

{KODE {KLOUD

about Golang

Golang

- created by the engineers at Google.
- Go was created to combine -
 - the ease of programming of an interpreted, dynamically typed language (such as Python)
 - with the efficiency and safety of a statically typed, compiled language. (such as C++)
 - It also aimed to be modern, with support for networked and multicore computing.

installing Golang

installing Go

- <https://go.dev/doc/install>

Download and install - The Go x +

go.dev/doc/install

GO

Why Go Get Started Docs Packages

Download and install

Download and install Go quickly with the steps described here.

For other content on installing, you might be interested in:

- [Managing Go installations](#) -- How to install multiple versions and uninstall.
- [Installing Go from source](#) -- How to check out the sources, build them on your own machine, and run them.

1. Go download.

Click the button below to download the Go installer.

Download Go for Mac
go1.17.5.darwin-amd64.pkg (131 MB)

Don't see your operating system here? Try one of the [other downloads](#).

Note: By default, the go command downloads and authenticates modules using the Go module mirror and Go checksum database run by Google. [Learn more](#).

2. Go install.

Select the tab for your computer's operating system below, then follow its installation instructions.

Linux Mac Windows

1. Open the package file you downloaded and follow the prompts to install Go.

The package installs the Go distribution to `/usr/local/go`. The package should put the `/usr/local/go/bin` directory in your PATH environment variable. You may need to restart any open Terminal sessions for the change to take effect.
2. Verify that you've installed Go by opening a command prompt and typing the following command:

```
$ go version
```
3. Confirm that the command prints the installed version of Go.

installing Go



```
$ go version  
go version go1.17.3 darwin/amd64
```

installing Go



```
$ go help
```

```
Go is a tool for managing Go source code.
```

```
Usage:
```

```
go <command> [arguments]
```

```
The commands are:
```

bug	start a bug report
build	compile packages and dependencies
clean	remove object files and cached files
doc	show documentation for package or symbol
env	print Go environment information
fix	update packages to use new APIs
fmt	gofmt (reformat) package sources
generate	generate Go files by processing source
get	add dependencies to current module and install them
install	compile and install packages and dependencies
list	list packages or modules
mod	module maintenance
run	compile and run Go program
test	test packages

{KODE {KLOUD

Hello World in Golang

Go program



main.go

```
package main
```

```
import "fmt"
```

```
// this is a comment
```

```
func main() {
```

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

```
}
```

types of comments

- single line comment

```
// comment
```

- multi line comment

```
/*  
multi  
line  
*/
```



main.go

```
package main  
import "fmt"
```

```
// this is a single line comment  
// this is a single line comment
```

```
func main() {
```

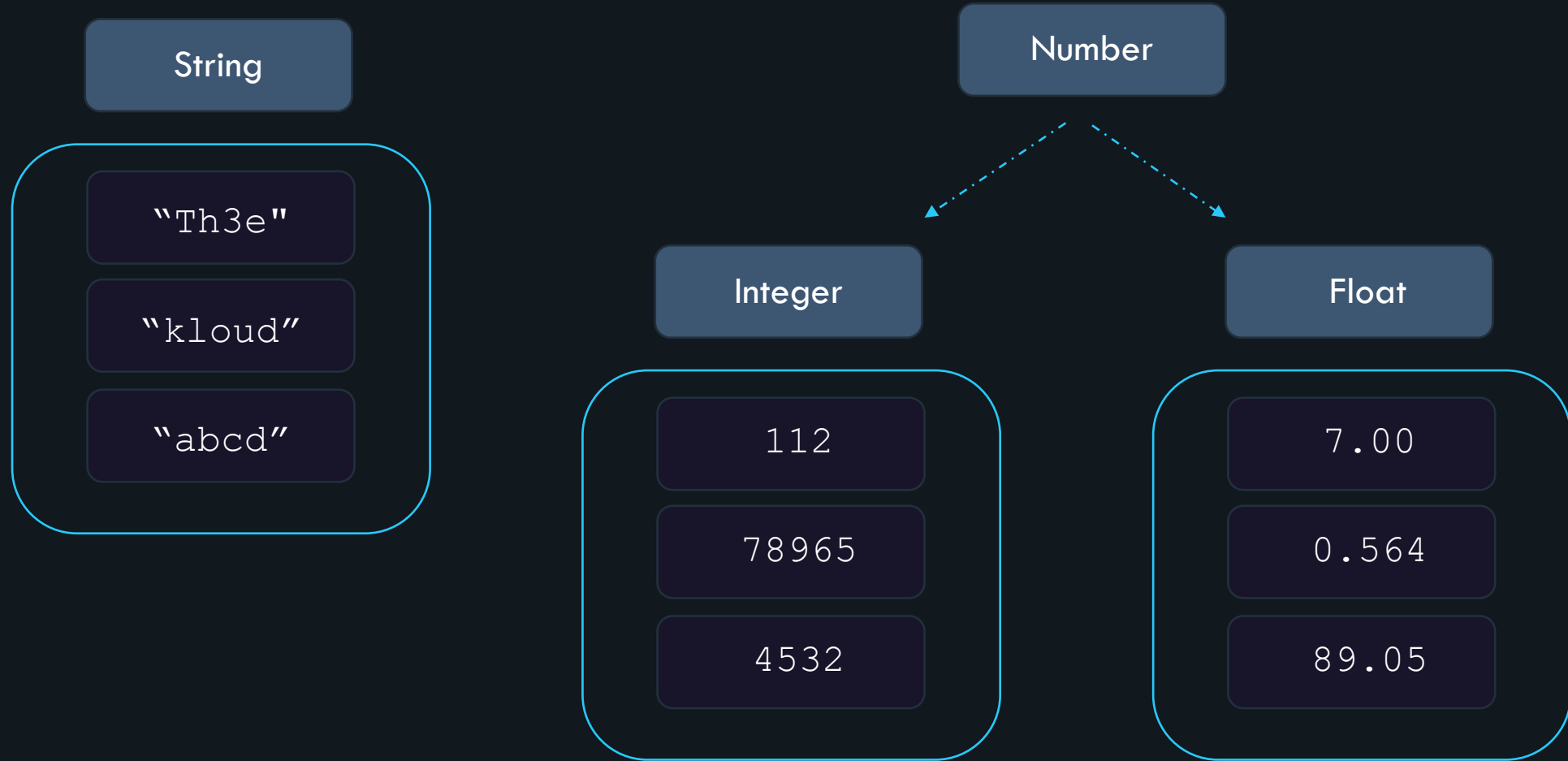
```
/*  
this is a  
multi line  
comment  
*/
```

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

```
}
```

{KODE {KLOUD

Data type



Boolean

```
true
```

```
false
```

Arrays & Slices

```
[1, 2, 4, 9]
```

```
["foo", "bar"]
```

```
[7.0, 9.43, 0.65]
```

Maps

```
"x" -> 30
```

```
1 -> 100
```

```
"key" -> "value"
```


Why are data types needed ?

- categorize a set of related values
- describe the operations that can be done on them

Number

$7 + 8$

$-7 * 19$

$-7 - 19$

String

"kodekloud"

to uppercase

"KODEKLOUD"

"kodekloud"

length

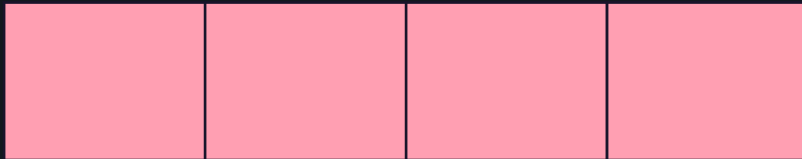
9

Why are data types needed ?

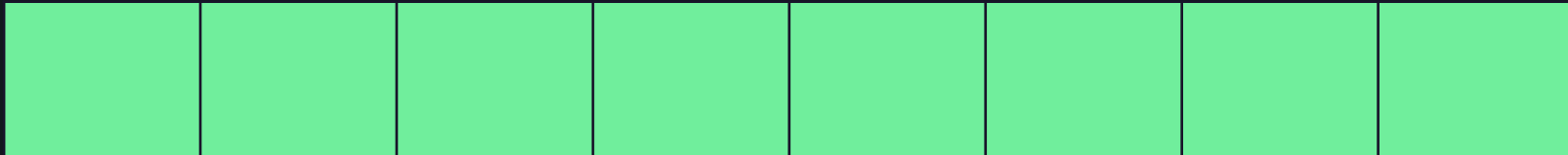
- categorize a set of related values
- describe the operations that can be done on them
- define the way the data is stored

memory allocation

Boolean



1 byte⁴ bytes
(32-bit machine)



8 bytes
(64-bit machine)

why are data types needed ?

- categorize a set of related values
- describe the operations that can be done on them
- define the way the data is stored

{KODE {KLOUD

Static vs Dynamic typed languages

Static typed

- Compiler throws an error when types are used incorrectly.
- Examples: C++, Java

main.cpp

```
void add(int a, int b) {  
    cout<<a+b  
}
```

```
add(1,2) => 3
```

```
add(1,"two") => ERROR
```

Dynamic typed

- Compiler does not enforce the type system.
- Examples: Python, Javascript

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

```
add(1,2) => 3
```

```
add(1,"two") => 1two
```

Static typed advantages:

- Better performance.
- Bugs can often be caught by a compiler.
- Better data integrity.

Dynamic typed advantages:

- Faster to write code.
- Generally, less rigid.
- Shorter learning curve.

Golang

- Go has a concept of types that is either explicitly declared by a programmer or is inferred by the compiler.
- It's a fast, **statically typed**, compiled language that **feels like a dynamically typed**, interpreted language.



Golang



main.go

```
package main
import ("fmt")

func main() {

    name := "Lisa"
    fmt.Println(name)

}
```

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

Lisa

{KODE {KLOUD

Kinds of Data types

Numbers

Integer

Float

Integers

```
int
```

```
200
```

```
-900
```

```
-90
```

```
1000000
```

```
453
```

Integers

- `uint` means “unsigned integer”.
- `int` means “signed integer”.

Data Type	Memory
<code>uint8</code>	8 bits or 1 byte
<code>uint16</code>	16 bits or 2 bytes
<code>uint32</code>	32 bits or 4 bytes
<code>uint64</code>	64 bits or 8 bytes
<code>int8</code>	8 bits or 1 byte
<code>int16</code>	16 bits or 2 bytes
<code>int32</code>	32 bits or 4 bytes
<code>int64</code>	64 bits or 8 bytes
<code>int</code>	4 bytes for 32-bit machines, 8 bytes for 64-bit machines

Float

```
float64
```

```
80.09
```

```
50.76
```

```
0.775
```

```
543333.2
```

```
654.11
```

Float

Data Type	Memory
float32	32 bits or 4 bytes
float64	64 bits or 8 bytes

String

(16 bytes)

```
string
```

```
"abc"
```

```
"90%"
```

```
"home"
```

```
"cat"
```

```
"kodekloud"
```

Boolean

(1 byte)

```
bool
```

```
true
```

```
false
```

Arrays, Slices and Maps

Arrays & Slices

```
[1, 2, 4, 9]
```

```
["foo", "bar"]
```

```
[7.0, 9.43, 0.65]
```

Maps

```
"x" -> 30
```

```
1 -> 100
```

```
"key" -> "value"
```


{KODE {KLOUD

Variables

Variables

name

"Harry"

pincode

778866

grades

89.05

Declaring Variables

- Go is statically typed.
- Variables are assigned a type, either explicitly or implicitly.

Syntax:

```
var
```

```
<variable  
name>
```

```
<data  
type>
```

```
=
```

```
<value>
```

Syntax:

```
var
```

```
s
```

```
string
```

```
=
```

```
"Hello world"
```

```
var
```

```
i
```

```
int
```

```
=
```

```
100
```

```
var
```

```
b
```

```
bool
```

```
=
```

```
false
```

```
var
```

```
f
```

```
float64
```

```
=
```

```
77.90
```

Declaring Variables

```
main.go

package main
import ("fmt")

func main() {

    var greeting string = "Hello World"
    fmt.Println(greeting)

}

>>> go run main.go
Hello World
```

{KODE {KLOUD

Printing Variables

Printing a string

```
main.go

package main
import "fmt"
func main() {

    fmt.Print("Hello World")

}

>>> go run main.go
Hello World
```

Printing a variable



main.go

```
package main
import "fmt"

func main() {
    var city string = "Kolkata"
    fmt.Print(city)
}
```

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

```
Kolkata
```

Print variable and string



main.go

```
package main
import "fmt"

func main() {

    var name string = "KodeKloud"
    var user string = "Harry"

    fmt.Print("Welcome to ", name, ", ", " ", user)

}
```

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

```
Welcome to KodeKloud, Harry
```

Printing on newline



main.go

```
package main
import "fmt"

func main() {
    var name string = "KodeKloud"
    var user string = "Harry"
    fmt.Print(name)
    fmt.Print(user)
}
```

```
>>> go run main.go
KodeKloudHarry
```

newline character

```
\n
```

- `\n` is called the `Newline` character.
- It is used to create a new line.
- Placed within string expressions.
- When inserted in a string, all the characters after `\n` are added to a new line.

Printing on newline



main.go

```
package main
import ("fmt")

func main() {
    var name string = "KodeKloud"
    var user string = "Harry"
    fmt.Print(name, "\n")
    fmt.Print(user)
}
```

```
>>> go run main.go
KodeKloud
Harry
```

Print In



main.go

```
package main
import ("fmt")

func main() {
    var name string = "KodeKloud"
    var user string = "Harry"
    fmt.Println(name)
    fmt.Println(user)
}
```

```
>>> go run main.go
KodeKloud
Harry
```


Printf

```
fmt.Printf("Template string %s", Object args(s))
```

Printf – format specifier

`%v`

- `%v` formats the value in a default format.

```
var name string = "KodeKloud"  
fmt.Printf("Nice to see you here, at %v", name)  
>>> Nice to see you here, at KodeKloud
```

Printf – format specifier

`%d`

- `%d` formats decimal integers.

```
var grades int = 42

fmt.Printf("Marks: %d", grades)

>>> Marks: 42
```

Print f



main.go

```
package main
import ("fmt")

func main() {
    var name string = "Joe"
    var i int = 78
    fmt.Printf("Hey, %v! You have scored %d/100 in Physics", name, i)
}
```

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

```
Hey, Joe! You have scored 78/100 in Physics
```

Printf – format specifiers

Verb	Description
<code>%v</code>	default format
<code>%T</code>	type of the value
<code>%d</code>	integers
<code>%c</code>	character
<code>%q</code>	quoted characters/string
<code>%s</code>	plain string
<code>%t</code>	true or false
<code>%f</code>	floating numbers
<code>%.2f</code>	floating numbers upto 2 decimal places

{KODE {KLOUD

Declaring Variables

Declaration + Initialization



main.go

```
package main
import ("fmt")

func main() {
    var user string

    user = "Harry"

    fmt.Println(user)
}
```

```
>>> go run main.go
Harry
```


Incorrect Values



main.go

```
package main
import ("fmt")

func main() {
    var s string
    s = 123
    fmt.Println(s)
}
```

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

```
Error: cannot use 123 (type untyped int) as type string
in assignment
```

Shorthand way



main.go

```
package main
import ("fmt")

func main() {
    var s,t string = "foo", "bar"
    fmt.Println(s)
    fmt.Println(t)
}
```

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

```
foo
```

```
bar
```

Shorthand way



main.go

```
package main
import ("fmt")

func main() {
    var (
        s string = "foo"
        i int = 5)

    fmt.Println(s)
    fmt.Println(i)
}
```

```
>>> go run main.go
foo
5
```

Short Variable Declaration

`s``:=``"Hello World"`

Short Variable Declaration



main.go

```
package main
import ("fmt")

func main() {

    name := "Lisa"
    name = "Peter"
    fmt.Println(name)

}
```

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

Peter

Short Variable Declaration



main.go

```
package main
import ("fmt")

func main() {
    name := "Lisa"
    name = 12
    fmt.Println(name)
}
```

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

```
Error: cannot use 12 (type untyped int) as type
string in assignment
```

{KODE {KLOUD

Variable Scope

Block

- Inner blocks can access variables declared within outer blocks.
- Outer blocks cannot access variables declared within inner blocks.

```
{  
  // outer block  
  
  {  
    // inner block  
  }  
}
```

Inner & Outer Block



main.go

```
func main() {  
    city := "London"  
    {  
        country := "UK"  
        fmt.Println(country)  
        fmt.Println(city)  
    }  
    fmt.Println(country)  
    fmt.Println(city)  
}
```

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

```
UK  
Error: ./main.go: Line 10: undefined: country  
London  
London
```

Local vs Global Variables

Local Variables

- Declared inside a function or a block.
- not accessible outside the function or the block.
- can also be declared inside looping and conditional statements.

Local Variables



main.go

```
package main
import ("fmt")

func main() {

    name := "Lisa"
    fmt.Println(name)

}
```

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

```
Lisa
```

Global Variables

- Declared outside of a function or a block.
- available throughout the lifetime of a program.
- declared at the top of the program outside all functions or blocks.
- can be accessed from any part of the program.

Global Variables



main.go

```
package main
import ("fmt")

var name string = "Lisa"

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

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

```
Lisa
```

{KODE {KLOUD

Zero Values

zero values

bool

false

int

0

float64

0.0

string

""

zero values - int



main.go

```
package main
import "fmt"

func main() {
    var i int
    fmt.Printf("%d", i)
}
```

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

```
0
```

zero values - float



main.go

```
package main
import "fmt"

func main() {
    var f1 float64
    fmt.Printf("%.2f", f1)
}
```

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

```
0.00
```

zero values

Verb	Description
int	0
float64	0.0
string	""
bool	false
pointers, functions, interfaces, maps	nil

{KODE {KLOUD

User Input

Scanf

```
fmt.Sprintf("%<format specifier > (s)", Object_arguments)
```


Single input



main.go

```
package main
import "fmt"

func main() {

    var name string
    fmt.Print("Enter your name: ")
    fmt.Scanf("%s", &name)
    fmt.Println("Hey there, ", name)

}
```

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

```
Enter your name: Priyanka
```

```
Hey there, Priyanka
```

Multiple Input



main.go

```
package main
import ("fmt")

func main() {

    var name string
    var is_muggle bool

    fmt.Print("Enter your name & are you a muggle: ")
    fmt.Scanf("%s %t", &name, &is_muggle)

    fmt.Println(name, is_muggle)

}
```

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

```
Enter your name & are you a muggle: Hermione false
```

```
Hermione false
```

Scanf return values

`count`

the number of arguments that the function writes to

`err`

any error thrown during the execution of the function

Multiple Input



main.go

```
package main
import ("fmt")

func main() {
    var a string
    var b int

    fmt.Print("Enter a string and a number: ")

    count, err := fmt.Scanf("%s %d", &a, &b)

    fmt.Println("count : ", count)
    fmt.Println("error: ", err)
    fmt.Println("a: ", a)
    fmt.Println("b: ", b)
}
```

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

```
Enter a string and a number: Priyanka yes
count : 1
error: expected integer
a: priyanka
b: 0
```

{KODE {KLOUD

find type of variable

type of variable

- `%T` format specifier
- `reflect.TypeOf` function from the reflect package.

using %T



main.go

```
package main
import "fmt"

func main() {
    var grades int = 42
    var message string = "hello world"
    var isCheck bool = true
    var amount float32 = 5466.54

    fmt.Printf("variable grades = %v is of type %T \n", grades, grades)
    fmt.Printf("variable message = '%v' is of type %T \n", message, message)
    fmt.Printf("variable isCheck = '%v' is of type %T \n", isCheck, isCheck)
    fmt.Printf("variable amount = %v is of type %T \n", amount, amount)
}

>>> go run main.go

variable grades = 42 is of type int
variable message = 'hello world' is of type string
variable isCheck = 'true' is of type bool
variable amount = 5466.54 is of type float32
```


using reflect.TypeOf()



main.go

```
package main
import (
    "fmt"
    "reflect"
)

func main() {
    fmt.Printf("Type: %v \n", reflect.TypeOf(1000))
    fmt.Printf("Type: %v \n", reflect.TypeOf("priyanka"))
    fmt.Printf("Type: %v \n", reflect.TypeOf(46.0))
    fmt.Printf("Type: %v \n", reflect.TypeOf(true))
}

>>> go run main.go

Type: int
Type: string
Type: float64
Type: bool
```

using `reflect.TypeOf()`

main.go

```
package main
import (
    "fmt"
    "reflect")

func main() {
    var grades int = 42
    var message string = "hello world"

    fmt.Printf("variable grades=%v is of type %v \n", grades, reflect.TypeOf(grades))
    fmt.Printf("variable message='%v' is of type %v \n", message, reflect.TypeOf(message))
}
```

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

```
variable grades = 42 is of type int
variable message = 'hello world' is of type string
```

{KODE {KLOUD

converting between data types

Type Casting

- The process of converting one data type to another is known as Type Casting.
- Data types can be converted to other data types, but this does not guarantee that the value will remain intact.

integer to float



main.go

```
package main
import "fmt"

func main() {
    var i int = 90
    var f float64 = float64(i)
    fmt.Printf("%.2f\n", f)
}
```

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

```
90.00
```

float to integer



main.go

```
package main
import "fmt"

func main() {
    var f float64 = 45.89
    var i int = int(f)
    fmt.Printf("%v\n", i)
}
```

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

```
45
```

strconv package

Itoa ()

- converts integer to string
- returns one value – string formed with the given integer.

integer to string

main.go

```
package main
import (
    "fmt"
    "strconv")

func main() {
    var i int = 42
    var s string = strconv.Itoa(i) // convert int to string
    fmt.Printf("%q", s)
}
```

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

```
"42"
```

strconv package

Itoa()

- converts integer to string.
- returns one value – string formed with the given integer.

Atoi()

- converts string to integer.
- returns two values – the corresponding integer, error (if any).

string to integer



main.go

```
package main
import (
    "fmt"
    "strconv")

func main() {
    var s string = "200"
    i, err := strconv.Atoi(s)
    fmt.Printf("%v, %T \n", i, i)
    fmt.Printf("%v, %T", err, err)
}
```

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

```
200, int
```

```
<nil>, <nil>
```

string to integer



main.go

```
package main
import (
    "fmt"
    "strconv")

func main() {
    var s string = "200abc"
    i, err := strconv.Atoi(s)
    fmt.Printf("%v, %T \n", i, i)
    fmt.Printf("%v, %T", err, err)
}
```

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

```
0, int
```

```
strconv.Atoi: parsing "200a": invalid syntax, *strconv.NumError
```

{KODE {KLOUD

Constants

Syntax:

const

<const
name>

<data
type>

=

<value>

Untyped constant

- constants are untyped unless they are explicitly given a type at declaration.
- allow for flexibility.

```
const age = 12
```

```
const h_name, h_age = "Hermione", 12
```


Typed constant

- constants are typed when you explicitly specify the type in the declaration.
- flexibility that comes with untyped constants is lost.

```
const name string = "Harry Potter"
```

```
const age int = 12
```

Understanding constants

main.go

```
package main
import "fmt"

func main() {
    const name = "Harry Potter"
    const is_muggle = false
    const age = 12

    fmt.Printf("%v: %T \n", name, name)
    fmt.Printf("%v: %T \n", is_muggle, is_muggle)
    fmt.Printf("%v: %T", age, age)
}

>>> go run main.go
Harry Potter: string
false: bool
12: int
```

Understanding constants



main.go

```
package main
import "fmt"

func main() {
    const name = "Harry Potter"
    name = "Hermione Granger"
    fmt.Printf("%v: %T \n", name, name)
}
```

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

```
Error: cannot assign to name (declared const)
```

Understanding constants



main.go

```
package main
import "fmt"

func main() {
    const name
    name = "Hermione Granger"
    fmt.Printf("%v: %T \n", name, name)
}
```

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

```
missing value in const declaration,
undefined: name
```

Understanding constants



main.go

```
package main
import "fmt"

func main() {
    const name := "Hermione Granger"
    fmt.Printf("%v: %T \n", name, name)
}
```

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

```
Error: syntax error: unexpected :=, expecting =
```

Understanding constants

```
package main
import "fmt"

const PI float64 = 3.14 // global constant

func main() {
    var radius float64 = 5.0
    var area float64
    area = PI * radius * radius
    fmt.Println("Area of Circle is : ", area)
}

>>> go run main.go
Area of Circle is : 78.5
```

{KODE {KLOUD

Operators

Operators & Operands



Kinds of Operators

Comparison
Operators

==

!=

<

<=

>

>=

Arithmetic
Operators

+

-

*

/

%

++

--

Kinds of Operators

Assignment Operators

=

+=

-=

*=

/=

%=

Bitwise Operators

&

|

<<

>>

^

Kinds of Operators

Logical
Operators

&&

||

!

{KODE {KLOUD

Comparison Operators

Comparison Operators

- compare two operands and yield a Boolean value.
- allow values of the same data type for comparisons
- common comparisons -
 - Does one string match another ?
 - Are two numbers the same ?
 - Is one number greater than another ?

Comparison Operators

A dark purple rounded square icon with a white border, containing the text '==' in white.

`==`
equal

A dark purple rounded square icon with a white border, containing the text '!=' in white.

`!=`
not
equal

A dark purple rounded square icon with a white border, containing the text '<' in white.

`<`
less
than

A dark purple rounded square icon with a white border, containing the text '<=' in white.

`<=`
less
than or
equal to

A dark purple rounded square icon with a white border, containing the text '>' in white.

`>`
greater
than

A dark purple rounded square icon with a white border, containing the text '>=' in white.

`>=`
greater
than or
equal to

Equal to (==)

- returns True when the values are equal.

```
package main
import "fmt"

func main() {
    var city string = "Kolkata"
    var city_2 string = "Calcutta"
    fmt.Println(city == city_2)
}
```

```
>>> go run main.go
false
```

not equal to (!=)

- returns True when the values are not equal.

```
package main
import "fmt"

func main() {
    var city string = "Kolkata"
    var city_2 string = "Calcutta"
    fmt.Println(city != city_2)
}
```

```
>>> go run main.go
true
```

less than (<)

- returns True when the left operand is lesser than the right operand.

```
package main
import "fmt"

func main() {
    var a, b int = 5, 10
    fmt.Println(a < b)
}
```

```
>>> go run main.go
true
```

less than or equal to (<=)

- returns True when the left operand is lesser or equal to the right operand.

```
package main
import "fmt"

func main() {
    var a, b int = 10, 10
    fmt.Println(a <= b)
}
```

```
>>> go run main.go
true
```

greater than (>)

- returns True when the left operand is greater than the right operand.

```
package main
import "fmt"

func main() {
    var a, b int = 20, 10
    fmt.Println(a > b)
}
```

```
>>> go run main.go
true
```

greater than or equal to (\geq)

- returns True when the left operand is greater or equal to the right operand.

```
package main
import "fmt"

func main() {
    var a, b int = 20, 20
    fmt.Println(a >= b)
}
```

```
>>> go run main.go
true
```

{KODE {KLOUD

Arithmetic Operators

Arithmetic Operators

- used to perform common arithmetic operations, such as addition, subtraction, multiplication etc.
- common comparisons -
 - Does the sum of two numbers equal a particular value?
 - Is the difference between two numbers lesser than a particular value?

Arithmetic Operators



addition



subtraction



multiplication



division



modulus



increment



decrement

addition (+)

- adds the left and right operand.

```
main.go

package main
import "fmt"

func main() {

    var a,b string = "foo", "bar"
    fmt.Println(a + b)

}

>>> go run main.go
foobar
```

subtraction (-)

- subtracts the right operand from the left operand.

```
main.go

package main
import "fmt"

func main() {
    var a,b string = "foo", "bar"
    fmt.Println(a - b)
}

>>> go run main.go
invalid operation: a - b (operator - not
defined on string)
```

subtraction (-)

- subtracts the right operand from the left operand.

```
package main
import "fmt"

func main() {

    var a, b float64 = 79.02, 75.66
    fmt.Printf("%.2f", a - b)

}
```

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

```
3.36
```

multiplication (*)

- multiplies both operands.

```
package main
import "fmt"

func main() {
    var a, b int = 12, 2
    fmt.Println(a * b)
}

>>> go run main.go
24
```

division (/)

- returns the quotient when left operand is divided by right operand.

```
package main
import "fmt"

func main() {
    var a, b int = 24, 2
    fmt.Println(a / b)
}
```

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

```
12
```

modulus (%)

- returns the remainder when left operand is divided by right operand.

```
package main
import "fmt"

func main() {
    var a, b int = 24, 7
    fmt.Println(a % b)
}
```

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

```
3
```


increment (++)

- unary operator.
- increments the value of the operand by one.

```
package main
import "fmt"

func main() {
    var i int = 1
    i++
    fmt.Println(i)
}
```

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

```
2
```

decrement (--)

- unary operator.
- decrements the value of the operand by one.

```
package main
import "fmt"

func main() {
    var i int = 1
    i--
    fmt.Println(i)
}
```

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

```
0
```

{KODE {KLOUD

Logical Operators

Logical Operators

- used to determine the logic between variables or values.
- common logical comparisons -
 - Are two variables both true ?
 - Does either of two expressions evaluate to true?

Logical Operators



&&

Logical
AND



||

Logical
OR



!

Logical
NOT

AND (&&)

- returns true if both the statements are true.
- returns false when either of the statement is false.

```
package main
import "fmt"

func main() {

    var x int = 10
    fmt.Println((x < 100) && (x < 200))
    fmt.Println((x < 300) && (x < 0))

}
```

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

```
true
```

```
false
```

OR (||)

- returns true if one of the statement is true.
- returns false when both statements are false.

```
package main
import "fmt"

func main() {

    var x int = 10
    fmt.Println((x < 0) || (x < 200))
    fmt.Println((x < 0) || (x > 200))

}
```

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

```
true
```

```
false
```


NOT (!)

- unary operator.
- Reverses the result, returns false if the expression evaluates to true and vice versa.

```
package main
import "fmt"

func main() {

    var x, y int = 10, 20
    fmt.Println(!(x > y))
    fmt.Println(!(true))
    fmt.Println(!(false))

}
```

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

```
true
false
true
```

{KODE {KLOUD

Assignment Operators

Assignment Operators



assign



add and
assign



subtract and
assign



multiply and
assign



divide and
assign
quotient



divide and
assign
modulus

assign (=)

- assigns left operand with the value to the right.

- `x = y`

```
package main
import "fmt"

func main() {
    var x int = 10
    var y int
    y = x
    fmt.Println(y)
}
```

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

```
10
```

add and assign (+=)

- assigns left operand with the addition result.
- `x += y` means `x = x + y`

```
package main
import "fmt"

func main() {

    var x, y int = 10, 20
    x += y
    fmt.Println(x)

}
```

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

```
30
```

subtract and assign (-=)

- assigns left operand with the subtraction result.
- `x -= y` means `x = x - y`

```
main.go

package main
import "fmt"

func main() {

    var x, y int = 10, 20
    x -= y
    fmt.Println(x)

}

>>> go run main.go
-10
```

multiply and assign (*=)

- assigns left operand with the multiplication result.
- `x *= y` means `x = x * y`

```
package main
import "fmt"

func main() {

    var x, y int = 10, 20
    x *= y
    fmt.Println(x)

}
```

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

```
200
```


divide and assign quotient (/=)

- assigns left operand with the quotient of the division.
- `x/= y` means `x = x / y`

```
package main
import "fmt"

func main() {

    var x, y int = 200, 20
    x /= y
    fmt.Println(x)

}
```

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

```
10
```

divide and assign modulus (%=)

- assigns left operand with the remainder of the division.
- `x%= y` means `x = x % y`

```
package main
import "fmt"

func main() {

    var x, y int = 210, 20
    x %= y
    fmt.Println(x)

}
```

```
>>> go run main.go
10
```

{KODE {KLOUD

Bitwise Operators

Bitwise Operators



bitwise
AND



bitwise
OR



bitwise
XOR



right
shift



left
shift

bitwise AND (&)

- takes two numbers as operands and does AND on every bit of two numbers.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

	0	0	0	0	1	1	0	0
&	0	0	0	1	1	0	0	1
	0	0	0	0	1	0	0	0

= 8 (In decimal)

bitwise AND (&)

```
package main
import "fmt"

func main() {

    var x, y int = 12, 25
    z := x & y
    fmt.Println(z)

}
```

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

```
8
```

bitwise OR (|)

- takes two numbers as operands and does OR on every bit of two numbers.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

	0	0	0	0	1	1	0	0
	0	0	0	1	1	0	0	1
<hr/>								
	0	0	0	1	1	0	1	= 29 (In decimal)

bitwise OR (|)

```
package main
import "fmt"

func main() {

    var x, y int = 12, 25
    z := x | y
    fmt.Println(z)

}
```

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

```
29
```

bitwise XOR (^)

- takes two numbers as operands and does XOR on every bit of two numbers.
- The result of XOR is 1 if the two bits are opposite.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

	0	0	0	0	1	1	0	0
^	0	0	0	1	1	0	0	1
<hr/>								
	0	0	0	1	0	1	0	1

= 21 (In decimal)

bitwise XOR (^)

```
package main
import "fmt"

func main() {

    var x, y int = 12, 25
    z := x ^ y
    fmt.Println(z)

}

>>> go run main.go
21
```

left shift (<<)

- shifts all bits towards left by a certain number of specified bits.
- The bit positions that have been vacated by the left shift operator are filled with 0.

212 = 11010100 (in binary)

212 << 1

11010100
110101000 = 424 (in decimal)

left shift (<<)

```
package main
import "fmt"

func main() {

    var x int = 212
    z := x << 1
    fmt.Println(z)

}
```

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

```
424
```

right shift (>>)

- shifts all bits towards right by a certain number of specified bits.
- excess bits shifted off to the right are discarded.

212 = 11010100 (in binary)

212 >> 2

11010100



00110101 = 53 (in decimal)

right shift (>>)

```
package main
import "fmt"

func main() {

    var x int = 212
    z := x >> 2
    fmt.Println(z)

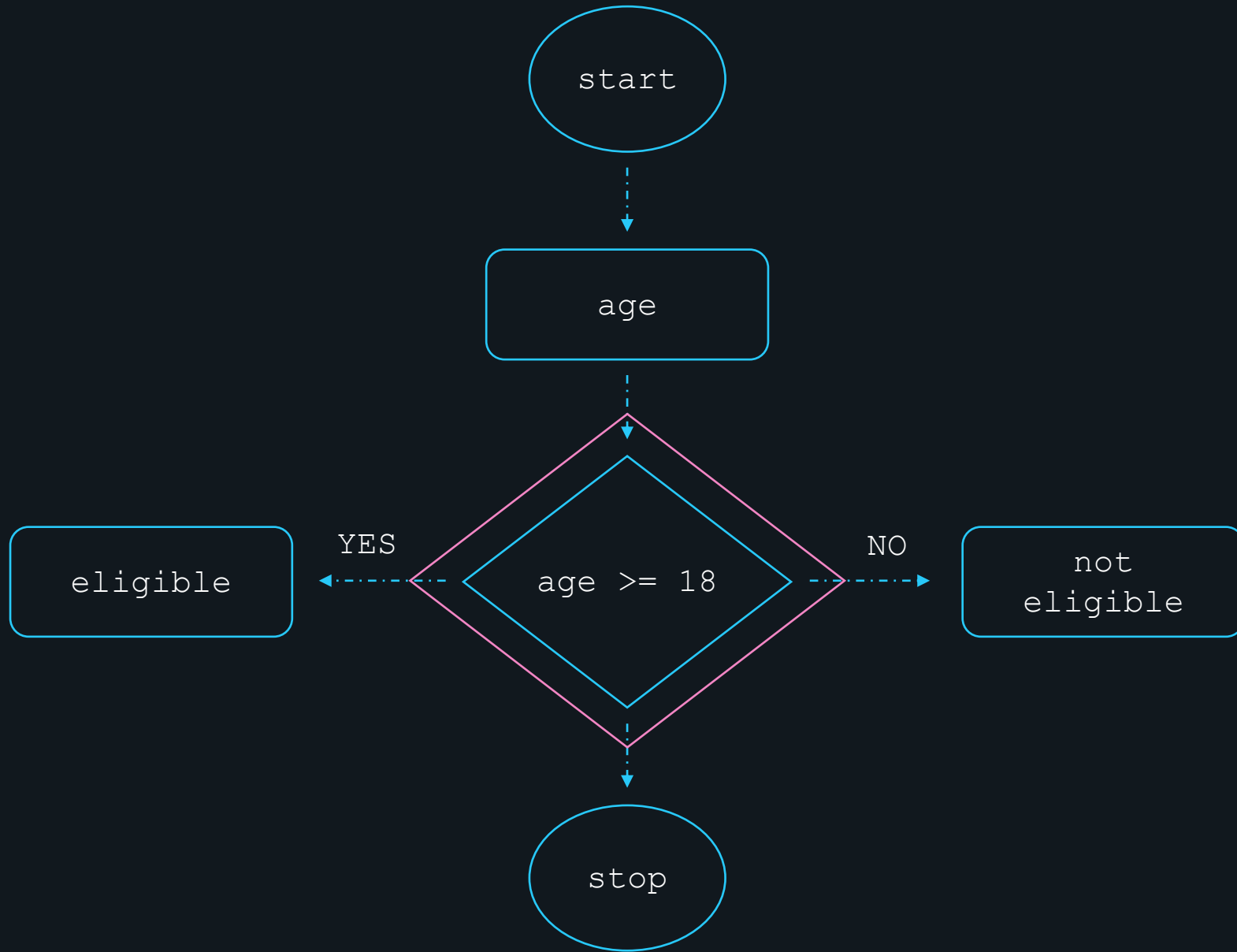
}
```

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

```
53
```

{KODE {KLOUD

Control Flow



if-else

syntax

```
if (condition) {
```

```
// executes when condition is true
```

```
}
```

if statement

```
package main
import "fmt"

func main() {
    var a string = "happy"
    if a == "happy" {
        fmt.Println(a)
    }
}

>>> go run main.go
happy
```

syntax

```
if condition {
```

```
// executes when condition is true
```

```
} else {
```

```
// executes when condition is false
```

```
}
```

if-else statement

```
package main
import "fmt"

func main() {
    var fruit string = "grapes"
    if fruit == "apples" {
        fmt.Println("Fruit is apple")
    }
    else {
        fmt.Println("Fruit is not apple")
    }
}

>>> go run main.go
syntax error: unexpected else, expecting }
```

if-else statement

```
main.go

package main
import "fmt"

func main() {

    var fruit string = "grapes"
    if fruit == "apples" {
        fmt.Println("Fruit is apple")
    } else {
        fmt.Println("Fruit is not apple")
    }

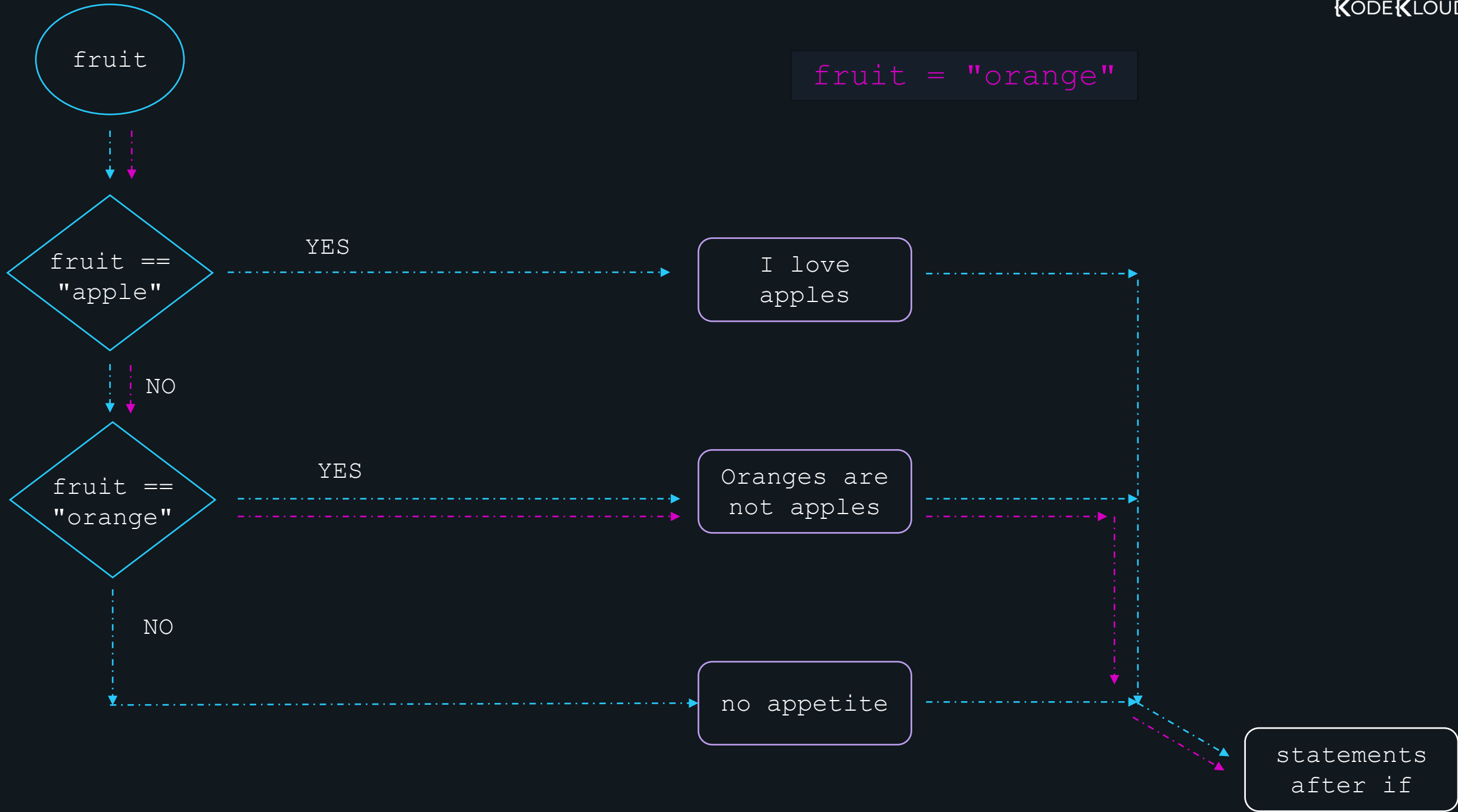
}

>>> go run main.go
Fruit is not apple
```



```
if condition_1 {  
    // execute when condition_1 is true  
  
} else if condition_2 {  
    /* execute when condition_1 is false,  
    and condition_2 is true */  
  
} else if condition_3 {  
    /* execute when condition_1 and 2 are false,  
    and condition_3 is true */  
  
} else {  
    // when none of the above conditions are true  
  
}
```

```
fruit = "orange"
```



if- else-if else statement

```
main.go

package main
import "fmt"

func main() {
    fruit := "grapes"
    if fruit == "apple" {
        fmt.Println("I love apples")
    } else if fruit == "orange" {
        fmt.Println("Oranges are not apples")
    } else {
        fmt.Println("no appetite")
    }
}

>>> go run main.go
no appetite
```

switch-case

syntax

```
switch expression {  
  case value_1:  
    // execute when expression equals to value_1  
  case value_2:  
    // execute when expression equals to value_2  
  default:  
    // execute when no match is found  
}
```

switch statement

```
main.go

package main
import "fmt"

func main() {
    var i int = 800
    switch i {
        case 10:
            fmt.Println("i is 10")
        case 100, 200:
            fmt.Println("i is either 100 or 200")
        default:
            fmt.Println("i is neither 0, 100 or 200")
    }
}

>>> go run main.go

i is neither 0, 100 or 200
```

fallthrough

- The `fallthrough` keyword is used in switch-case to force the execution flow to fall through the successive case block.

```
main.go

func main() {
    var i int = 10
    switch i {
        case -5:
            fmt.Println("-5")
        case 10:
            fmt.Println("10")
            fallthrough
        case 20:
            fmt.Println("20")
            fallthrough
        default:
            fmt.Println("default")
    }
}
```

10
20
default

switch with conditions

```
switch {  
  
  case condition_1:  
    // execute when condition_1 is true  
  
  case condition_2:  
    // execute when condition_2 is true  
  
  default:  
    // execute when no condition is true  
  
}
```


switch with conditions

```
main.go

func main() {
    var a, b int = 10, 20
    switch {
        case a+b == 30:
            fmt.Println("equal to 30")
        case a+b <= 30:
            fmt.Println("less than or equal to 30")
        default:
            fmt.Println("greater than 30")
    }
}
```

```
>>> go run main.go
equal to 30
```

{KODE {KLOUD

looping with for loop

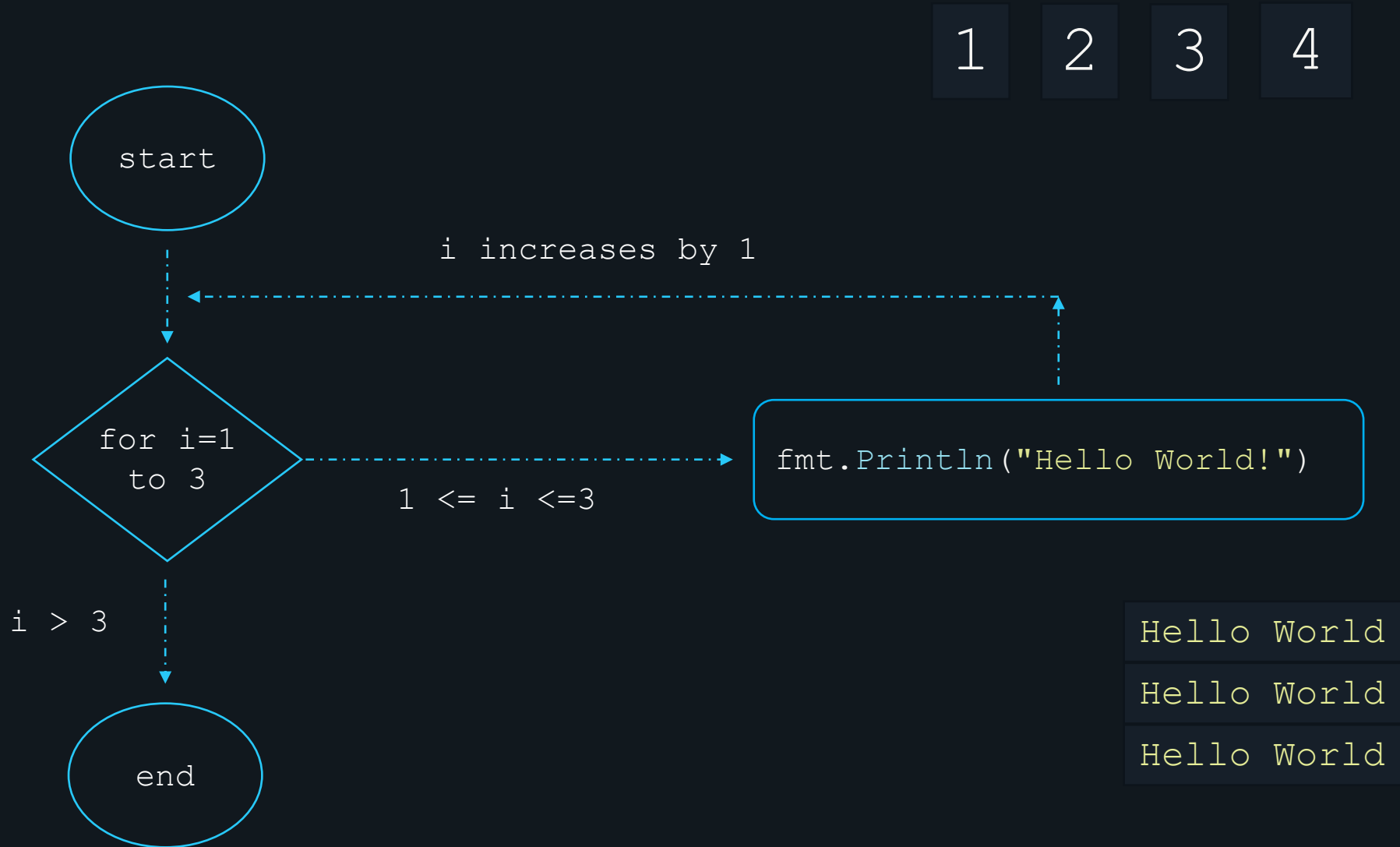
loop

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

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

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

loop



for loop syntax:

```
for [initialization; condition; post] {  
    // statements  
}
```

for loop

```
for i := 1; i <= 3; i++ {  
    fmt.Println("Hello World")  
}
```

for loop

```
main.go

package main
import "fmt"

func main() {
    for i := 1; i <= 5; i++ {
        fmt.Println(i*i)
    }
}

>>> go run main.go
1
4
9
16
25
```


for loop

```
main.go

package main
import "fmt"

func main() {
    i := 1
    for i <= 5 {
        fmt.Println(i * i)
        i += 1
    }
}

>>> go run main.go
1
4
9
16
25
```

infinite loop



main.go

```
package main
import "fmt"

func main() {
    sum := 0
    for {
        sum++ // repeated forever
    }
    fmt.Println(sum) // never reached
}
```

Break & Continue

break statement

- the `break` statement ends the loop immediately when it is encountered.

```
package main
import "fmt"

func main() {
    for i := 1; i <= 5; i++ {
        if i == 3 {
            break
        }
        fmt.Println(i)
    }
}

>>> go run main.go
1
2
```

continue statement

- the `continue` statement skips the current iteration of loop and continues with the next iteration.

```
package main
import "fmt"

func main() {
    for i := 1; i <= 5; i++ {
        if i == 3 {
            continue
        }
        fmt.Println(i)
    }
}

>>> go run main.go

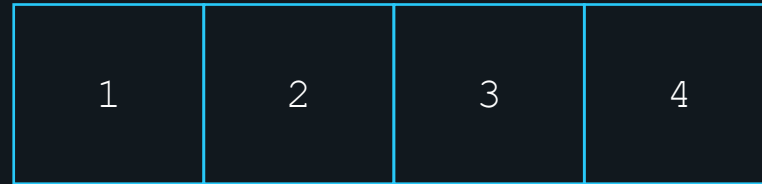
1
2
4
5
```

{KODE {KLOUD

Arrays

Arrays

- An array is a collection of similar data elements stored at contiguous memory locations.



Arrays

array
elements:



memory
address:

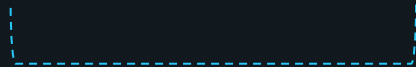
200

204

208

212

216



4 bytes

why we need arrays

grade_chem

90

grade_math

85

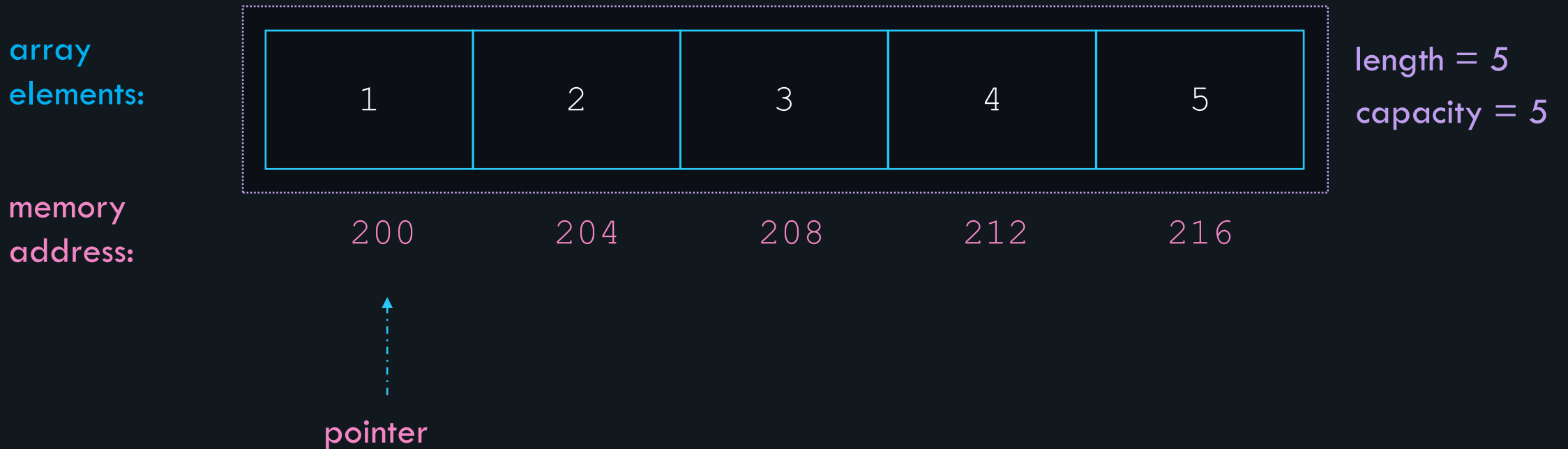
grade_phy

70

grades

Arrays

- fixed length.
- elements should be of the same data type



array declaration

syntax:

```
var
```

```
<array  
name>
```

```
[<size of  
the array>]
```

```
<data  
type>
```

```
var
```

```
grades
```

```
[5]
```

```
int
```

```
var
```

```
fruits
```

```
[3]
```

```
string
```

array declaration

```
main.go

package main
import "fmt"

func main() {

    var grades [5]int
    fmt.Println(grades)

    var fruits [3]string
    fmt.Println(fruits)

}

>>> go run main.go

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

array initialization

```
var grades [3]int = [3]int{10, 20, 30}
```

```
grades := [3]int{10, 20, 30}
```

```
grades := [...]int{10, 20, 30}
```

array initialization



main.go

```
package main
import "fmt"

func main() {
    var fruits [2]string = [2]string{"apples", "oranges"}
    fmt.Println(fruits)

    marks := [3]int{10, 20, 30}
    fmt.Println(marks)

    names := [...]string{"Rachel", "Phoebe", "Monica"}
    fmt.Println(names)
}

>>> go run main.go

[apples oranges]
[10 20 30]
[Rachel Phoebe Monica]
```

len ()

- The length of the array refers to the number of elements stored in the array.

array length



main.go

```
package main
import "fmt"

func main() {
    var fruits [2]string = [2]string{"apples", "oranges"}
    fmt.Println(len(fruits))
}
```

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

```
2
```

indexes in array

```
0 <= index <= len-1
```

grades:

90	86	76	42	85
----	----	----	----	----

index:

0	1	2	3	4
---	---	---	---	---

```
grades[1] => 86
```

```
grades[0] => 90
```

array indexing



main.go

```
package main
import "fmt"

func main() {

    var fruits [5]string = [5]string{"apples",
    "oranges", "grapes", "mango", "papaya"}

    fmt.Println(fruits[2])

}

>>> go run main.go
grapes
```

array indexing



main.go

```
package main
import "fmt"

func main() {

    var fruits [5]string = [5]string{"apples",
    "oranges", "grapes", "mango", "papaya"}

    fmt.Println(fruits[6])

}
```

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

```
invalid array index 6 (out of bounds for 5-
element array)
```

array indexing



main.go

```
package main
import "fmt"

func main() {

    var grades [5]int = [5]int{90, 80, 70, 80, 97}
    fmt.Println(grades)
    grades[1] = 100
    fmt.Println(grades)

}
```

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

```
[90 80 70 80 97]
[90 100 70 80 97]
```

looping through an array

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

looping through an array

```
main.go

package main
import "fmt"

func main() {

    var grades [5]int = [5]int{90, 80, 70, 80, 97}

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

>>> go run main.go
90
80
70
80
97
```

looping through an array

```
for index, element := range grades {  
    fmt.Println(index, "=>", element)  
}
```


looping through an array

```
main.go

package main
import "fmt"

func main() {

    var grades [5]int = [5]int{90, 80, 70, 80, 97}

    for index, element := range grades {
        fmt.Println(index, "=>", element)
    }
}

>>> go run main.go
0 => 90
1 => 80
2 => 70
3 => 80
4 => 97
```

multidimensional arrays



```
arr[2][1] => 64
```

```
arr[1][0] => 4
```

```
arr[0][0] => 2
```

multidimensional arrays

```
main.go

package main
import "fmt"

func main() {

    arr := [3][2]int{{2, 4}, {4, 16}, {8, 64}}
    fmt.Println(arr[2][1])

}
```

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

```
64
```

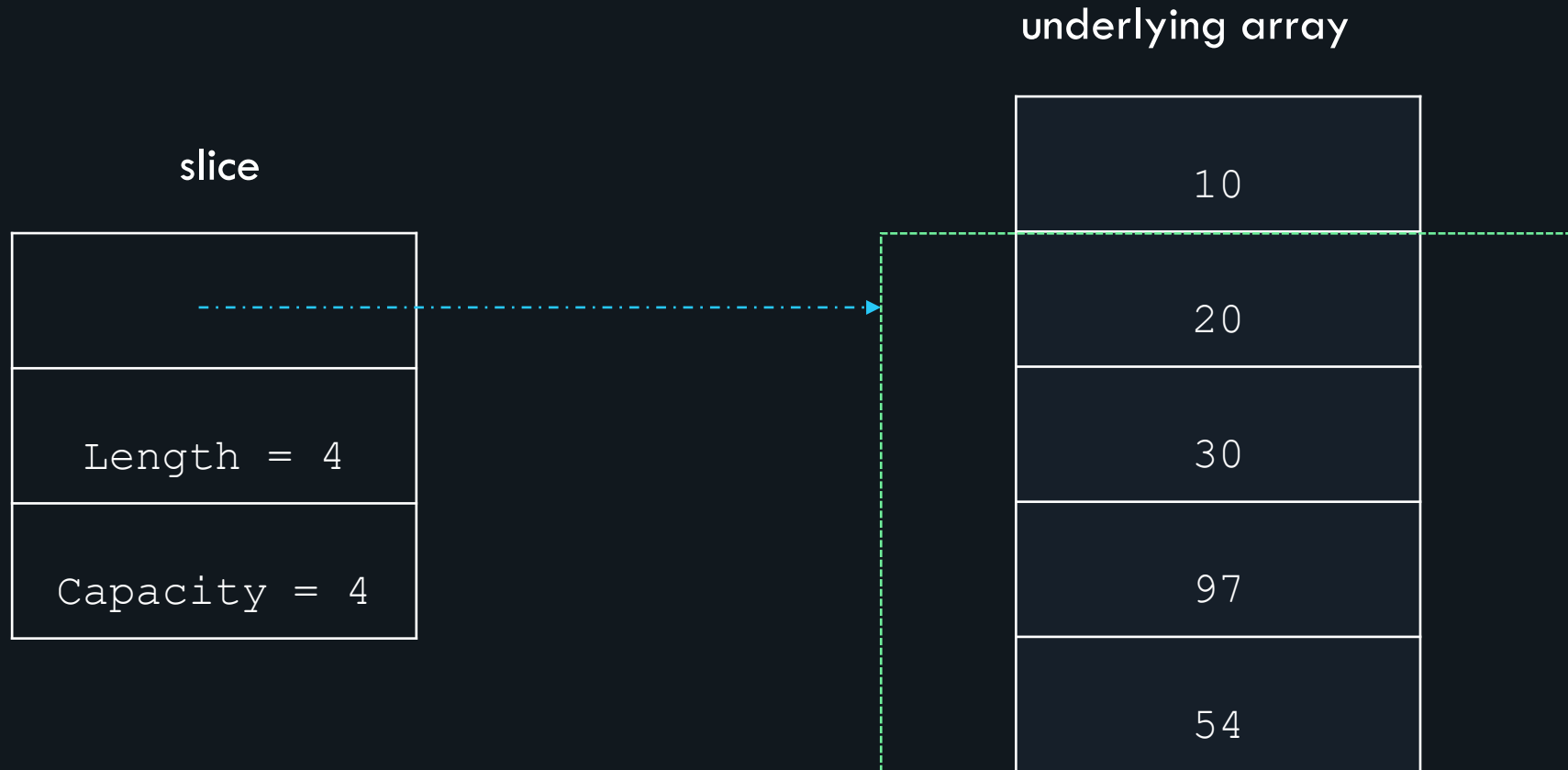
{KODE {KLOUD

Slice

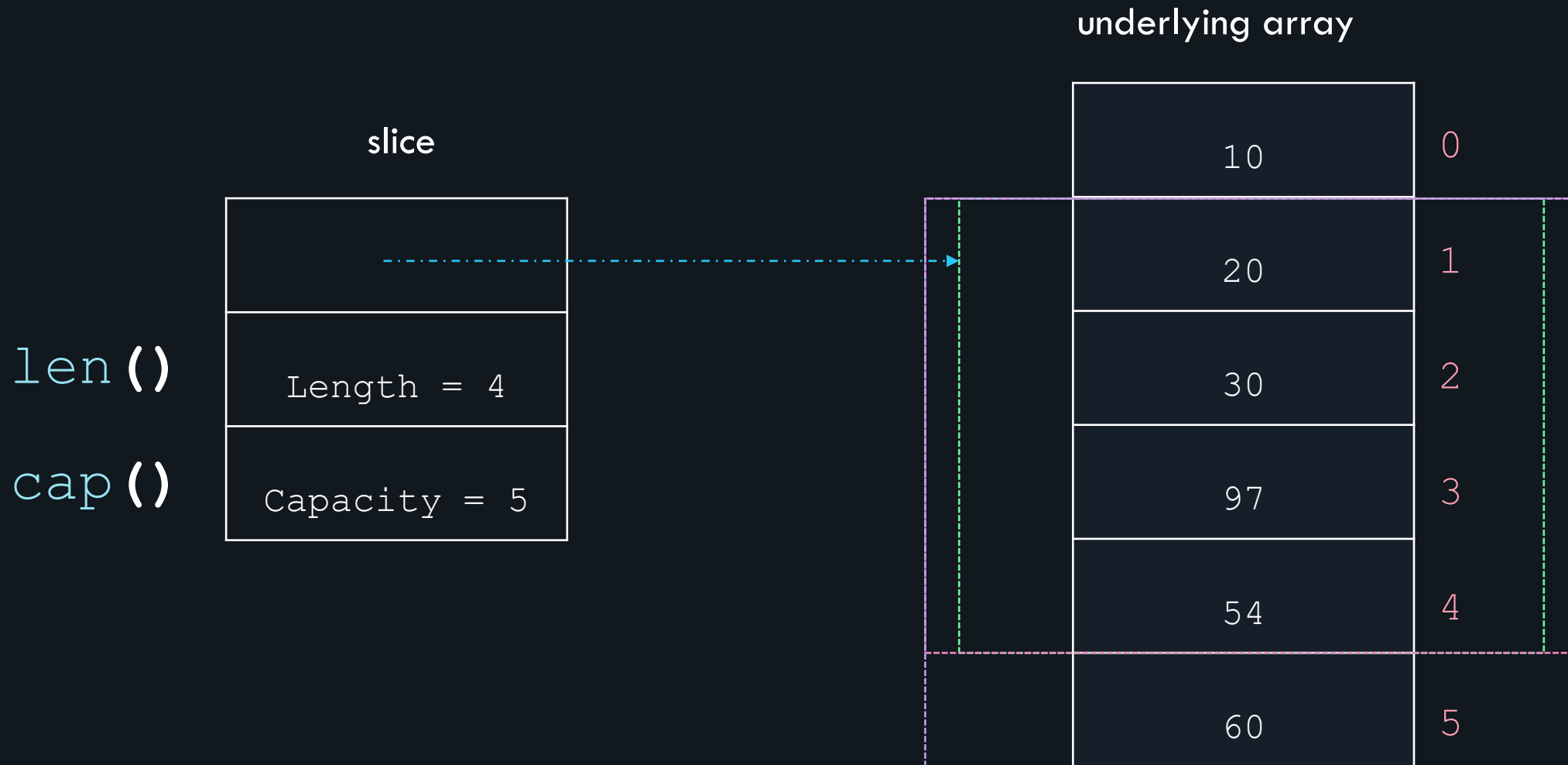
Slice

- continuous segment of an underlying array.
- variable typed (elements can be added or removed)
- more flexible

Slice



components of a Slice



declaring and initializing a slice

```
<slice_name> := []<data_type>{<values>}
```

```
grades := []int{10, 20, 30}
```

declaring and initializing a slice

```
package main
import "fmt"

func main() {

    slice := []int{10, 20, 30}

    fmt.Println(slice)

}
```

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

```
[10 20 30]
```

declaring and initializing a slice

```
array[start_index : end_index]
```

```
array[0 : 3]
```

```
array[1 : 6]
```

```
array[ : 4]
```

```
array[ : ]
```

underlying array

	10	0
	20	1
	30	2
	97	3
	54	4
	60	5

declaring and initializing a slice

```
main.go

package main
import "fmt"

func main() {

    arr := [10]int{10, 20, 30, 40, 50, 60, 70, 80,
    90, 100}

    slice_1 := arr[1:8]

    fmt.Println(slice_1)

}

>>> go run main.go

[20 30 40 50 60 70 80]
```

declaring and initializing a slice

```
main.go

package main
import "fmt"

func main() {

    arr := [10]int{10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

    slice := arr[1:8]
    fmt.Println(slice)

    sub_slice := slice[0:3]
    fmt.Println(sub_slice)

}
```

```
>>> go run main.go
[20 30 40 50 60 70 80]
[20 30 40]
```

declaring and initializing a slice

```
slice := make([]<data_type>, length, capacity)
```

```
slice := make([]int, 5, 10)
```

declaring and initializing a slice

```
main.go

package main
import "fmt"

func main() {

    slice := make([]int, 5, 8)

    fmt.Println(slice)

    fmt.Println(len(slice))

    fmt.Println(cap(slice))

}

>>> go run main.go
[0 0 0 0 0]
5
8
```

declaring and initializing a slice



main.go

```
package main
import "fmt"

func main() {

    arr := [10]int{10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
    slice := arr[1:8]

    fmt.Println(cap(arr))

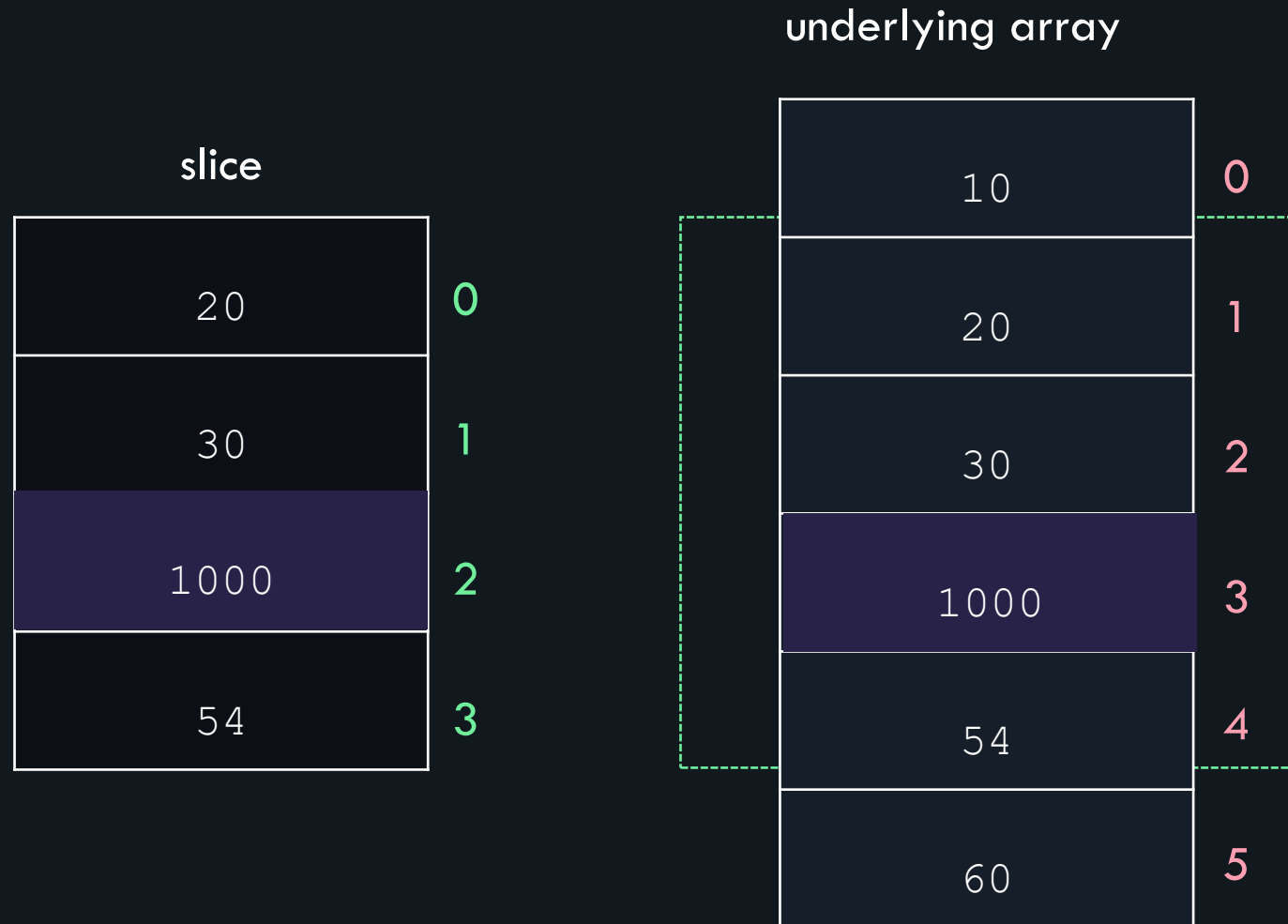
    fmt.Println(cap(slice))

}

>>> go run main.go

10
9
```


slice and index numbers



slice and index numbers

```
main.go

package main
import "fmt"

func main() {
    arr := [5]int{10, 20, 30, 40, 50}
    slice := arr[:3]

    fmt.Println(arr)
    fmt.Println(slice)

    slice[1] = 9000

    fmt.Println("after modification")
    fmt.Println(arr)
    fmt.Println(slice)
}

>>> go run main.go

[10 20 30 40 50]
[10 20 30]
after modification
[10 9000 30 40 50]
[10 9000 30]
```

appending to a slice

```
func append(s []T, vs ...T) []T
```

```
slice = append(slice, element-1, element-2)
```

```
slice = append(slice, 10, 20, 30)
```

appending to a slice

underlying array

10
20
30
40

slice

```
len = 2  
cap = 3
```

slice_1

20
30
900

```
len = 3  
cap = 3
```

slice_2

20
30
-90
500

```
len = 4  
cap = 6
```

```
slice := arr[1:3]
```

```
slice_1 := append(slice, 900)
```

```
slice_2 := append(slice, -90, 500)
```

appending to a slice

```
package main
import "fmt"

func main() {
    arr := [4]int{10, 20, 30, 40}
    slice := arr[1:3]

    fmt.Println(slice)
    fmt.Println(len(slice))
    fmt.Println(cap(slice))

    slice = append(slice, 900, -90, 50)

    fmt.Println(slice)
    fmt.Println(len(slice))
    fmt.Println(cap(slice))
}
```

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

```
[20 30]
```

```
2
```

```
3
```

```
[20 30 900 -90 50]
```

```
5
```

```
6
```

appending to a slice

```
slice = append(slice, anotherSlice...)
```

appending to a slice

```
package main
import "fmt"

func main() {
    arr := [5]int{10, 20, 30, 40, 50}
    slice := arr[:2]
    arr_2 := [5]int{5, 15, 25, 35, 45}
    slice_2 := arr_2[:2]
    new_slice := append(slice, slice_2...)
    fmt.Println(new_slice)
}
```

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

```
[10 20 5 15]
```

deleting from a slice

```
package main
import "fmt"

func main() {

    arr := [5]int{10, 20, 30, 40, 50}
    i := 2

    fmt.Println(arr)

    slice_1 := arr[:i]
    slice_2 := arr[i+1:]

    new_slice := append(slice_1, slice_2...)

    fmt.Println(new_slice)

}
```

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

```
[10 20 30 40 50]
```

```
[10 20 40 50]
```


copying from a slice

```
func copy(dst, src []Type) int
```

```
num := copy(dest_slice, src_slice)
```

copying from a slice

```
package main
import "fmt"

func main() {
    src_slice := []int{10, 20, 30, 40, 50}
    dest_slice := make([]int, 3)
    num := copy(dest_slice, src_slice)
    fmt.Println(dest_slice)
    fmt.Println("Number of elements copied: ", num)
}
```

```
>>> go run main.go
[10 20 30]
Number of elements copied: 3
```

looping through a slice

```
main.go

package main
import "fmt"

func main() {
    arr := []int{10, 20, 30, 40, 50}

    for index, value := range arr {
        fmt.Println(index, "=>", value)
    }
}

>>> go run main.go
0 => 10
1 => 20
2 => 30
3 => 40
4 => 50
```

looping through a slice

```
main.go

package main
import "fmt"

func main() {
    arr := []int{10, 20, 30, 40, 50}

    for _, value := range arr {
        fmt.Println(value)
    }
}

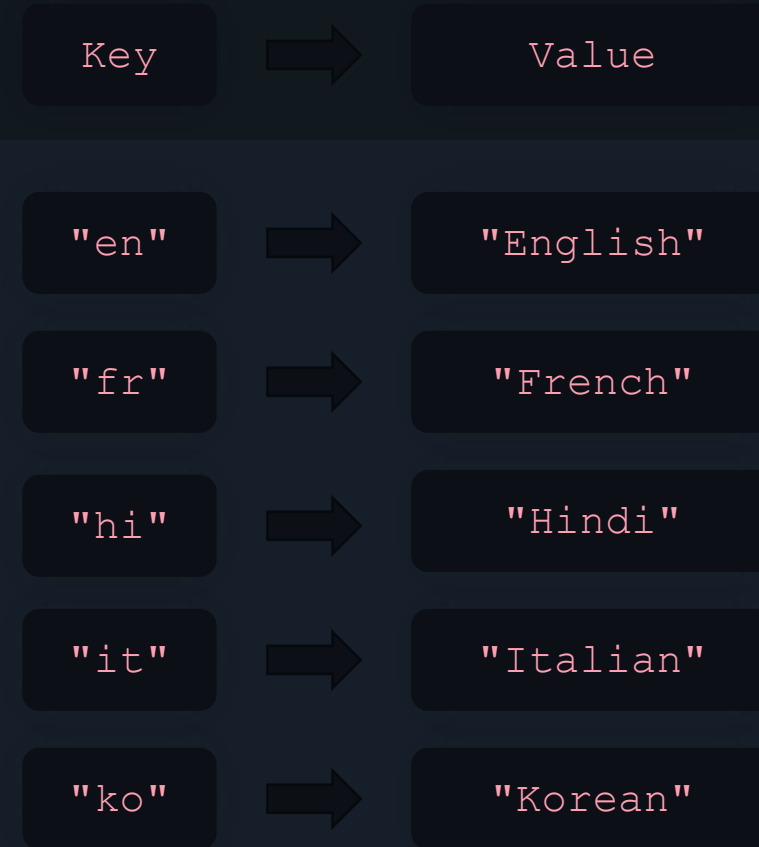
>>> go run main.go
10
20
30
40
50
```

{KODE {KLOUD

maps

maps

- unordered collection of key/value pairs.
- implemented by hash tables.
- provide efficient add, get and delete operations.



declaring and initializing a map

```
var <map_name> map[<key_data_type>] <value_data_type>
```

```
var my_map map[string] int
```


declaring and initializing a map

```
main.go

package main
import "fmt"

func main() {

    var codes map[string]string

    codes["en"] = "English"

    fmt.Println(codes)

}

>>> go run main.go

panic: assignment to entry in nil map
```

declaring and initializing a map

```
<map_name> := map[<key_data_type>]<value_data_type>{<key-value-pairs>}
```

```
codes := map[string]string{"en": "English", "fr": "French"}
```

declaring and initializing a map

```
package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
                                "fr": "French"}

    fmt.Println(codes)
}
```

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

```
map[en:English fr:French]
```

declaring and initializing a map – make () function

```
<map_name> :=  
make(map[<key_data_type>]<value_data_type>,  
<initial_capacity>)
```

declaring and initializing a map

```
package main
import "fmt"

func main() {
    codes := make(map[string]int)

    fmt.Println(codes)
}
```

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

```
map[]
```

length of a map

- `len()`

```
package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}

    fmt.Println(len(codes))
}

>>> go run main.go

3
```

accessing a map

- `map[key]`

```
main.go

package main
import "fmt"

func main() {

    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}

    fmt.Println(codes["en"])
    fmt.Println(codes["fr"])
    fmt.Println(codes["hi"])

}

>>> go run main.go

English
French
Hindi
```

getting a key

```
value, found := map_name[key]
```


getting a key

- `map[key]`



main.go

```
package main
import "fmt"

func main() {
    codes := map[string]int{"en": 1, "fr": 2, "hi": 3}
    value, found := codes["en"]
    fmt.Println(found, value)
    value, found = codes["hh"]
    fmt.Println(found, value)
}

>>> go run main.go
true 1
false 0
```

adding key-value pair

- `map[key] = value`

```
package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}

    codes["it"] = "Italian"
    fmt.Println(codes)
}
```

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

```
map[en:English fr:French hi:Hindi it:Italian]
```

update key-value pair

- `map[key] = value`

```
main.go

package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}

    codes["en"] = "English Language"

    fmt.Println(codes)
}

>>> go run main.go

map[en:English Language fr:French hi:Hindi]
```

delete key-value pair

- `delete(map, key_name)`

```
main.go

package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}
    fmt.Println(codes)
    delete(codes, "en")
    fmt.Println(codes)
}

>>> go run main.go

map[en:English fr:French hi:Hindi]
map[fr:French hi:Hindi]
```

iterate over a map

```
package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}
    for key, value := range codes {
        fmt.Println(key, "=>", value)
    }
}
```

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

```
en => English
fr => French
hi => Hindi
```

truncate a map

```
package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}

    for key, value := range codes {
        delete(codes, key)
    }

    fmt.Println(codes)
}

>>> go run main.go

map[]
```

truncate a map

```
main.go

package main
import "fmt"

func main() {
    codes := map[string]string{"en": "English",
    "fr": "French", "hi": "Hindi"}

    codes = make(map[string]string)

    fmt.Println(codes)
}

>>> go run main.go

map[]
```

{KODE {KLOUD

functions

functions

- self contained units of code which carry out a certain job.
- help us divide a program into small manageable, repeatable and organisable chunks.

functions



why use functions

- Reusability
- Abstraction

{KODE {KLOUD

functions

functions syntax

```
func <function_name> (<params>) <return type> {  
    // body of the function  
}
```

functions syntax

```
func addNumbers(a int, b int) int {
```

```
// body of the function
```

```
}
```


functions syntax



return keyword

```
func addNumbers(a int, b int) int {  
    // body of the function  
}
```

return keyword

```
func addNumbers(a int, b int) int {  
    sum := a + b  
    return sum  
}
```

calling a function

```
<function_name>(<argument(s)>)
```

```
addNumbers(2, 3)
```

```
sumOfNumbers := addNumbers(2, 3)
```

naming convention for functions

- must begin with a letter.
- can have any number of additional letters and symbols.
- cannot contain spaces.
- case-sensitive.

```
add_2
```

```
add_numbers
```

parameters vs arguments

- **Function parameters** are the names listed in the function's definition.
- **Function arguments** are the real values passed into the function.

```
func addNumbers(a int, b int) int {  
    sum := a + b  
    return sum  
}  
  
func main() {  
    sumOfNumbers := addNumbers(2, 3)  
    fmt.Print(sumOfNumbers)  
}
```

functions

```
package main
import "fmt"

func printGreeting(str string) {
    fmt.Println("Hey there,", str)
}

func main() {
    printGreeting("Priyanka")
}

>>> go run main.go

Hey there, Priyanka
```

{KODE {KLOUD

functions- return types

returning single value

```
package main
import "fmt"

func addNumbers(a int, b int) string {
    sum := a + b
    return sum
}

func main() {
    sumOfNumbers := addNumbers(2, 3)
    fmt.Print(sumOfNumbers)
}

>>> go run main.go

cannot use sum (type int) as type string in
return argument
```

returning multiple values

```
main.go

package main
import "fmt"

func operation(a int, b int) (int, int){
    sum := a + b
    diff := a - b
    return sum, diff
}

func main() {
    sum, difference := operation(20, 10)
    fmt.Println(sum, difference)
}

>>> go run main.go

30 10
```

named return values

```
main.go

package main
import "fmt"

func operation(a int, b int) (sum int, diff int){
    sum = a + b
    diff = a - b
    return
}

func main() {
    sum, difference := operation(20, 10)
    fmt.Println(sum, " ", difference)
}

>>> go run main.go

30 10
```

variadic functions

- function that accepts variable number of arguments.
- it is possible to pass a varying number of arguments of the same type as referenced in the function signature.
- to declare a variadic function, the type of the final parameter is preceded by an ellipsis "..."
- Example - `fmt.Println` method

variadic functions

```
func <func_name>(param-1 type, param-2 type, para-3 ...type) <return_type>
```

```
func sumNumbers(numbers ...int) int
```

```
func sumNumbers(str string, numbers ...int)
```

variadic functions

```
package main
import "fmt"

func sumNumbers(numbers ...int) int {
    sum := 0
    for _, value := range numbers {
        sum += value
    }
    return sum
}

func main() {
    fmt.Println(sumNumbers())
    fmt.Println(sumNumbers(10))
    fmt.Println(sumNumbers(10, 20))
    fmt.Println(sumNumbers(10, 20, 30, 40, 50))
}

>>> go run main.go

0
10
30
150
```

variadic functions

```
package main
import "fmt"

func printDetails(student string, subjects ...string) {
    fmt.Println("hey ", student, ", here are your subjects - ")
    for _, sub := range subjects {
        fmt.Printf("%s, ", sub)
    }
}

func main() {
    printDetails("Joe", "Physics", "Biology")
}

>>> go run main.go

hey Joe , here are your subjects -
Physics, Biology,
```


blank identifier



```
package main
import "fmt"
func f() (int, int) {
    return 42, 53
}

func main() {
    a, b := f()
    fmt.Println(a)
}
```

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

```
42
53
assignment mismatch: 1 variable but
f returns 2 values
```

{KODE {KLOUD

recursive functions

recursive functions

- **Recursion** is a concept where a function calls itself by direct or indirect means.
- the function keeps calling itself until it reaches a base case.
- used to solve a problem where the solution is dependent on the smaller instance of the same problem.

recursive functions

`factorial(5) = 5*4*3*2*1`

```
main.go

package main
import "fmt"

func factorial(n int) int {
    if n == 0 {
        return 1
    }
    return n * factorial(n-1)
}

func main() {
    n := 5
    result := factorial(n)
    fmt.Println("Factorial of", n, "is :", result)
}

>>> go run main.go
Factorial of 5 is : 120
```

recursive functions

```
factorial(5)    120
```



```
return 5 * factorial(4) = 120
```



```
return 4 * factorial(3) = 24
```



```
return 3 * factorial(2) = 6
```



```
return 2 * factorial(1) = 2
```



```
1
```

{KODE {KLOUD

anonymous functions

anonymous functions

- An anonymous function is a function that is declared without any named identifier to refer to it.
- They can accept inputs and return outputs, just as standard functions do.
- They can be used for containing functionality that need not be named and possibly for short-term use.

function inside function

```
package main
import "fmt"

func main() {
    x := func(l int, b int) int {
        return l * b
    }
    fmt.Printf("%T \n", x)
    fmt.Println(x(20, 30))
}
```

```
>>> go run main.go
func(int, int) int
600
```

function inside function

```
package main
import "fmt"

func main() {
    x := func(l int, b int) int {
        return l * b
    }(20, 30)
    fmt.Printf("%T \n", x)
    fmt.Println(x)
}
```

```
>>> go run main.go
int
600
```

{KODE {KLOUD

high order functions

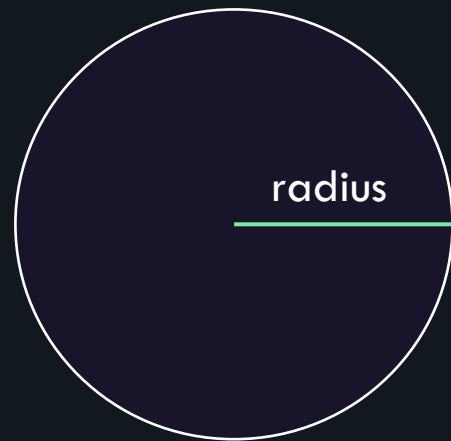
high order functions

- function that receives a function as an argument or returns a function as output.

why use high order functions

- Composition
 - creating smaller functions that take care of certain piece of logic.
 - composing complex function by using different smaller functions.
- Reduces bugs
- Code gets easier to read and understand

use case



Circle

1 - Area

2 - Perimeter

3 - Diameter

main.go

```
package main
import "fmt"


func calcArea(r float64) float64 {
    return 3.14 * r * r
}

func calcPerimeter(r float64) float64 {
    return 2 * 3.14 * r
}

func calcDiameter(r float64) float64 {
    return 2 * r
}
```

main.go

```
func main() {  
    var query int  
    var radius float64  
  
    fmt.Print("Enter the radius of the circle: ")  
    fmt.Scanf("%f", &radius)  
    fmt.Printf("Enter \n 1 - area \n 2 - perimeter \n 3 - diameter: ")  
    fmt.Scanf("%d", &query)  
  
    if query == 1 {  
        fmt.Println("Result: ", calcArea(radius))  
    } else if query == 2 {  
        fmt.Println("Result: ", calcPerimeter(radius))  
    } else if query == 3 {  
        fmt.Println("Result: ", calcDiameter(radius))  
    } else {  
        fmt.Println("Invalid query")  
    }  
}
```



```
main.go
```

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

```
Enter the radius of the circle: 9.1
```

```
Enter
```

```
1 - area
```

```
2 - perimeter
```

```
3 - diameter: 1
```

```
Result: 260.0234
```

```
Thank you!
```

main.go

```
func main() {  
    var query int  
    var radius float64  
  
    fmt.Print("Enter the radius of the circle: ")  
    fmt.Scanf("%f", &radius)  
    fmt.Printf("Enter \n 1 - area \n 2 - perimeter \n 3 - diameter: ")  
    fmt.Scanf("%d", &query)  
  
    if query == 1 {  
        fmt.Println("Result: ", calcArea(radius))  
    } else if query == 2 {  
        fmt.Println("Result: ", calcPerimeter(radius))  
    } else if query == 3 {  
        fmt.Println("Result: ", calcDiameter(radius))  
    } else {  
        fmt.Println("Invalid query")  
    }  
}
```

main.go

```
func printResult(radius float64, calcFunction func(r float64) float64) {  
    result := calcFunction(radius)  
    fmt.Println("Result: ", result)  
    fmt.Println("Thank you!")  
}  
  
func getFunction(query int) func(r float64) float64 {  
    query_to_func := map[int]func(r float64) float64{  
        1: calcArea,  
        2: calcPerimeter,  
        3: calcDiameter,  
    }  
    return query_to_func[query]  
}
```

main.go

```
func main() {  
    var query int  
    var radius float64  
  
    fmt.Print("Enter the radius of the circle: ")  
    fmt.Scanf("%f", &radius)  
    fmt.Printf("Enter \n 1 - area \n 2 - perimeter \n 3 - diameter: ")  
    fmt.Scanf("%d", &query)  
  
    if query == 1 {  
        result := calcArea(radius, getFunction(query))  
        fmt.Println("Result: ", calcArea(radius))  
    } else if query == 2 {  
        fmt.Println("Result: ", calcPerimeter(radius))  
    } else if query == 3 {  
        fmt.Println("Result: ", calcDiameter(radius))  
    } else {  
        fmt.Println("Invalid query")  
    }  
}
```



main.go

```
>>> go run main.go
Enter the radius of the circle: 7
Enter
 1 - area
 2 - perimeter
 3 - diameter: 3
Result: 14
Thank you!
```

{KODE {KLOUD

defer
statement

defer statement

- A defer statement delays the execution of a function until the surrounding function returns.
- The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

defer statement

```
package main
import "fmt"

func printName(str string) {
    fmt.Println(str)
}

func printRollNo(rno int) {
    fmt.Println(rno)
}

func printAddress(adr string) {
    fmt.Println(adr)
}

func main() {
    printName("Joe")
    defer printRollNo(23)
    printAddress("street-32")
}

>>> go run main.go

Joe
street-32
23
```

{KODE {KLOUD

pointers

pointers

```
x := 1
```

```
var ptr *int := &x
```

memory address	memory
0x0301	1
0x0302	
0x0303	
0x0304	0x0301
0x0305	
0x0306	

pointers

- A pointer is a variable that holds memory address of another variable.
- They point where the memory is allocated and provide ways to find or even change the value located at the memory location.

{KODE {KLOUD

address and dereference operators

address and dereference operators

- `& operator` - The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as `address-of operator`.
- `* operator` - It is known as the `dereference operator`. When placed before an address, it returns the value at that address.

address and dereference operators

```
x := 77
```



memory address	memory
0x0301	77
0x0302	

```
&x = 0x0301
```

```
*0x0301 = 77
```

address and dereference operators

```
main.go

package main
import "fmt"

func main() {
    i := 10
    fmt.Printf("%T %v \n", &i, &i)
    fmt.Printf("%T %v \n", *(&i), *(&i))
}

>>> go run main.go
*int 0xc00018c008
int 10
```

{KODE {KLOUD

declaring and initializing pointers

declaring a pointer

```
var <pointer_name> *<data_type>
```

```
var ptr_i *int
```

```
var ptr_s *string
```

declaring a pointer

```
package main
import "fmt"

func main() {
    var i *int
    var s *string
    fmt.Println(i)
    fmt.Println(s)
}

>>> go run main.go

<nil>
<nil>
```


initializing a pointer

```
var <pointer_name> *<data_type> = &<variable_name>
```

```
i := 10  
var ptr_i *int = &i
```

initializing a pointer

```
var <pointer_name> = &<variable_name>
```

```
s := "hello"  
var ptr_s = &s
```

initializing a pointer

```
<pointer_name> := &<variable_name>
```

```
s := "hello"  
ptr_s := &s
```

initializing a pointer

```
package main
import "fmt"

func main() {
    s := "hello"
    var b *string = &s
    fmt.Println(b)
    var a = &s
    fmt.Println(a)
    c := &s
    fmt.Println(c)
}
```

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

```
0xc000010230
```

```
0xc000010230
```

```
0xc000010230
```

{KODE {KLOUD

dereferencing
a pointer

dereferencing a pointer

```
*<pointer_name>
```

```
*<pointer_name> = <new_value>
```

dereferencing a pointer

```
x := 77
```

```
var ptr *int := &x
```

```
*ptr = 100
```

memory address	memory
0x0301	100
0x0302	
0x0303	
0x0304	0x0301
0x0305	
0x0306	

dereferencing a pointer

```
package main
import "fmt"

func main() {
    s := "hello"
    fmt.Printf("%T %v \n", s, s)
    ps := &s
    *ps = "world"
    fmt.Printf("%T %v \n", s, s)
    fmt.Printf("%T %v \n", ps, *ps)
}
```

```
>>> go run main.go

string hello
string world
*string world
```

{KODE {KLOUD

passing by
value

passing by value in functions

- Function is called by directly passing the value of the variable as an argument.
- the parameter is copied into another location of your memory.
- So, when accessing or modifying the variable within your function, only the copy is accessed or modified, and the original value is never modified.
- All basic types (int, float, bool, string, array) are passed by value.

passing by value in functions

```
func modify(a int) {  
    a += 100  
}  
  
func main() {  
    a := 10  
    fmt.Println(a)  
    modify(a)  
    fmt.Println(a)  
}
```

a

memory
address

0x0301

memory

10

0x0302

a

0x0303

100

0x0304

0x0305

0x0306

passing by value in functions

```
main.go

package main
import "fmt"

func modify(s string) {
    s = "world"
}

func main() {
    a := "hello"
    fmt.Println(a)
    modify(a)
    fmt.Println(a)
}

>>> go run main.go

hello
hello
```

{KODE {KLOUD

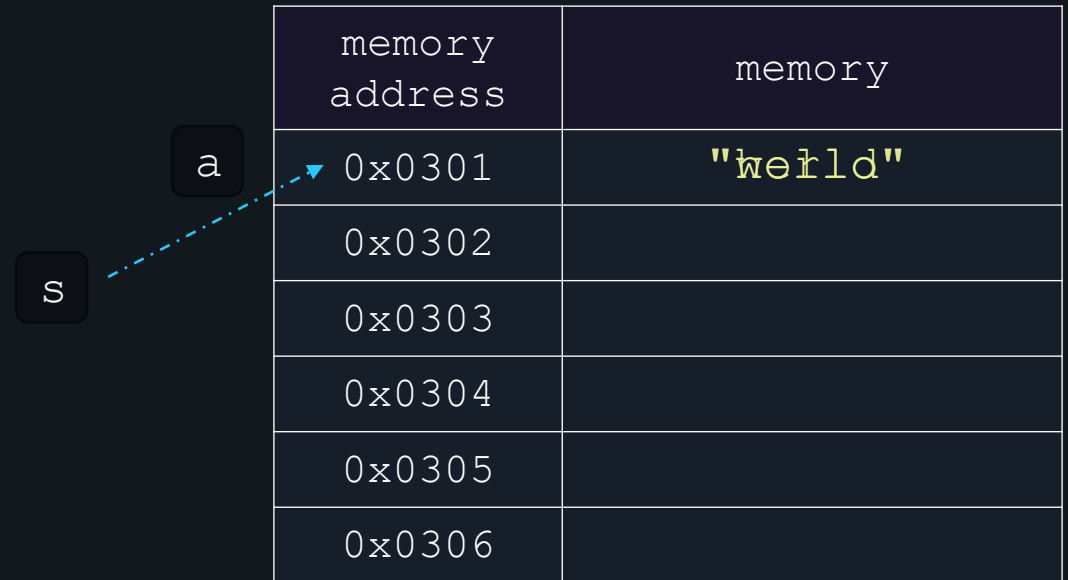
passing by
reference

passing by reference in functions

- Go supports pointers, allowing you to pass references to values within your program.
- In call by reference/pointer, the address of the variable is passed into the function call as the actual parameter.
- All the operations in the function are performed on the value stored at the address of the actual parameters.

passing by reference in functions

```
func modify(s *string) {  
    *s = "world"  
}  
  
func main() {  
    a := "hello"  
    fmt.Println(a)  
    modify(&a)  
    fmt.Println(a)  
}
```



passing by reference in functions

```
main.go

package main
import "fmt"

func modify(s *string) {
    *s = "world"
}

func main() {
    a := "hello"
    fmt.Println(a)
    modify(&a)
    fmt.Println(a)
}

>>> go run main.go

hello
world
```

passing by reference in functions

- Slices are passed by reference, by default.

```
main.go

package main
import "fmt"

func modify(s []int) {
    s[0] = 100
}

func main() {
    slice := []int{10, 20, 30}
    fmt.Println(slice)
    modify(slice)
    fmt.Println(slice)
}

>>> go run main.go

[10 20 30]
[100 20 30]
```

passing by reference in functions

- Maps, as well, are passed by reference, by default.

```
main.go

package main
import "fmt"

func modify(m map[string]int) {
    m["K"] = 75
}

func main() {
    ascii_codes := make(map[string]int)
    ascii_codes["A"] = 65
    ascii_codes["F"] = 70
    fmt.Println(ascii_codes)
    modify(ascii_codes)
    fmt.Println(ascii_codes)
}

>>> go run main.go

map[A:65 F:70]
map[A:65 F:70 K:75]
```

{KODE {KLOUD

Struct, Methods and Interfaces

{KODE {KLOUD

Struct - Introduction

struct

- user-defined data type.
- a structure that groups together data elements.
- provide a way to reference a series of grouped values through a single variable name.
- used when it makes sense to group or associate two or more data variables.

struct

name

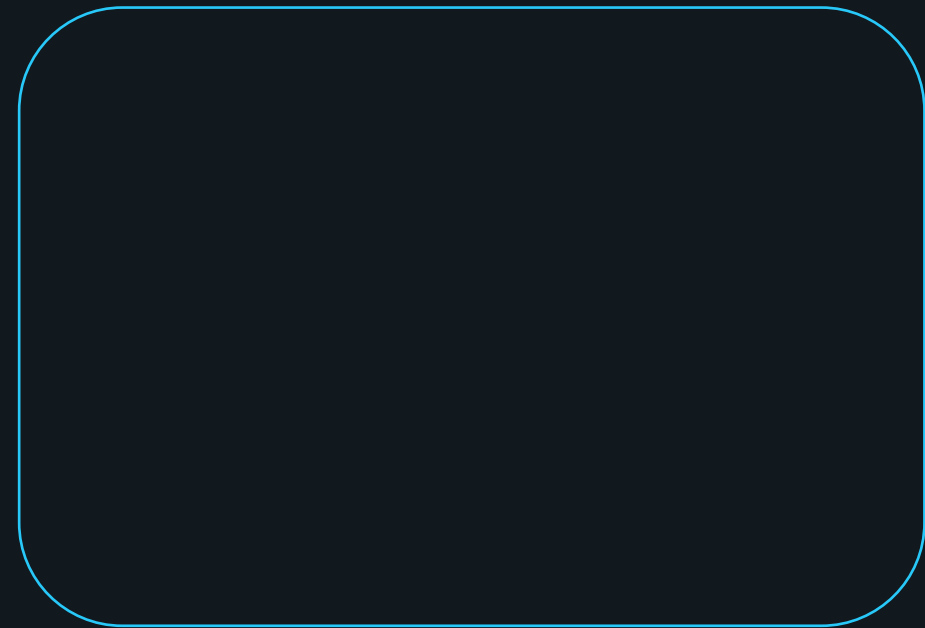
grades

rollNo

phoneNo

address

Student



{KODE {KLOUD

declaring and initialising a struct

struct - declaration

```
type <struct_name> struct {  
    // list of fields  
}
```

```
type Circle struct {  
    x float64  
    y float64  
    r float64  
}
```

struct - declaration

```
type Student struct {  
    name string  
    rollNo int  
    marks []int  
    grades map[string]int  
}
```

struct - initialization

```
var <variable_name> <struct_name>
```

```
var s Student
```


struct - initialization

```
package main
import "fmt"

type Student struct {
    name string
    rollNo int
    marks []int
    grades map[string]int
}

func main() {
    var s Student
    fmt.Printf("%+v", s)
}

>>> go run main.go
{name: rollNo:0 marks:[] grades:map[]}
```

struct - initialization

```
<variable_name> := new(<struct_name>)
```

```
st := new(Student)
```

struct - initialization

```
main.go

package main
import "fmt"

type Student struct {
    name string
    rollNo int
    marks []int
    grades map[string]int
}

func main() {
    st := new(Student)
    fmt.Printf("%+v", st)
}

>>> go run main.go
    &{name: rollNo:0 marks:[] grades:map[]}
```

struct - initialization

```
<variable_name> := <struct_name> {  
    <field_name>: <value>,  
    <field_name>: <value>,  
}
```

```
st := Student{  
    name: "Joe",  
    rollNo: 12,  
}
```

struct - initialization

```
package main
import "fmt"

type Student struct {
    name string
    rollNo int
}

func main() {
    st := Student{
        name: "Joe",
        rollNo: 12,
    }
    fmt.Printf("%+v", st)
}

>>> go run main.go

{name:Joe rollNo:12}
```

struct - initialization

```
main.go

package main
import "fmt"

type Student struct {
    name string
    rollNo int
}

func main() {
    st := Student{"Joe", 12}
    fmt.Printf("%+v", st)
}

>>> go run main.go

{name:Joe rollNo:12}
```

{KODE {KLOUD

accessing struct fields

struct - accessing fields

```
<variable_name>.<field_name>
```

struct - accessing fields

```
package main
import "fmt"

type Circle struct {
    x int
    y int
    radius int
}

func main() {
    var c Circle
    c.x = 5
    c.y = 5
    c.radius = 5
    fmt.Printf("%+v \n", c)
}

>>> go run main.go

{x:5 y:5 radius:5}
```

struct - accessing fields

```
main.go

package main
import "fmt"

type Circle struct {
    x int
    y int
    radius int
}

func main() {
    var c Circle
    c.x = 5
    c.y = 5
    c.radius = 5
    fmt.Printf("%+v \n", c)
    fmt.Printf("%+v \n", c.area)
}

>>> go run main.go

c.area undefined (type Circle has no field or method area)
```

{KODE {KLOUD

passing structs to functions

passing structs to functions

```
package main
import "fmt"

type Circle struct {
    x int
    y int
    radius float64
    area float64
}

func calcArea(c Circle) {
    const PI float64 = 3.14
    var area float64
    area = (PI * c.radius * c.radius)
    c.area = area
}
```

passing structs to functions

```
main.go

func main() {
    c := Circle{x: 5, y: 5, radius: 5, area: 0}
    fmt.Printf("%+v \n", c)
    calcArea(c)
    fmt.Printf("%+v \n", c)
}

>>> go run main.go

{x:5 y:5 radius:5 area:0}
{x:5 y:5 radius:5 area:0}
```

passing structs to functions

```
package main
import "fmt"

type Circle struct {
    x int
    y int
    radius float64
    area float64
}

func calcArea(c *Circle) {
    const PI float64 = 3.14
    var area float64
    area = (PI * c.radius * c.radius)
    (*c).area = area
}
```


passing structs to functions



main.go

```
func main() {  
    c := Circle{x: 5, y: 5, radius: 5, area: 0}  
    fmt.Printf("%+v \n", c)  
    calcArea(&c)  
    fmt.Printf("%+v \n", c)  
}
```

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

```
{x:5 y:5 radius:5 area:0}  
{x:5 y:5 radius:5 area:78.5}
```

{KODE {KLOUD

comparing structs

comparing structs

- Structs of the same type can be compared using Go's equality operators.

==

!=

comparing structs

```
package main
import "fmt"

type s1 struct {
    x int
}

type s2 struct {
    x int
}

func main() {
    c := s1{x: 5}
    c1 := s2{x: 5}
    if c == c1 {
        fmt.Println("yes")
    }
}
```

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

```
invalid operation: c == c1 (mismatched types s1 and s2)
```

comparing structs

```
package main
import "fmt"

type s1 struct {
    x int
}

func main() {
    c := s1{x: 5}
    c1 := s1{x: 6}
    c2 := s1{x: 5}
    if c != c1 {
        fmt.Println("c and c1 have different values")
    }
    if c == c2 {
        fmt.Println("c is same as c2")
    }
}

>>> go run main.go

c and c1 have different values
c is same as c2
```

{KODE {KLOUD

Methods

Methods

- A method augments a function by adding an extra parameter section immediately after the `func` keyword that accepts a single argument.
- This argument is called a `receiver`.
- A method is a function that has a defined receiver.

```
func (<receiver>) <method_name>(<parameters>)  
<return_params> {  
  
    //code  
  
}
```

Methods

```
func (c Circle) area() float64 {  
  
    //code  
  
}
```

```
func (c *Circle) area() float64 {  
  
    //code  
  
}
```

Methods

```
package main
import "fmt"

type Circle struct {
    radius float64
    area float64
}

func (c *Circle) calcArea() {
    c.area = 3.14 * c.radius * c.radius
}

func main() {
    c := Circle{radius: 5}
    c.calcArea()
    fmt.Printf("%+v", c)
}

>>> go run main.go
{radius:5 area:78.5}
```

Methods

```
package main
import "fmt"

type Circle struct {
    radius float64
    area float64
}

func (c Circle) calcArea() {
    c.area = 3.14 * c.radius * c.radius
}

func main() {
    c := Circle{radius: 5}
    c.calcArea()
    fmt.Printf("%+v", c)
}

>>> go run main.go
{radius:5 area:0}
```

{KODE {KLOUD

Method Sets

Method sets

- set of methods that are available to a data type.
- useful way to encapsulate functionality.

Method sets

```
main.go

package main
import "fmt"

type Student struct {
    name string
    grades []int
}

func (s *Student) displayName() {
    fmt.Println(s.name)
}

func (s *Student) calculatePercentage() float64 {
    sum := 0
    for _, v := range s.grades {
        sum += v
    }
    return float64(sum*100) / float64(len(s.grades)*100)
}
```


Method sets



main.go

```
func main() {  
    s := Student{name: "Joe", grades: []int{90, 75, 80}}  
    s.displayName()  
    fmt.Printf("%.2f%%", s.calculatePercentage())  
}
```

```
>>> go run main.go  
Joe  
81.67%
```

{KODE {KLOUD

Interfaces

interfaces

- An interface specifies a method set and is a powerful way to introduce modularity in Go.
- Interface is like a blueprint for a method set.
- They describe all the methods of a method set by providing the function signature for each method.
- They specify a set of methods, but do not implement them.

interfaces - syntax

```
type <interface_name> interface {  
    // Method signatures  
}
```

```
type FixedDeposit interface {  
    getRateOfInterest() float64  
    calcReturn() float64  
}
```

implementing an interface

- A type implements an interface by implementing its methods.
- The go language interfaces are implemented implicitly.
- And it does not have any specific keyword to implement an interface.

{KODE {KLOUD

implementing interfaces

Method sets

```
package main
import "fmt"

type shape interface {
    area() float64
    perimeter() float64
}

type square struct {
    side float64
}

func (s square) area() float64 {
    return s.side * s.side
}

func (s square) perimeter() float64 {
    return 4 * s.side
}
```

Method sets

```
main.go

type rect struct {
    length, breadth float64
}

func (r rect) area() float64 {
    return r.length * r.breadth
}

func (r rect) perimeter() float64 {
    return 2*r.length + 2*r.breadth
}
```

Method sets

```
main.go

func printData(s shape) {
    fmt.Println(s)
    fmt.Println(s.area())
    fmt.Println(s.perimeter())
}

func main() {
    r := rect{length: 3, breadth: 4}
    c := square{side: 5}
    printData(r)
    printData(c)
}

>>> go run main.go

{3 4}
12
14
{5}
25
20
```

{KODE {KLOUD