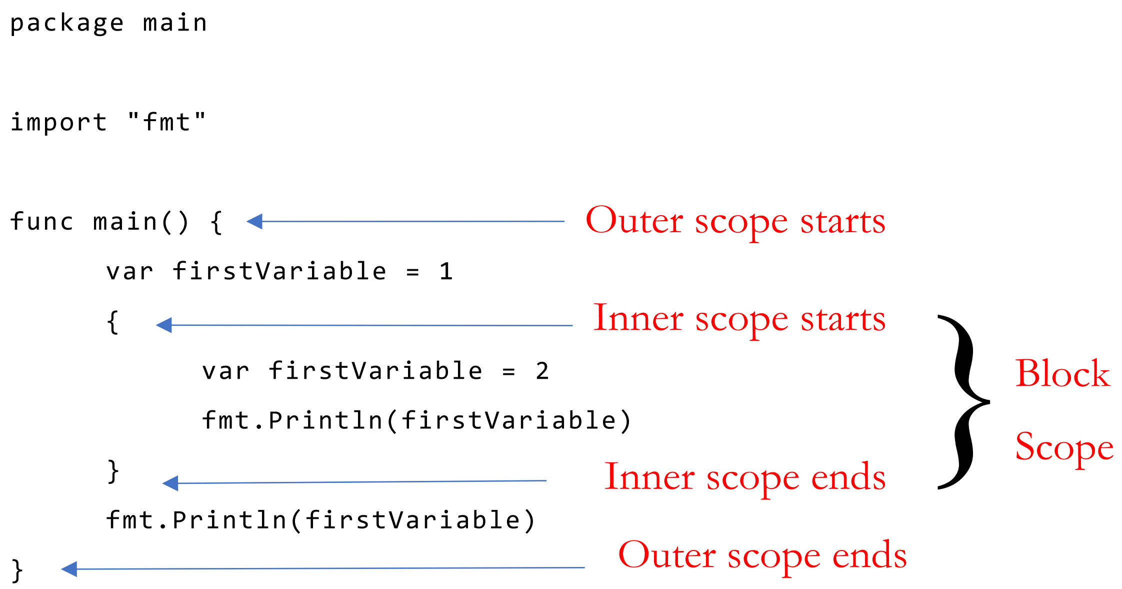
If you run the above code, you will get the following error because the compiler cannot identify which variable to use.

```
./main.go:7:6: firstVariable redeclared in this block
./main.go:6:6: other declaration of firstVariable
```

**Note:**

To increase the readability of your code, create an indentation with a tab whenever you create a new scope with curly brackets.

You can create another scope with matching curly brackets and declare new variables/types inside the new scope, which is called **block scope**.



```
package main

import "fmt"

func main() {
      var firstVariable = 1
      {
            var firstVariable = 2
            fmt.Println(firstVariable)
      }
      fmt.Println(firstVariable)
}
```

If you run the following code, it will print:

```
2
1
```

Even though we have **firstVariable** in the outer scope, we can use the same variable name inside the inner scope because every scope has its context. The compiler checks if you declared the same variable inside the same context. Hence, we can use the same name in two different blocks.

If you declare a variable inside an inner scope with a name that already exists in the outer scope, the variable is **firstVariable**; it is said that the variable in the inner scope is **shadowing** the variable in the outer scope. When you access a shadowed variable, you access the variable defined in the inner scope.

When a scope ends, **the types and variables created inside it are destroyed**, meaning **they are not accessible from the outer scope**. You will get a **compilation error** if you try to access a variable/type declared inside an inner scope from the outer scope.

```
package main

import "fmt"

func main() {
	outerScopeVariable := 1
	{
		innerScopeVariable := 2
		fmt.Println(innerScopeVariable)
		fmt.Println(outerScopeVariable)
	}
	fmt.Println(innerScopeVariable)
}
```

You will get the following compilation error:

```
./main.go:15:14: undefined: innerScopeVariable
```

When we tried to access the variable defined in the inner scope from the outer scope, we got the **undefined** error because the inner scope had already ended, and the variables were not accessible anymore. However, what happened to the memory allocated for the variable inside the inner scope? That memory block became **garbage**. It should be released so that it can be used again. The process of releasing inaccessible memory blocks is called **garbage collection**.

## Learn how to do garbage collection

In Go, garbage collection is managed by the language itself. As developers, you do not need to do anything in the code.

Let's remove the last line and run it again:

```
package main
import "fmt"

func main() {
      outerScopeVariable := 1
      {
            innerScopeVariable := 2
            fmt.Println(innerScopeVariable)
            fmt.Println(outerScopeVariable)
      }
}
```

It will print:

```
2
1
```

All the variables and the types we declared so far were within the scope of the main function. Do we have to write all the code to the main function? Of course not. We can define variables or types that can be accessed in the whole application. Those definitions are done in the **global scope**. If you write your declaration on the same level you use the **package** keyword **outside the functions**, those variables/and types can be accessed globally **throughout the package and anywhere in the program**.

```
package main

import "fmt"

var globalVariable = 1

func main() {
      fmt.Println(globalVariable)
}
```

It will print:

```
1
```

So far, when we created functions, we assigned them to variables. A function created this way is called an **anonymous function**.

**Remember that you can assign another value to an anonymous function in the runtime.** See the following example:

```
package main
import "fmt"
// an anonymous function is defined
var printMessage = func() {
      fmt.Println("printMessage is called")
}
func main() {
      printMessage()
      // a new value can be assigned because it is a variable
      printMessage = func() {
            fmt.Println("printMessage function is changed")
      }
      printMessage()
}
```

It will print:

```
printMessage is called
printMessage function is changed
```

Alternatively, in the global scope, functions can also be declared with the **func** keyword followed by the **function name, parameters,** and **return types.** A function created this way is called a **named function. Unlike the anonymous function**, **named functions** cannot be reassigned. See the following example:

```
package main
import "fmt"

func main() {
      printMessage()
}
func printMessage() { // named function declared in global scope
      fmt.Println("print is called")
}
```

It will print:

```
print is called
```

If you try to assign a new value to a named function, you will get a compilation error. Try the following code:

```
package main

import "fmt"

func main() {
      printMessage = func() {
            fmt.Println("printMessage is called")
      }
}

func printMessage () { // named function
      fmt.Println("print is called")
}
```

It will print:

```
./main.go:6:2: cannot assign to printMessage (neither addressable
nor a map index expression)
```

**In your code, choose the named function over an anonymous function unless you want to assign new values in the runtime.**

**Note:**

Define functions and types in the global scope so that you can share them with other functions.

**Note:**

The function name combined with parameters and return types without its body is called **function signature**.

## Methods

Some functions might be tightly coupled with structs. Take the following as an example:

```
package main

import "fmt"

type Student struct {
	TestScores [2]float32
}
func main() {
	alice := Student{TestScores: [2]float32{65, 92}}
	passScore := float32(75.0)
	hasAlicePassed := hasPassed(alice, passScore)
	fmt.Println(hasAlicePassed)
}

func hasPassed(student Student, passScore float32) bool {
	totalScore := student.TestScores[0] + student.TestScores[1]

	return totalScore/2 >= passScore
}
```

The **student** type has the **TestScores** field, and there is the **hasPassed** function, which determines whether a student passed the course. The **hasPassed** function always requires a **student** struct as input to calculate whether the student's average score is greater than or equal to **passScore**.

Instead of passing the **student** struct as a parameter in every call, it is preferable to associate the function with the struct, allowing the function to access the struct's fields directly. Functions that are associated with structs are called **methods**.

### Learn how to write methods

A **method** can be associated with a struct by adding a **receiver** written between parentheses after **func** keyword and before the **function**

**name.** In Go, a **receiver** is defined as the syntax of a variable: receiver name followed by receiver type. **A receiver is a variable that holds the instance on which the method is called.**

Receiver name

Receiver type

```
func (student Student) hasPassed(passScore float32) bool
```

You can call methods on a struct using **dot notation**, similar to accessing a struct field.

Call the method by putting dot after variable name

```
hasAlicePassed := alice.hasPassed(passScore)
```

**Fields of the struct** can be accessed within the method from the **receiver.** Inspect the following code:

```
package main

import "fmt"

func main() {
	alice := Student{TestScores: [2]float32{65, 92}}
	passScore := float32(75.0)
	hasAlicePassed := alice.hasPassed(passScore)
	fmt.Println(hasAlicePassed)
}
type Student struct {
	TestScores [2]float32
}

func (student Student) hasPassed(passScore float32) bool {
	totalScore := student.TestScores[0] + student.TestScores[1]

	return totalScore/2 >= passScore
}
```

**Note:**

Receivers are unique to the Go language. Other programming languages use different mechanisms to access the current instance. For example, Python uses the **self** keyword, while Java, C++, and C# use the **this** keyword.

As the application grows, it is wise to organize the code so that related types, variables, constants, and functions are grouped together logically. A collection of code files organized in this manner is called a package. Packages also help developers find code quickly and enhance overall readability.

**Learn how to create packages**

## Packages

In Go projects (**modules**), each directory represents a package. A package contains multiple Go files, and all files within the same package must start with the same package name. By convention, the directory name is used as the package name.

First, create a project called **management-app**. Within it, create a directory named **school**, and inside the **school** directory, add the files **course.go** and **student.go**.

```
management-app
        ├── main.go
        └── school
            ├── course.go
            └── student.go
```

Let’s create the student type in student.go:

```
package school

type Student struct {
      Name string
      TestScores [2]float32
}
```

```
func (student Student) HasPassed(passScore float32) bool {
	totalScore := student.TestScores[0] + student.TestScores[1]

	return totalScore/2 >= passScore
}
```

Let’s create the course type in course.go:

```
package school
import "fmt"

type Course struct {
	Name string
}

func (course Course) registerStudent(student Student) {
	fmt.Println(student.Name + " is registered to course " +
	course.Name)
}
```

The **school** package contains two Go files. Both files must begin with the same package name. If you do not use the same package name in all files within the same package, you will encounter a compilation error.

Notice that you can access the **student** type from **course.go** because they are in the same package.

**Each package creates its own global scope**. This means you cannot define the same types and variables multiple times within the same package; names must be unique within the package.

How can we use these types inside another package such as main?

## Learn how to use resources from other files and packages

If you want to use resources defined in another package, you must import them using the **import** keyword. So, how does Go locate these packages? First, you need to understand how Go organizes its projects (**modules**).

Go manages modules and binaries in a specific directory called **workspace** and is defined as in (https://go.dev/doc/gopath_code):

A workspace is a directory hierarchy with two directories at its root:

- **src** contains Go source files
- **bin** contains executable commands

The location of the Go workspace is stored in the **GOPATH** environment variable. To view the value of **GOPATH**, the **echo** command can be used as follows:

```
/projects/src # echo $GOPATH
/projects
/projects/src #
```

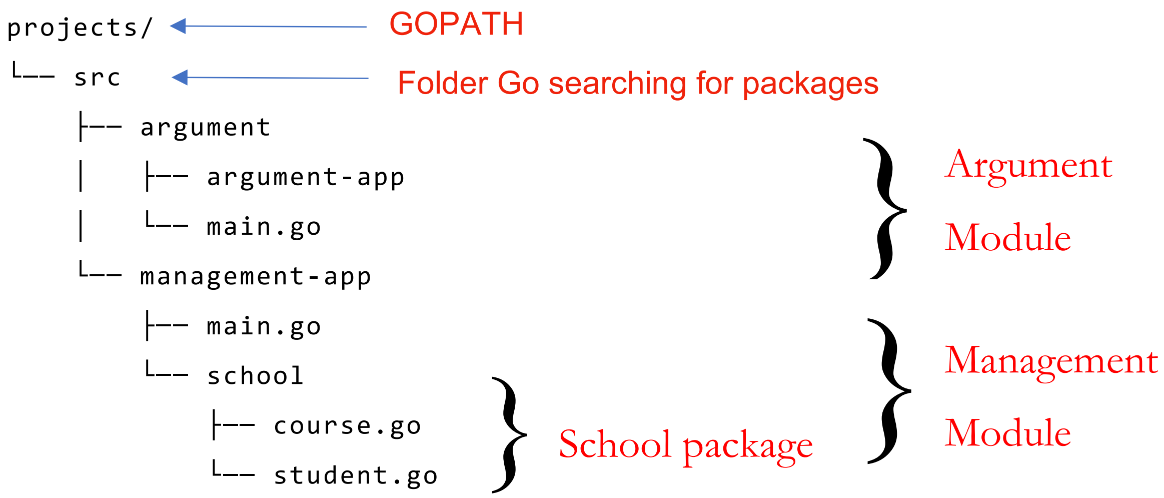
To learn more about how **GOPATH** works and what its stores run **go tool gopath** command.

```
/projects/src # go help gopath
The Go path is used to resolve import statements.
It is implemented by and documented in the go/build package.

…

/projects/src #
```

Go modules are created inside the **src** file under the **GOPATH**, and you can only import from the modules which are inside **$GOPATH/src** directory.

```
projects/
└── src
    ├── argument
    │   ├── argument-app
    │   └── main.go
    └── management-app
        ├── main.go
        └── school
            ├── course.go
            └── student.go
```

To use resources from another package, you should specify its path relative to **$GOPATH/src** after the **import** keyword. For example, if you want to use the **Student** type from the **school** package, you need to import its directory path: **management-app/school**.



```
projects/
└── src
    ├── argument
    │   ├── argument-app
    │   └── main.go
    └── management-app
        ├── main.go
        └── school
            ├── course.go
            └── student.go
```

Let’s create two students and one course and register the students for the course in **main.go**. First, we will import the package and attempt to create a student.

```
package main

import "management-app/school"

func main() {
      alice := Student{Name: "Alice"}
}
```

Let’s run the application:

```
./main.go:4:2: undefined: Student
```

Go could not find the student type even though we imported it the package. Why?

Go searches for the type in the current package when you use a type. If you want to use variables or types from another package, you must prefix them with the package name using dot notation. Therefore, the correct way to use the **Student** type from the **school** package is **school.Student**. Let's specify the package name before **Student** and create all variables accordingly:

```
package main

import "fmt"
import "management-app/school"

func main() {
	alice := school.Student{Name: "Alice"}
	bob := school.Student{Name: "Bob"}
	programmingCourse := school.Course{Name: "Go programming"}
	fmt.Println(alice)
	fmt.Println(bob)
	fmt.Println(programmingCourse)
}
```

Let’s run the application:

```
{Alice [0 0]}
{Bob [0 0]}
{Go programming}
```

Finally, we were able to use types from another package. Now, let's try registering students for the programming course using the **registerStudent** method.

```
package main
import "management-app/school"

func main() {
	alice := school.Student{Name: "Alice"}
	bob := school.Student{Name: "Bob"}
	programmingCourse := school.Course{Name: "Go programming"}
	programmingCourse.registerStudent(alice)
	programmingCourse.registerStudent(bob)
}
```

Let’s run the application:

```
./main.go:10:20: programmingCourse.registerStudent undefined (type
school.Course has no field or method registerStudent)
./main.go:11:20: programmingCourse.registerStudent undefined (type
school.Course has no field or method registerStudent)
```

Even though the **course** type has the **registerStudent** method, Go could not find it. Why? Because sometimes, you may want to prevent other packages from modifying or accessing your package's variables, types, fields, functions, and methods. This concept is called encapsulation. Encapsulation is a mechanism that determines who can access data and how that data can be manipulated.

## Learn how to encapsulate

In some languages, encapsulation is achieved using keywords like **private**, **public**, and **protected**. In Go, these keywords do not exist. Instead, if you want your data to be accessible or modifiable from outside your package, you need to export them. Otherwise, they remain unexported. How can we export data in Go?

In Go, if a resource’s name starts with a capital letter, it is exported and accessible from other packages. If it begins with a lowercase letter, it is unexported and only accessible within its own package. Let’s take another look at the **registerStudent** method of the **course** type.

```
func (course Course) registerStudent(student Student) {
      fmt.Println(student.Name + " is registered to course " +
      course.Name)
}
```

Since the method name starts with a lowercase 'r', it cannot be called from outside the package. To fix this, we need to capitalize the method name.

```
func (course Course) RegisterStudent(student Student) {
      fmt.Println(student.Name + " is registered to course " +
      course.Name)
}
```

Let’s call the exported method from main:

```
package main

import "management-app/school"

func main() {
	alice := school.Student{Name: "Alice"}
	bob := school.Student{Name: "Bob"}
	programmingCourse := school.Course{Name: "Go programming"}
	programmingCourse.RegisterStudent(alice)
	programmingCourse.RegisterStudent(bob)
}
```

We will get the following output when you re-run main:

```
Alice is registered to course Go programming
Bob is registered to course Go programming
```

## Summary

- Every matching curly bracket creates a **scope**.
- Inside the same scope, the same variable or type names cannot be used twice.
- Inside a scope, another scope can be created. Variables declared in an inner scope cannot be accessed from an outer scope.
- Declaring the same variable name that is used in an outer scope is called **shadowing**.
- Variables are destroyed when the scopes that they belong end.
- An inaccessible memory block is called **garbage**, and the process of releasing the garbage memory blocks is called **garbage collection**.
- **Function signature** is the combination of function name, parameters, and return types.
- Functions attached to a type are called **methods**. A method for a struct can be defined with the **receiver**. The **receiver** is a variable storing the instance of a struct from which the **method** is called.
- **Package** is a directory containing many go files inside.
- All go files inside a package must have the same package name.
- **Global scope** is a package-level scope.
- **GOPATH** is the environment variable where the go workspace is.
- Go modules should be created under **$GOPATH/src** directory.
- Modules under **$GOPATH/src** can be imported by other modules using the path under the src folder.
- Struct fields and methods can be accessed with the dot notation.
- **Encapsulation** is to restrict access to resources from other packages. In Go, fields, variables, or functions whose names start with a capital letter are exported and can be accessed from

other packages. Otherwise, those resources can only be accessed in the same package.

- When exported types are used in different packages, the package should be imported, and types should adhere to their package, such as io.Reader, fmt.Formatter.

## Learn more

- Create a teacher project under $GOPATH/src. New project should have a teacher package. Teacher struct should be defined in the package. Teacher struct should have the name, surname, age, and email fields. All fields should be exported. Write a FullName method, which returns the name and surname combined with a space. Create another module, and in its main function, use the teacher type from the first module. Create a teacher with all fields filled out and print the full name on the terminal.

# Chapter 7

## Flow of control

You will learn:

- Relational Operators
- If statements
- Logical Operators
- Loops
- Range keyword

Statements inside a function are executed sequentially from top to bottom.

```
package main

import "fmt"
import "time"
import "math/rand"

func main() {
      rand.Seed(time.Now().UnixNano())
      age := rand.Intn(100)
      fmt.Println(age)
}
```

Executes from top to bottom

It will print:

```
81
```

The **age** variable is assigned a random value generated by the **rand.Intn** function. The function call **rand.Intn(100)** returns a random integer between 0 and 99.

If you want to skip the execution of some statements based on specific conditions, you need to control the flow of execution. In the following example, we should print "you can vote" if the age is greater than or equal to 18. Otherwise, we should print "you cannot vote"

```
package main

import "fmt"
import "time"
import "math/rand"

func main() {
      rand.Seed(time.Now().UnixNano()
      age := rand.Intn(100)
      fmt.Println(age)
      fmt.Println("you can vote")
      fmt.Println("you cannot vote")
}
```

Firstly, we need to evaluate the **condition** that the value of the age variable is greater than or equal to 18. How can we ask this question to a computer?

## Asking questions with relational operators

You can ask questions to computers using relational operators, and the computer's response will always be a boolean value (**true** or **false**). In short, the answer will be either yes or no. **Relational operators** work by comparing the operands and evaluate to either **true** or **false**. For example, to check if 10 equals 3, we use the **==** relational operator to make this comparison.

`10 == 3` → false

> **Note:**
>
> Do not confuse the **==** operator with the **=** operator. The **=** operator copies the value of the right-hand side to the variable on the left. In contrast, the **==** operator compares whether the left and right operands are equal and returns **true** or **false** based on the comparison.

Other operators are:

| **Operator** | **Description** |
|---|---|
| == | It checks if the values of two operands are equal or not; if yes, the condition becomes true. |
| != | It checks if the values of two operands are equal or not; if the values are not equal, then the condition becomes true. |
| > | It checks if the value of left operand is greater than the value of right operand; if yes, the condition becomes true. |
| < | It checks if the value of left operand is less than the value of the right operand; if yes, the condition becomes true. |
| >= | It checks if the value of the left operand is greater than or equal to the value of the right operand; if yes, the condition becomes true. |
| <= | It checks if the value of left operand is less than or equal to the value of right operand; if yes, the condition becomes true. |

*Table 7-1. Relational operators*

See the following code:

```
package main

import "fmt"

func main() {
      var result bool

      result = 1 == 2 // false
      fmt.Println(result)

      result = 1 > 2 // false
      fmt.Println(result)

      result = 1 != 2 // true
      fmt.Println(result)

      result = 1 < 2 // true
      fmt.Println(result)

      result = 1 <= 2 // true
      fmt.Println(result)
}
```

It will print:

```
false
false
true
true
true
```

## If statements

**If** statements allow you to execute statements conditionally. An **if** statement starts with the **if** keyword, followed by a condition, and then a block of code enclosed in curly brackets. If the condition evaluates to **true**, Go executes the statements inside the block. If the condition is **false**, Go evaluates subsequent **else if** statements if they exist. If none of the **if** or **else if** conditions evaluate to **true**, the **else** block is executed, provided it exists.

**If** statements must begin with an **if** block. They can have multiple **else if** blocks, but at most one **else** block.

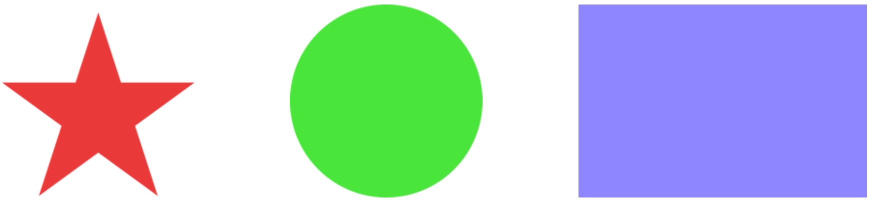
See following code:

```
package main


import "fmt"
import "math/rand"
import "time"


func main() {
    rand.Seed(time.Now().UnixNano())
    value := rand.Intn(200)
    fmt.Println(value)


    if value > 100 {
        fmt.Println("value is greater than 100")
    } else if value > 60 {
        fmt.Println("value is between 61 and 100")
    } else if value > 40 {
        fmt.Println("value is between 41 and 60")
    } else {
        fmt.Println("value if less than or equal to 41")
    }
}
```

It will print:

```
81
value is between 61 and 100
```

Run the same application many times and see that it will give different outputs.



For the shapes described above, let's see how the following **if** condition works (assuming that we store the shape type and color in the corresponding variables):

```
package main
import "fmt"

func main() {
    var shape string
    var color string

    if shape == "star" {
        fmt.Println("The shape is a star")
    }else if color == "red" {
        fmt.Println("The shape is red")
    }else if color == "green" {
        fmt.Println("The shape is green")
    }else {
        fmt.Println("It doesn't match any condition")
    }
    fmt.Println("Done")
}
```

Executes from top to bottom

For the shape ★, it does the following:

1. Checks if the shape is a star, and it evaluates to **true**
2. Executes the codes under the if block and prints "*The shape is a star*"
3. If the statement is completed, it continues with the following statements in the code and prints "*Done*"

**Note:**

Even if the shape's color is red, it does not print "*The shape is red*" because the first if block is executed.

For the shape 🟢, it does the following:

1. Checks if the shape is a star and it evaluates to **false**
2. Checks of the color is red and it evaluates to **false**
3. Checks of the color is green and it evaluates to **true**
4. Executes the codes under the if block and prints "*The shape is green*"
5. If the statement is completed, it continues with the following statements in the code and prints "*Done*"

For the shape 🟦, it does the following:

1. Checks if the shape is a star and it evaluates to **false**
2. Checks of the color is red and it evaluates to **false**
3. Checks of the color is green and it evaluates to **false**
4. None of the conditions evaluates to true, so it executes the else statement and prints "*It doesn't match any condition*"
5. If the statement is completed, it continues with the following statements in the code and prints "*Done*"

> **Note:**
>
> Notice that only one block within the if statement executes based on the value.

Let's revisit the first case and write the **if** statement with the **else** block.

See the following code:

```
package main

import "fmt"
import "time"
import "math/rand"

func main() {
      rand.Seed(time.Now().UnixNano())
      age := rand.Intn(100)

      fmt.Println(age)

      if age >= 18 {
            fmt.Println("you can vote")
      } else {
            fmt.Println("you cannot vote")
      }
}
```

It will print:

```
57
you can vote
```

**Note:**

There is also the **switch** statement for executing statements conditionally, but it is beyond the scope of this book.

## Logical Operators

Sometimes, you may need to check multiple conditions. For example, to execute certain statements only if all conditions are true, you can combine them with the logical AND operator **&&**. If you want to execute the statements if at least one condition is true, combine them with the logical OR operator **||**. See the table below for all logical operators:

| Operator | Syntax | Description |
|---|---|---|
| && (Logical AND) | exp1 && exp2 | returns true if both expressions exp1 and exp2 are true |
| \| \| (Logical OR) | exp1 \| \| exp2 | returns true if any one of the expressions is true` |
| ! (Logical NOT) | !exp | returns true if exp is false and returns false if exp is true. |

*Table 7-2. Logical operators*

So we can combine conditions as follows:

```
if shape == "star" && color == "red" {
      // will only be executed if two conditions are true
      fmt.Println("It is a red star")
}else if shape == "rectangle" || color == "blue"{
      // will only be executed if either condition is true and
      // if it is not a red star
      fmt.Println("It is either a rectangle or blue")
}
```

The **not (!)** operator inverts the boolean values. See the following example:

```
package main

import "fmt"

func main() {
    isAdmin := false

    if !isAdmin {
        fmt.Println("User is not an admin.")
    } else {
        fmt.Println("User is an admin.")
    }
}
```

It will print:

```
User is not an admin.
```

## Loops

You may need to execute the same statements multiple times. The most common scenario is to perform the same operation on all items in a slice or array. Consider the following example:

```
package main

import "fmt"

type Student struct {
      Name string
}
func main() {
      allStudents := getAllStudents()
      fmt.Println(allStudents[0].Name)
      fmt.Println(allStudents[1].Name)
      fmt.Println(allStudents[2].Name)
}
func getAllStudents() []Student {
      allStudents := make([]Student, 0)
      allStudents = append(allStudents, Student{Name: "Alice"})
      allStudents = append(allStudents, Student{Name: "Bob"})
      allStudents = append(allStudents, Student{Name: "Eve"})

      return allStudents
}
```

We want to print all students' names in the **allStudents** slice. If you have 200 students, writing the same line 200 times would be inefficient, and making changes would require editing all lines. The best approach is to repeat the statement for each item in the slice. Repeating the same block of code is called **iteration**, which can be achieved using **loops**.

If we examine the following statement, only the index changes within the loop.

```
fmt.Println(allStudents[0].Name)
fmt.Println(allStudents[1].Name)
fmt.Println(allStudents[2].Name)
```

To iterate through all values in the **allStudents** slice, you need an index starting at 0, incrementing by 1 each time until it reaches the length of the slice (which is 3). Programming languages provide **loops** that enable you to execute statements repeatedly as long as certain conditions are met.

### Learn how to iterate through lists, arrays, strings and maps

In Go, iterations can be achieved with the **for** loop. A **for** loop allows you to repeat statements in its body as long as its condition evaluates to true.

```
for condition {
        // loop body
}
```

**Note:**

If the condition of a **for** loop never evaluates to false, it will result in an infinite loop. This means the program will continuously execute the same statements until it is forcefully terminated. Unless intentionally desired, infinite loops should be avoided because they consume processing resources and prevent the program from proceeding to subsequent statements.

```
// condition will never be false, so it is an infinite loop
for true {
        fmt.Println("it going going to print this message
        forever")
}
```

First, we need to declare an integer variable to serve as the index, starting from 0. The loop's condition is that the index must be less than the length of the **allStudents** slice. Finally, we should increment the index by one after each iteration.

**Note:**

The **len** function in Go is a built-in function that returns the length of a provided parameter, depending on the parameter's type.

The signature of the **len** function is shown below:

```
func len(v Type) int
```

The **len** function takes a single required parameter, **v**, which can be a string, array, slice, or map.

Change the main function to:

```
func main() {
	allStudents := getAllStudents()
	fmt.Println(len(allStudents))                         │
	                                                      │   Executes from
	i := 0  → Variable to store current index             │   top to bottom
	for i < len(allStudents) {  → condition               │
		fmt.Println(allStudents[i].Name)              │
		i++  → Increment index by one                 ▼
	}
}
```

It will print:

```
3
Alice
Bob
Eve
```

Statements are still executed from top to bottom. When it encounters the loop, it does the following steps:

1. Loop starts
2. Check the condition if **i** (0) is less than **len(allStudents)** (3); it is true
3. Print to the name field of **student** at the index i(0) of **allStudents** slice
4. Increment i by one, now **i** is 1
5. Go back to the beginning of the loop

6. Check the condition if **i** (1) is less than **len(allStudents)** (3); it is true
7. Print to the name field of **student** at the index i(1) of **allStudents** slice
8. Increment **i** by one, now **i** is 2
9. Go back to the beginning of the loop
10. Check the condition if **i** (2) is less than **len(allStudents)** (3); it is true
11. Print to the name field of **student** at the index **i**(2) of **allStudents** slice
12. Increment **i** by one, **now i** is 3
13. Go back to the beginning of the loop
14. Check the condition if **i** (3) is less than **len(allStudents)** (3); it is false
15. Loop ends

**Note:**

**For** loops and **if** statements create their own block scope. **Moreover, if and for statements can only be written inside functions; for example, you cannot define an if block in the global scope.**

To iterate over **allStudents** slice, we declared a variable **i** to store the index value starting from 0, and we incremented it at the end of each iteration. A **for** statement can be simplified as follows:

```
for init statement; condition; post statement {
      // loop body
}
```

The **init statement** is executed once before the loop starts and can be used to initialize the index variable. The condition is evaluated before each iteration, and the loop continues as long as the condition evaluates to **true**. The **post statement** is executed at the end of each iteration and is typically used to increment or update the index variable.

```
i := 0                          → Init statement
for i < len(allStudents) {
      fmt.Println(allStudents[i].Name)
      i++                       → Post statement
}
```

So the above loop can be simplified as follows:

```
for i := 0; i < len(allStudents); i++ {
      fmt.Println(allStudents[i].Name)
}
```

Change the main function to:

```
func main() {
      allStudents := getAllStudents()
      fmt.Println(len(allStudents))
      for i := 0; i < len(allStudents); i++ {
            fmt.Println(allStudents[i].Name)
      }
}
```

It will print:

```
3
Alice
Bob
Eve
```

A **for** loop can be used to iterate through the characters of a string. See the following example:

```
package main

import "fmt"

func main() {
      str := "string"
      for i := 0; i < len(str); i++ {
            fmt.Println(str[i])
      }
}
```

It will print:

```
115
116
114
105
110
103
```

As you've seen in previous examples, when iterating through arrays or slices, we increase the index and access the value at a specific index, such as **allStudents[i]**. The **range** keyword in Go simplifies this process, as it returns both the current index and the value during each iteration.

```
func main() {
    allStudents := getAllStudents()
Index                            Value at the index
    for i, student := range allStudents {
        fmt.Println("current index", i)
        fmt.Println(student.Name)
    }
}
```

It will print:

```
current index 0
Alice
current index 1
Bob
current index 2
Eve
```

A **range** loop can also be used to iterate through maps. In this case, the first variable represents the key, and the second variable represents the corresponding value, not an index. See the following code:

```
package main
import "fmt"

func main() {
      m := map[string]int{
            "key1": 1,
            "key2": 2,
            "key3": 3,
      }
      for key, value := range m {
            fmt.Println(key, value)
      }
}
```

It will print:

```
key1 1
key2 2
key3 3
```

**Note:**

In other languages, you can find **while** and **do-while** loops, which are used to execute statements repeatedly. However, Go does not have these loops. Instead, Go uses the **for** loop to serve all these purposes.

## In A Nutshell

You should use **if** statements when you need to execute certain statements conditionally. Conversely, **for** loops are used to execute statements repeatedly as long as a specified condition holds true.

**Note:**

Be open-minded and think creatively. Remember that, as long as you follow the language rules, you can do almost anything. You can combine statements and create complex code blocks—for example, by writing an **if** statement inside a loop. See the following example:

```
package main

import "fmt"

func main() {
      scores := []float64{20.3, 44, 95, 55}
      numberOfSuccessfulStudents := 0
      for _, score := range scores {
            if score >= 45 {
                  numberOfSuccessfulStudents++
            }
      }
      fmt.Printf("Number of Successful Students: %d\n",
      numberOfSuccessfulStudents)
}
```

## Summary

- Some statements can be executed conditionally with the **if statements**. If statements start with a mandatory if block and can have many **else if** blocks to execute some statements if previous if and else if conditions did not run. If no if and else if conditions are not met, the else statement is executed. There must be at most one **else** statement.
- With **relational operators**, we can compare values. Relational operators evaluate to either true or false.
- Loops are used to repeat the same statements. **For** loop executes statements until its condition evaluates to false. If the condition of the loop is never evaluated as false, it causes an infinite loop
- **If statements** and **for loops** can only be written inside a function.
- **If statements** and **for loops** create their block scopes.
- **Range** keyword also iterates through **arrays**, **slices**, and **maps**. While the range keyword returns the index and the value at that index for **arrays** and **slices**, it returns the key and value for **maps**.

## Learn More

- Read about the **while** and **do-while** loops in the other languages.

# Chapter 8

## More to functions and methods

You will learn:

- Pointers

An application consists of many functions; the **main** function is where a Go application begins execution. From the **main** function, we can call other functions and execute statements within them.

The following code is a simple tax calculator. The **deductTax** function deducts 1000 from the **salaryInUSD**.

```
package main

import "fmt"

func main() {
      salaryInUSD := 5000.0
      deductTax(salaryInUSD)
      fmt.Printf("Your net salary is %f\n”, salaryInUSD)
}

func deductTax(salary float64) {
      salary = salary - 1000.0
}
```

We pass the **salaryInUSD** variable as an argument to the **deductTax** function, which modifies the value of the **salary** parameter. After **the deductTax** method is called, we print the value and expect to see the deducted amount. It should print:

```
Your net salary is 4000.000000
```

Let’s execute the function, and it is going to print:

```
Your net salary is 5000.000000
```

Even though we changed the **salary** parameter’s value in the **deductTax** method, the **salaryInUSD** variable in the main function did not change. Why?

**When you pass arguments to a function, the function creates new variables for every parameter under the hood and copies values from corresponding arguments**. Behind the scenes, Go uses the = operator to initialize the parameter:

```
salary = salaryInUSD // value is copied from the argument to the
parameter
```

Suppose you assign a new value to a parameter inside a function. In that case, it will only change the local copy of the parameter, not the original argument's value. Inside the function, you lose access to the original argument, and any modifications do not affect it.

First, let's demonstrate that a function creates new variables for its parameters. Previously, we discussed how creating a variable allocates memory. If a function creates new variables for its parameters, then the **salaryInUSD** variable in the **main** function and the **salary** parameter in the **deductTax** method should have different memory addresses.

How can we view the memory addresses of the variables to verify that they are different? In Go, we use the **&** operator followed by the variable's name to obtain its memory address. Let's print the addresses of the variables to see the difference.

```
package main

import "fmt"

func main() {
        salaryInUSD := 5000.0
        fmt.Printf("Address in main function = %v\n", & salaryInUSD)
        deductTax(salaryInUSD)
        fmt.Printf("Your net salary is %f\n", salaryInUSD)
}

func deductTax(salary float64) {
        fmt.Printf("Address in deductTax function = %v\n", & salary)
        salary = salary - 1000.0
}
```

It will print:

```
Address in main function = 0xc0000a0008
Address in deductTax function = 0xc0000a0010
Your net salary is 5000.000000
```

The address of the **salaryInUSD** variable in the **main** function is **0xc0000a0008**, while the address of the **salary** variable in the **deductTax** function is **0xc0000a0010**. This shows that they occupy different memory locations. Therefore, when we change the **salary** variable inside the **deductTax** function, it does not affect the **salaryInUSD** variable in the **main** function.

How can we fix this issue? There are two approaches. First, we can modify the **deductTax** function to return the new salary value, and then assign this returned value to the **salaryInUSD** variable in the **main** function.

```
package main

import "fmt"

func main() {
      salaryInUSD := 5000.0
      salaryInUSD = deductTax(salaryInUSD)
      fmt.Printf("Your net salary is %f\n", salaryInUSD)
}
func deductTax(salary float64) float64 {
      return salary - 1000.0
}
```

It will print:

```
Your net salary is 4000.000000
```

## Pointers

Another approach is to obtain the memory address of the **salaryInUSD** variable in the **main** function and modify the value at that address. This means passing a reference to the variable's memory location rather than passing the value itself. In Go, this is achieved using pointers.

**Pointers are variables that store the address of another variable**. **Pointer types** are represented with ***** followed by another type.

```
var intPointer *int
```

The **intPointer** variable only stores an address of an **int** type variable. You can assign the address of the variable with the **&** operator.

```
intVariable := 40
intPointer = &intVariable
```

**Note:**

A pointer is 4 bytes in size on a 32-bit architecture processor and 8 bytes on a 64-bit architecture processor. **For simplicity, it will be displayed as one block in the following examples.**

A memory address is an integer displayed as a hexadecimal (base 16) representation starting with **0x.** For example, address **0xc00011a018** shows that the memory location is c00011a018 in hexadecimal and 211106232532992 in decimal.

$$c00011a018_{16}=211106232532992_{10}$$

When a pointer is created, all bits are set to zeros, just like the other types we have learned. Therefore, its initial value is 0x0 in hexadecimal. Zero value **0x0** is called **nil**, showing that no value has yet been assigned to the pointer.

Suppose you need to **access or modify** the value of **intVariable** from the pointer **intPointer**. In that case, you must use the *** (dereference)** operator on the pointer.

```
fmt.Println(*intPointer) // it will print 40
*intPointer = 50 // changes the value in the address to 50
```

See the following code:

```
package main
import "fmt"

func main() {
      intVariable := 40

      var intPointer *int // it is nil
      fmt.Println(intPointer)
      // now it stores the address of the variable
      intPointer = &intVariable

      fmt.Printf("Address of intPointer = %v\n", &intPointer)
      fmt.Printf("Address of intVariable = %v\n", &intVariable)
      fmt.Printf("Value of intPointer = %v\n", intPointer)

      // getting the value in the pointer
      var intValue int = *intPointer
      fmt.Printf("Value of stored in the pointer = %v\n",
      intValue)

      // to change the value
      *intPointer = 50
      fmt.Printf("Value of stored in the pointer = %v\n",
      *intPointer)
}
```

It will print:

```
<nil>
Address of intPointer = 0xc00011a018
Address of intVariable = 0xc000122008
Value of intPointer = 0xc000122008
Value of stored in the pointer = 40
Value of stored in the pointer = 50
```

Let’s dissect the code.

We created an **int** variable to store the value 40. Go allocated memory for this variable and assigned it the value.

```
intVariable := 40
```

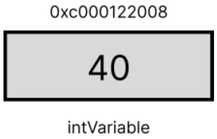


*Figure 8-1. Memory address storing value of a variable*

Then, we created a variable of pointer type to store the address of another variable of the same type. However, we have not assigned any value yet. The variable cannot point to another variable until we set the address of another variable. Therefore, its zero value is nil.

```
var intPointer *int
```

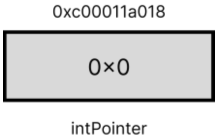


*Figure 8-2. Pointer variable with its zero value*

Then, we assign the address of **intVariable** to **intPointer** as the value.

```
intPointer = &intVariable
```

Now, the pointer type stores the address of the **intVariable**.

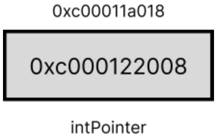


*Figure 8-3. Pointer variable stores the address of another variable*

To access or modify the value stored at the address contained in the pointer (e.g., at **0xc000122008**), we should dereference the pointer using the ***** operator before the pointer variable.

```
var intValue int = *intPointer
fmt.Printf("Value of stored in the pointer = %v\n", intValue)
```

It will print:

```
Value of stored in the pointer = 40
```

If we need to change the value at the address the pointer is referencing, we can assign a new value by dereferencing the pointer using the ***** operator:

```
*intPointer = 50
fmt.Printf("Value of stored in the pointer = %v\n", *intPointer)
```

It will print:

```
Value of stored in the pointer = 50
```

Let's revisit the **deductTax** method and modify it to use a pointer, so we can pass the address of the variable instead of passing it by value.

```
package main


import "fmt"


func main() {
      salaryInUSD := 5000.0
      deductTax(& salaryInUSD)
      fmt.Printf("Your net salary is %f\n", salaryInUSD)
}
func deductTax(salaryInUSD *float64) {
      *salaryInUSD = *salaryInUSD - 1000.0
}
```

It will print:

```
Your net salary is 4000.000000
```

Now, it works as it is supposed to.

**Note:**

Pointers can also point to a struct. If you want to modify the fields of a struct pointer, you can do so directly without explicitly dereferencing the pointer. See the following example:

```
package main
type Student struct {
      Name string
}
func main() {
      alice := &Student{}
      alice.Name = "Alice"
}
```

You could do `*alice.Name` if the Name field were a pointer. Otherwise, it will cause a compilation error.

## Pointers as methods' receivers

Keep also in mind that methods' receivers also pass by value. Let's refactor the code and make the **deductTax** a method of a struct holding the salary.

```
package main

import "fmt"

func main() {
	payer := TaxPayer{
		SalaryInUSD: 5000.0,
	}
	fmt.Printf("Address in struct is %v\n", &payer.SalaryInUSD)
	payer.DeductTax()
	fmt.Printf("Your net salary is %f\n", payer.SalaryInUSD)
}

type TaxPayer struct {
	SalaryInUSD float64
}

func (payer TaxPayer) DeductTax() {
	fmt.Printf("Address in method is %v\n", &payer.SalaryInUSD)
	payer.SalaryInUSD = payer.SalaryInUSD - 1000
}
```

It will print:

```
Address in struct is 0xc0000b6008
Address in method is 0xc0000b6010
Your net salary is 5000.000000
```

**In Go, receivers are also treated as new variables.** When a method is called on a variable, the variable's value is copied to the receiver. This is why the addresses of the struct and the receiver are different in the example above, and why changes to the receiver's fields do not affect the original struct. To fix this, you can make the receiver a pointer type**. If the receiver is a pointer, the method will copy the address of the**

**variable to the receiver, allowing modifications to affect the original struct.**

```
func (payer *TaxPayer) DeductTax() {
      fmt.Printf("Address in method is %v\n", &payer.SalaryInUSD)
      payer.SalaryInUSD = payer.SalaryInUSD - 1000
}
```

When we execute the code again, it will print:

```
Address in struct is 0xc000096008
Address in method is 0xc000096008
Your net salary is 4000.000000
```

**Note:**

You must use a pointer receiver whenever you need to modify the a field’s value of a receiver.

## Learn if types pass by value or reference

If you pass a variable of primitive types to the function, the value is copied to the parameter. However, **types like slices and maps are pointers behind the scenes**. If you pass a slice or map to a function, they pass by reference. If you modify the value in a specific index or key, the corresponding value in the argument will also change.

**Note:**

In Go, all types having zero value as **nil** are pointers under the hood, and when they are passed to a function, their addresses are copied.

```
package main

import "fmt"

func main(){
	salaries := map[string]float64{
		"Alice": 85,
		"Bob": 70,
	}
	fmt.Printf("Salary of Alice before the raise is: %g\n", salaries["Alice"])
	giveRaise(salaries)
	fmt.Printf("Salary of Alice after the raise is: %g\n", salaries["Alice"])
}

func giveRaise(salaries map[string]float64) {
	for key := range salaries {
		// increase the salaries by 10 percent
		salaries[key] =salaries[key] *1.1
	}
}
```

It will print:

```
Salary of Alice before the raise is: 85
Salary of Alice after the raise is: 93.50000000000001
```

## Reading Input From User

A good use case for understanding pointers is getting user input from the **stdin**. Let's say we need to get a string input from the user and store it in a variable. The **fmt.Scan** function reads the value from the **stdin** and saves the value to the address in the argument. See the following code:

```
package main

import "fmt"

func main(){
      fmt.Println("Enter the user name:")
      var userName string
      fmt.Scan(&userName)
      fmt.Printf("User Name is: %s\n", userName)
}
```

It will ask for the input after printing the prompt:

```
Enter the user name:
Alice
User Name is: Alice
```

**Note:**

You can also scan multiple values with a format. See the example:

```
var userName string
var userAge int
fmt.Scanf("%s %d",&userName, &userAge)
```

## Summary

- When a function is called, new variables are created for every parameter, and the values are copied from the arguments. Changing the parameter value inside the function body only changes the value in the new variable, not the argument value.
- A variable's address can be accessed with the **&** operator.
- To modify argument's value inside a function body, the address of the argument should be passed to the parameter. Addresses can only be stored in pointer types that are preceded by the ***type** syntax such as *string and *int.
- The zero value of a pointer type is **nil (0x0)** because it cannot point to any addresses until you assign it.
- To modify or get the value in a pointer variable, you must deference by using the ***** operator.
- To change the receiver's field in a method body, the receiver must be a pointer type.

## Learn more

- Create a function, pass variables of all types you learned to the function, and change the parameter value inside the function body. Check if the argument's value changed. Try to observe if the types pass by value or reference.

# Chapter 9

## Abstraction

You will learn:

- Interfaces

Abstraction is a fundamental concept in programming languages. It is basically about hiding the implementation irrelevant to your code logic. Take the following as an example:

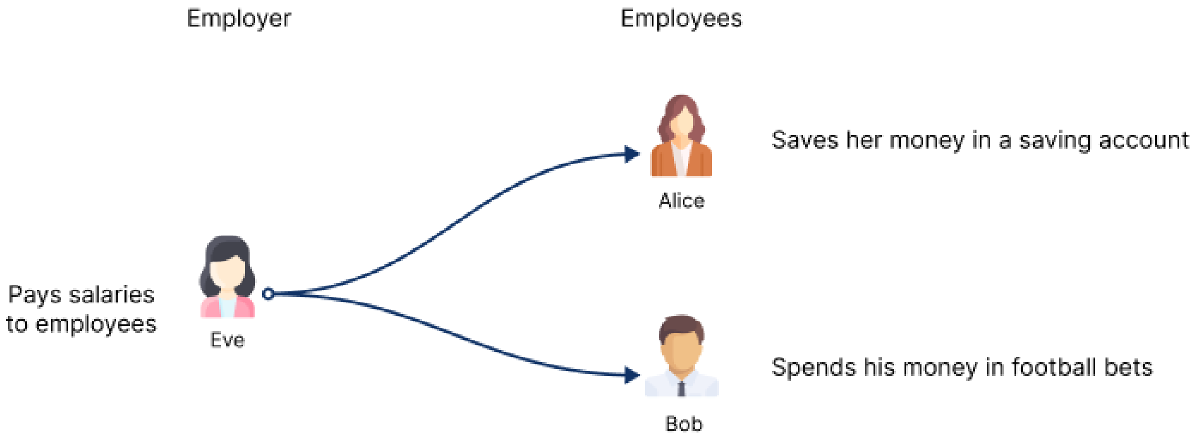

Employer
Employees
Saves her money in a saving account
Alice
Pays salaries
to employees
Eve
Spends his money in football bets
Bob

*Figure 9-1. Employer and employee relationship*

Employer Eve pays her employees, Alice and Bob, monthly. Alice is a thrifty employee who saves most of her salary in a savings account. In contrast, Bob is extravagant and spends his money on football bets. As an employer, Eve doesn't concern herself with how her employees choose to spend their salaries. From Eve's perspective, everyone she pays is simply an **Employee**. Her only responsibility is to provide the compensation. In this context, Eve is **abstracted** from the employees' behaviors and there are two types of **Employee**: **Thrifty Employee** and **Extravagant Employee**. These types are called **concrete types**.

**Learn how to achieve abstraction**

## Interfaces

In Go, abstraction is achieved through **interface** types. Interfaces specify a list of **method signatures** that concrete types must implement. Any struct type that has all the methods defined in an interface can be used as a value of that interface type.

**Type syntax:** In Go, the interface type starts with the interface keyword, followed by opening and closing brackets. Inside the brackets, you should write functions signatures line by line.

```
var employeeInterface interface {
    // function signatures, it can have many signatures
    PaySalary(salary float32)
}
```

**Note:**

Generally interface types are declared in the global scopes for reusability.

**Zero value:** Zero value of a variable of interface type is **nil**.

**Interface Literal:** Any value of a struct type having all the methods defined in the interface type can be used as an interface value.

Try the following code:

```
package main

import "fmt"

type Employee interface {
    PaySalary(salary float32)
}
                           Interface type         Concrete struct type
func main() {
    var employee Employee = ThriftyEmployee{
        Name: "Alice",
    }
    employee.PaySalary(10000.0)
}

type ThriftyEmployee struct {
    Name string
}

func (t ThriftyEmployee) PaySalary(salary float32) {
    fmt.Println("put money in saving account")
}
```

It will print:

```
put money in saving account
```

Add the following code to the main function to get the Name field from the underlying **ThriftyEmployee** struct type.

```
fmt.Println(employee.Name)
```

You will get the following error:

```
employee.Name undefined (type Employee has no field or method
Name)
```

**You get the above error because the interface knows nothing about the underlying type. It just knows that the underlying type has all the methods in the interface type declaration, and you can only call those methods from the variable.**

Try the following code:

```
package main

import "fmt"

type Employee interface {
    PaySalary(salary float32)
}

func main() {
    var zeroValueInterface Employee
    fmt.Println(zeroValueInterface)

    employees := make([]Employee, 0)
    thriftyEmployee := ThriftyEmployee{
        Name: "Alice",
    }

    extravagantEmployee := ExtravagantEmployee{
        Name: "Bob",
    }
    employees = append(employees, thriftyEmployee)
    employees = append(employees, extravagantEmployee)
    for _, employee := range employees {
        employee.PaySalary(10000.0)
    }
}

type ThriftyEmployee struct {
    Name string
}

func (t ThriftyEmployee) PaySalary(salary float32) {
    fmt.Println("put money in saving account")
}

type ExtravagantEmployee struct {
    Name string
}

func (t ExtravagantEmployee) PaySalary(salary float32) {
    fmt.Println("betting on football matches")
}
```

It will print:

```
<nil>
put money in saving account
betting on football matches
```

Interfaces are primarily used in well-structured, professional applications. You will encounter and utilize interfaces more frequently when engaging in software design and architecture. Therefore, interfaces might be considered an advanced topic in some languages, but in Go, they are an essential part of the language. The Go standard library includes many predefined interfaces, such as **io.Reader** and **io.Writer**, which are defined as follows in the **io** package:

```
type Reader interface {
      Read(p []byte) (int, error)
}

type Writer interface {
      Write(p []byte) (int,error)
}
```

You will primarily use these interfaces when reading from or writing to files.

**The next chapter will cover a built-in interface that you use most frequently in Go.**

## Summary

- **Abstraction** is the hiding of implementation irrelevant to the business logic.
- In Go, abstraction can be achieved with **interfaces**.
- The **interface** is a type consisting of function signatures.
- A struct with all the function signatures in the interface as methods can be used as interface's value.

## Learn more

- Write **Notifier** interface, which has the following signature.

  ```
  Notify(message string)
  ```

  Create two concrete types: **EmailNotifier** and **SMSNotifier**.

  - The **EmailNotifier** type should have a string field **email** and implement a method that prints: "Sending the message as email" along with the email address.

  - The **SMSNotifier** type should have a string field **phoneNumber** and implement a method that prints: "Sending the message as SMS" along with the phone number.

  Next, create a variable of type **Notifier** (an interface). Assign an **EmailNotifier** instance to this variable and call its **Notify** method with a message. Then, assign an **SMSNotifier** instance to the same variable and call **Notify** again. Observe that it prints different outputs, and there are no compilation errors.

# Chapter 10

## Handling errors

You will learn:

- Errors

When codes are executed, we may encounter cases that can cause unexpected situations. Let’s inspect the following case:

```
package main

import "fmt"

func main() {
      myBankAccount := BankAccount{
            BalanceInUSD: 1000.0,
      }
      myBankAccount.Withdraw(2000.0)
}

type BankAccount struct {
      BalanceInUSD float64
}

func (b *BankAccount) Withdraw(amountInUSD float64) {
      fmt.Printf("Withdrawing %f USD from the balance %f\n",
      amountInUSD,b.BalanceInUSD)
      b.BalanceInUSD = b.BalanceInUSD - amountInUSD
}
```

It will print:

```
Withdrawing 2000.000000 USD from the balance 1000.000000
```

In the above example, we attempt to withdraw 2000 USD from an account that only has 1000 USD. We should prevent account holders from withdrawing amounts exceeding their balance. To do this, we need to intervene in the normal flow of the program and prevent the deduction if the balance is insufficient. This kind of intervention can be handled using **errors** in programming languages.

## Learn error handling

## Errors

Error handling in Go is different from that in many other languages. While other languages use exceptions and **try/catch** blocks, Go has a built-in **error** interface to manage errors. The **error** interface is defined as follows:

```
type error interface {
      Error() string
}
```

New errors can be created with the **errors.New(message string)** function. It gets an error message as a string parameter.

The **New** function returns a pointer of **fundamental** struct in the error package.

```
func New(message string) error {
      return &fundamental{
            msg:   message,
            stack: callers(),
      }
}
```

A fundamental pointer has all the methods in the **error** interface. Hence, it can be used as an error.

```
type fundamental struct {
      msg string
      *stack
}

func (f *fundamental) Error() string {
      return f.msg
}
```

Let's modify the **Withdraw** method to return an error if the withdrawal amount exceeds the current balance. To create an error, we will use the **errors.New** function.

```
package main

import (
    "errors"
    "fmt"
)


func main() {
    myBankAccount := BankAccount{
        BalanceInUSD: 1000.0,
    }
    err := myBankAccount.Withdraw(2000.0)
    if err != nil {
        fmt.Println("could not withdraw money")
        return
    }
}

type BankAccount struct {
    BalanceInUSD float64
}
func (b *BankAccount) Withdraw(amountInUSD float64) error {
    fmt.Printf("Withdrawing %f USD from the balance %f\n",
    amountInUSD,b.BalanceInUSD)
    if b.BalanceInUSD < amountInUSD {
        return errors.New("withdrawal amount is more than
        balance")
    }
    b.BalanceInUSD = b.BalanceInUSD - amountInUSD

    return nil
}
```

It will print:

```
Withdrawing 2000.000000 USD from the balance 1000.000000
could not withdraw money
```

Let’s dissect the code:

```
func (b *BankAccount) Withdraw(amountInUSD float64) error {
	fmt.Printf("Withdrawing %f USD from the balance %f\n",
	amountInUSD,b.BalanceInUSD)
	if b.BalanceInUSD < amountInUSD {
		return errors.New("withdrawal amount is more than
		balance")
	}
	b.BalanceInUSD = b.BalanceInUSD - amountInUSD

	return nil
}
```

We changed the **Withdraw** method to return an error. We checked if the balance was less than the withdrawal amount and returned an error created by **errors.New** function. The **errors.New** function expects a string to describe the error.

After calling the **Withdraw** method in the **main** function, we check if the error is not **nil.** You should always check the error returned from a function. You should not proceed if there is an error.

```
err := myBankAccount.Withdraw(2000.0)
if err != nil {
	fmt.Println("could not withdraw money")
	return // do not proceed
}
```

When a function returns an error, the specific type of error is unknown because **error** is an interface. You may want to execute different statements based on the error type. To check whether an error matches a specific error, you should use the **errors.Is** function. Keep in mind that **errors.Is** compares the underlying addresses of the errors; if they are the same, it returns **true**, otherwise it returns **false**.

Let's add another case where an account can be blocked. In this scenario, the user cannot withdraw money from a blocked account, even if the balance is sufficient.

```
package main

import (
      "errors"
      "fmt"
)

var (
      ErrAccountIsBlocked = errors.New("account is blocked")
      ErrInsufficientBalance = errors.New("amount is more than
      balance")
)
func main() {
      myBankAccount := &BankAccount{
            BalanceInUSD: 1000.0,
            IsBlocked: true,
      }
      err := myBankAccount.Withdraw(2000.0)
      if errors.Is(err, ErrAccountIsBlocked) {
            // Do operations when account is blocked
            fmt.Println("Your account is blocked")
            return // do not proceed
      } else if errors.Is(err, ErrInsufficientBalance) {
            // Do operations amount is not enough
            fmt.Println("Try lower amount")
            return // do not proceed
      }
      // Do operation when there is no error
      fmt.Println("Withdrawal is successful")
}

type BankAccount struct {
      BalanceInUSD float64
      IsBlocked bool
}
```

```
func (b *BankAccount) Withdraw(amountInUSD float64) error {
      if b.IsBlocked {
            return ErrAccountIsBlocked
      }else if b.BalanceInUSD < amountInUSD {
            return ErrInsufficientBalance
      }
      b.BalanceInUSD = b.BalanceInUSD - amountInUSD

      return nil
}
```

It will print:

```
Your account is blocked
```

Let's dissect the code.

Firstly, we created variables to store errors in the global scope. We have to do that because **errors.Is** function compares the addresses of the errors.

```
var (
      ErrAccountIsBlocked = errors.New("account is blocked")
      ErrExceedingAmount = errors.New("amount is more than
      balance")
)
```

Then, we added a bool field to store whether an account is blocked.

```
type BankAccount struct {
      BalanceInUSD float64
      IsBlocked bool
}
```

In the main function, we set the **IsBlocked** field to **true.**

```
myBankAccount := &BankAccount{
      BalanceInUSD: 1000.0,
      IsBlocked: true,
}
```

In the **Withdraw** method, we first check if the account is blocked, and return the **ErrAccountIsBlocked** if the **IsBlocked** is true.

```
func (b *BankAccount) Withdraw(amountInUSD float64) error {
	if b.IsBlocked {
		return ErrAccountIsBlocked
	}else if b.BalanceInUSD < amountInUSD {
		return ErrExceedingAmount
	}

	b.BalanceInUSD = b.BalanceInUSD - amountInUSD

	return nil
}
```

After calling the Withdraw method in the main function, we use the **errors.Is** function to check whether err is a specific error. Since the returned error is **ErrAccountIsBlocked**, **errors.Is(err, ErrAccountIsBlocked)** evaluates to **true**, and it prints **Your account is blocked**.

```
err := myBankAccount.Withdraw(2000.0)

if errors.Is(err, ErrAccountIsBlocked) {
	// Do operations when account is blocked
	fmt.Println("Your account is blocked")
	return // do not proceed
} else if errors.Is(err, ErrExceedingAmount) {
	// Do operations amount is not enough
	fmt.Println("Try lower amount")
	return // do not proceed
}
// Do operation when there is no error
fmt.Println("Withdrawal is successful")
```

**Note:**

The **Error** method on the error interface returns the error message. It can be used to log the error or to get the message.

```
fmt.Println(err.Error())
```

## Summary

- Errors in Go are values that indicate a problem, allowing the program to handle them explicitly instead of halting execution unexpectedly.
- Any concrete type that implements the error interface can be used as an error.

```
type error interface {
      Error() string
}
```

- New errors can be created with the **errors.New** function.
- To check if an error is a specific error, the **errors.Is** function is used. Error compared with the **errors.Is** function must refer to the same error variable.

## Learn more

- Create a struct that implements the error interface and returns it as an error from a function.
- Create a divide function that gets two arguments, dividend and divisor of integer types, and returns an error when the divisor is zero.
- Learn how to check whether a file or directory exists in the file system.

# Chapter 11

## Testing

You will learn:

- Tests

Software is developed collaboratively, and your teammates may modify the code you write. In fact, you will often revisit and change your own previous code in assignments or projects. To ensure that new changes do not break the existing business logic you've already validated and completed, what strategies can be used?

Testing ensures that your code functions as intended. While it may seem like an advanced topic in an introductory book, understanding the concept of testing and learning how to write simple tests can be invaluable for your projects. Go is an ideal language for testing because, unlike many other languages, it includes a built-in testing framework. As a result, you don't need to install or rely on any third-party libraries.

We have developed the following code throughout the last chapter.

```
package main
import "errors"


var (
	ErrAccountIsBlocked = errors.New("account is blocked")
	ErrExceedingAmount = errors.New("amount is more than
	balance")
)
type BankAccount struct {
	BalanceInUSD float64
	IsBlocked bool
}


func (b *BankAccount) Withdraw(amountInUSD float64) error {
	if b.IsBlocked {
		return ErrAccountIsBlocked
	}
	if b.BalanceInUSD < amountInUSD {
		return ErrExceedingAmount
	}
	b.BalanceInUSD = b.BalanceInUSD - amountInUSD


	return nil
}
```

Let's test the **Withdraw** methods of the **BankAccount** struct. There are the following cases to be tested:

- When the account balance is greater than or equal to the withdrawal amount, the method should proceed without returning an error, and the amount should be deducted from the balance
- When the account balance is less than the withdrawal amount, the method should return the **ErrExceedingAmount** error.
- When the account is blocked, the method should return the **ErrAccountIsBlocked** error, even if the account balance is sufficient for the withdrawal.

In Go, test code is stored in separate files, and these test files must have filenames ending with the **_test.go** suffix. By convention, a test file is created in the same directory as the source code it tests. To test the above logic, you should create a file named **main_test.go** in the same directory.

We will write our tests in the **main_test.go** file. In Go, every test function must start with the prefix **Test** and accept a single parameter of type ***testing.T**. Let's create our first test for the initial case.

```
package main

import "testing"

func TestBankAccount_Withdraw_With_Enough_Balance(t *testing.T) {
}
```

We chose a descriptive name for the test function, incorporating the struct and method names. Next, we need to create an account and call the **Withdraw** method with an amount less than the account's balance within the test function.

```
package main

import "testing"

func TestBankAccount_Withdraw_With_Enough_Balance(t *testing.T) {
      account := BankAccount{
            BalanceInUSD: 1000,
      }
      err := account.Withdraw(200)
}
```

We created an account with a balance of 1000 USD and attempted to withdraw 200 USD. We expect the error to be **nil** and the **BalanceInUSD** to be 800. If these values are not as expected, the test should fail. You can fail the test by calling the **Fail** method on the **t** parameter. Specifically, the test should fail if the returned error is not **nil** or if the balance is not 800 after the withdrawal.

```
package main

import "testing"

func TestBankAccount_Withdraw_With_Enough_Balance(t *testing.T) {
      account := &BankAccount{
            BalanceInUSD: 1000,
      }
      err := account.Withdraw(200)
      if err != nil {
            t.Fail()
      }
      if account.BalanceInUSD != 800 {
            t.Fail()
      }
}
```

We have written our first test. How can we execute the tests of our project? You should run all tests in your projects by executing **go test ./...** command. It will print:

```
ok argument 0.015s
```

If you want more detailed information about the tests, add **-v** argument (v stands for verbose). Let's execute **go test -v ./...** command.

```
=== RUN TestBankAccount_Withdraw_With_Enough_Balance
--- PASS: TestBankAccount_Withdraw_With_Enough_Balance (0.00s)
PASS
ok argument 0.009s
```

Suppose your teammate modifies the **Withdraw** method to deduct a 1 USD commission from every withdrawal.

```
b.BalanceInUSD = b.BalanceInUSD - amountInUSD – 1
```

If we execute the tests again, it will print:

```
=== RUN TestBankAccount_Withdraw_With_Enough_Balance
--- FAIL: TestBankAccount_Withdraw_With_Enough_Balance (0.00s)
FAIL
FAIL argument 0.011s
FAIL
```

Now, the test indicates a failure, but it should provide more information about why it failed. To include details about the failure, you can use the **t.Error** or **t.Errorf** methods, which accept strings and formatters as parameters, respectively. Let's replace **t.Fail** with **t.Errorf** and run the tests again.

```
if account.BalanceInUSD != 800 {
      t.Errorf("expected %f but got %f", 800.0,
      account.BalanceInUSD)
}
```

It will print:

```
=== RUN TestBankAccount_Withdraw_With_Enough_Balance
main_test.go:17: expected 800.000000 but got 790.000000
--- FAIL: TestBankAccount_Withdraw_With_Enough_Balance (0.00s)
FAIL
FAIL argument 0.010s
FAIL
```

Now, it also shows why it failed.

Writing tests in Go is quite straightforward. You can automatically verify your logic with tests instead of manually checking the results.

## Summary

- **Testing** is an automated way to check if code does what it should do.
- Test files are created in the same directory with the filename followed by **_test.go** suffix.
- Test files contain functions with func TestXxxxxx(t *testing.T) signature.
- Inside the test functions, actual methods or functions are called, and if they do not return the expected value, tests are failed by calling **t.Fail** or **t.Errorf** methods.
- All tests in the project can be executed with the **go test -v ./...** command to see if there are any failing tests.

## Learn More

- Write IsPrimeNumber function in a go file and create a test file and test function to test the function. Call the IsPrimeNumber function inside the test. Fail the test if it does not return true for the values 3,5,11,17,83, and 97. The test should also fail if the IsPrimeNumber function does not return false for the values 2, 48,49,103.
- Read about software testing.
- Learn about different types of tests in software testing.

# Chapter 12

## Way to Go

You will learn:

- A Tour of Go
- Calculator Project
- Debugging
- Git and Github
- Practice

## A tour of Go

The Go team created the A tour of Go (https://go.dev/tour/welcome/1) website to help you learn more about Go and its internals. The site gives you detailed information about Go internals and allows you to run your codes in the browser. Try to complete the tour and practice what you learn.

## Calculator project

Build a simple calculator that can get all the data as arguments, compile it, and name it as **calculator**. It should do multiplication, sum, division, and subtraction. It should give the following outputs:

```
./calculator multiply 2 4

8

./calculator sum 2 3

5

./calculator sub 2 3

-1

./calculator div 12 8

1.5
```

## Debugging

Debugging is the process of determining problems in your code when it does not work as expected. Integrated Development Environments (IDEs) have many tools for debugging. IDEs can stop code execution with breakpoints so that you can inspect the values in the variables and what functions return.

To start learning debugging, you should select an IDE first. Here are some tools you can use to code and debug:

- JetBrains IDEs
- Visual Studio Code

## Learn Git and Github

Git is a widely used version control application nowadays, regardless of the languages you use. It allows you to track the changes in your code. It is also an excellent tool for collaborating with many other developers and version control.

GitHub is a platform where you can store your source codes in repositories using Git and collaborate with millions of other developers. You can also see other repositories and read codes written by others.

## Practice

Create sample applications and store them in Github. You can find countless project ideas in your daily life. Keep in mind that practice makes it perfect.

Here are some project ideas:

- You can create an application for restaurants to track orders.
- You can create an application for hospitals to manage doctors' and nurses' availability.
- You can create an application to schedule bus rides in a municipality.

# Appendixes

## Formating verbs

| Verb | Explanation |
|---|---|
| %t | booleans |
| %d | int, int8, etc. |
| %g | float32, complex64, etc. |
| %s | string |
| %b | base 2 |
| %v | the value in a default format |
| %c | the character represented by the corresponding Unicode code point |
| %o | base 8 |
| %x | base 16 |

## Naming conventions

The space character is a reserved language key checked during compilation in programming languages. A name with many words should be combined by removing spaces. There are many alternative ways to do that. Let’s see the alternatives we can use for **my favourite variable**:

| Case | Example |
|---|---|
| Camel Case | myFavouriteVariable |
| Kebab Case | my-favourite-variable |
| Snake Case | my_favourite_variable |
| Pascal Case | MyFavouriteVariable |

## Essential Language Learning Checklist

- Learn whether your language is compiled, interpreted, or a hybrid language
- Learn how to cross compile
- Learn how to set exit code
- Learn how to access arguments
- Learn how to access to the three streams
- Learn how to read input from stdin
- Learn how to write to stdout and stderr
- Learn how to declare variables
- Learn whether the language is a statically typed language or dynamically typed language
- Learn how to create enumeration
- Learn all the types with their zero values and their literals
- Learn how to create, add to, or remove from lists
- Learn whether your language is functional and has first-class functions
- Learn the scopes in the language
- Learn how to do garbage collection
- Learn how to write methods
- Learn how to create packages
- Learn how to use resources from other files and packages
- Learn how to encapsulate
- Learn how to iterate through lists, arrays, strings and maps
- Learn if types pass by value or reference
- Learn how to achieve abstraction
- Learn error handling

## References

Fowler, Martin. (2014). *Shu Ha Ri*. Retrieved from https://martinfowler.com/bliki/ShuHaRi.html