SwiftUI is a apple framework for developing app

String:
var sami = "Onnk xoss. He is known as \"VP of NSU HR Club\" in NSU."
var multiline = """
bla bla
Bla bla bla
"""
to count the characters of string use sami.count

imojis have multiple internal characters.

To read some data don't need any parentheses
To do some work, we need parenthesis. We can also say we need function.

.hasprefix("A day") .hassuffix(".jpg")

For million we can use int.
To write billions, trillions, quadrillions, quintillions both positive and negative.
We can use underscore for example: 1_000_000 = 1000000
20.isMultiple(of: 5)

We cannot add double and int number.
CG float.

Toggle on boolean is same as !sth but useful

Concatenation
Swift add like "1" + "2" + "3" + ... + "n" by n times.
Like, "12" + "3" + ... + "n", "123" + ... + "n", ..., "123...n".

Swift introduced interpolation

Array
You can modify array after creating.
arrName.append("newElement1")
You can add two array just using plus sign +
Declare:

```
var scores = Array<Int>()
scores.append(32)
```

We can also use, var albums = [String]()
to search we can use
print(arrayName.contains("sth"))
Empty array:

```
var terms: [String] = [String]()
var Cities: [String] = []
var clues = [String]()
```

`count, append() and contains()`.

SET
Good for fixed data, it automatically remove duplicate data. We insert items not append like arrays. It keeps data in highly optimized order that make very fast to locate the data.

It used specially for **fast lookup** of items.
TIP: Alongside contains(), you'll also find count to read the number of items in a set, and sorted() to return a sorted array containing the set's items.

enum: Numeration
when we have many cases
0 means first case, 1 means second case

when should I use type annotations in Swift?
1 Swift can't figure out what type should be used.
2 You want Swift to use a different type from its default.
3 You don't want to assign a value just yet

Array: Arrays let us store lots of values in one place, then read them out using integer indices. one specific type

Dictionary: Dictionaries also let us store lots of values in one place, but let read them out using keys we specify. one specific type for key and anothe for the value.
Set: • Sets are a *third* way of storing lots of values in one place, but w don't get to choose the order in which they store those items. Sets are rea efficient at finding whether they contain a specific item.
Enums: • Enums let us create our own simple types in Swift so that can specify a range of acceptable values such as a list of actions the user can perform, the types of files we are able to write, or the types of notification to send to the user. Actually not store lots of data.

Condition
if anyCondition {
    Statement
}
Mutually exclusive
Multiple condition

```
if a {
    print("Code to run if a is true.")
} else if b {
    print("Code to run if a is false but b is true.")
} else {
    print("Code to run if both a and b are false.")
```

Switch case:
1. checking twice 2. Missing functionality 3. Repeated writing
if you explicitly *want* Swift to carry on executing subsequent cases, use **`fallthrough`**. This is *not* commonly used, but sometimes – just sometimes – it can help you avoid repeating work.

Loop
- We call the code inside the braces the *loop body*
- We call one cycle through the loop body a *loop iteration*.
- We call **`os`** the *loop variable*. This exists only inside the loop body, and will change to a new value in the next loop iteration. Here os is like i. But here ar

`x...y`

`for` loops are more common when you have a finite amount data to go through, such as a range or an array,
but **`while`** loops are really helpful when you need a custom condition.

us
r
e
lly
we
r[l]
of

This means we can repeat the same code until…

- …the user asks us to stop
- …a server tell us to stop
- …we've found the answer we're looking for
- …we've generated enough data

Function

- same functionality in many places
- breaking up code
- The last reason is more advanced: Swift lets us build new functions out of existing functions, which is a technique called *function composition*. By splitting your work into multiple small functions, function composition lets us build big functions by combining those small functions in various ways, a bit like Lego bricks.

In cocoa, there are 10 thousand functions.

```
func greet(name: String) -> String {
    name == "Taylor Swift" ? "Oh wow!" : "Hello, \(name)"
}
```

Multiple value return korte chaile Array return korte home
Destructuring

```
let (firstName, lastName) = getUser2()
```

• If you want to store a list of all words in a dictionary for a game, that has no duplicates and the order doesn't matter so you would go for a set.

• If you want to store all the articles read by a user, you would use a set if the order didn't matter (if all you cared about was whether they had read it or not), or use an array if the order *did* matter.

•\ If you want to store a list of high scores for a video game, that has an order that matters and might contain duplicates (if two players get the same score), so you'd use an array.

- If you want to store items for a todo list, that works best when the order is predictable so you should use an array.
- If you want to hold precisely two strings, or precisely two strings and an integer, or precisely three Booleans, or similar, you should use a tuple.

Error Catch
Builtin **error.localizedDescription**

We've covered a lot about functions in the previous chapters, so let's recap:

- Functions let us reuse code easily by carving off chunks of code and giving it a name.
- All functions start with the word **`func`**, followed by the function's name. The function's body is contained inside opening and closing braces.
- We can add parameters to make our functions more flexible – list them out one by one separated by commas: the name of the parameter, then a colon, then the type of the parameter.
- You can control how those parameter names are used externally, either by using a custom external parameter name or by using an underscore to disable the external name for that parameter.
- If you think there are certain parameter values you'll use repeatedly, you can make them have a default value so your function takes less code to write and does the smart thing by default.
- Functions can return a value if you want, but if you want to return multiple pieces of data from a function you should use a *tuple*. These hold several named elements, but it's *limited* in a way a dictionary is not – you list each element specifically, along with its type.
- Functions can throw errors: you create an enum defining the errors you want to happen, throw those errors inside the function as needed, then use **`do`**, **`try`**, and **`catch`** to handle them at the call site.

Closures

It starts by using **in**
But what if you wanted to *skip* creating a separate function, and just assign the functionality directly to a constant or variable?

Swift gives this the grandiose name *closure expression*, which is a fancy way of saying we just created a closure – a chunk of code we can pass around and call whenever we want. This one doesn't have a name, but otherwise it's effectively a function that takes no parameters and doesn't return a value.

```
let sayHello = { (name: String) -> String in
    "Hi \(name)!"
}
```

Correct! Closures cannot use external parameter labels.

trailing closures and shorthand syntax

Like I said, you're going to be using closures a *lot* with SwiftUI:

1 When you create a list of data on the screen, SwiftUI will ask you to provide a function that accepts one item from the list and converts it something it can display on-screen.
2 When you create a button, SwiftUI will ask you to provide one function to execute when the button is pressed, and another to generate the contents of the button – a picture, or some text, and so on.
3 Even just putting stacking pieces of text vertically is done using a closure.

Yes, you can create individual functions every time SwiftUI does this, but trust me: you won't. Closures make this kind of code completely natural, and I think you'll be amazed at how SwiftUI uses them to produce remarkably simple, clean code.
We've covered a lot about closures in the previous chapters, so let's recap:

- You can copy functions in Swift, and they work the same as the original except they lose their external parameter names.
- All functions have types, just like other data types. This includes the parameters they receive along with their return type, which might be **`Void`** – also known as "nothing".
- You can create closures directly by assigning to a constant or variable.
- Closures that accept parameters or return a value must declare this *inside* their braces, followed by the keyword **`in`**.
- Functions are able to accept other functions as parameters. They must declare up front exactly what data those functions must use, and Swift will ensure the rules are followed.
- In this situation, instead of passing a dedicated function you can also pass a closure – you can make one directly. Swift allows both approaches to work.
- When passing a closure as a function parameter, you don't need to explicitly write out the types inside your closure if Swift can figure it out automatically. The same is true for the return value – if Swift can figure it out, you don't need to specify it.
- If one or more of a function's final parameters are functions, you can use trailing closure syntax.
- You can also use shorthand parameter names such as **`$0`** and **`$1`**, but I would recommend doing that only under some conditions.
- You can make your own functions that accept functions as parameters, although in practice it's much more important to know how to *use* them than how to *create* them.

Of all the various parts of the Swift language, I'd say closures are the single toughest thing to learn. Not only is the syntax a little hard on your eyes at first, but the very concept of passing a function into a function takes a little time to sink in.

Struct

Struct is a constant type. To change data we use **muta**

- use tuples when you want to return two or more arbit
  from a function, but prefer structs when you have so
  to send or receive multiple times.

Struct have two type property. (i) stored property (ii) C
Computed properties must always have an explicit type.

Access control:

• Use **`private`** for "don't let anything outside the struct u
• Use **`fileprivate`** for "don't let anything outside the cu
• Use **`public`** for "let anyone, anywhere use this."
• **`private(set)`**.let anyone read this property, but on
it.
• **Static**

If you use **`private`** access control for one or more pro
you'll need to create your own initializer.

If you want to mix and match static and non-static prop
there are two rules:
1. To access non-static code from static code… you're
properties and methods can't refer to non-static prope
because it just doesn't make sense – which instance o
be referring to?
2. To access static code from non-static code, always
such as **`School.studentCount`**. You can also use **Se**
current type.

Structs are used almost everywhere in Swift: **`String`**,
and even **`Bool`** are all implemented as structs, and no
that a function such as **`isMultiple(of:)`** is really a
`Int`.
Let's recap what else we learned:
• You can create your own structs by writing **`struct`**, g
placing the struct's code inside braces.
• Structs can have variable and constants (known as p
functions (known as methods)
• If a method tries to modify properties of its struct, yo
as **`mutating`**.
• You can store properties in memory, or create compu
calculate a value every time they are accessed.
• We can attach **`didSet`** and **`willSet`** property obser
inside a struct, which is helpful when we need to be su
always executed when the property changes.
• Initializers are a bit like specialized functions, and Sw
all structs using their property names.
• You can create your own custom initializers if you wa
always make sure all properties in your struct have a v
initializer finishes, and before you call any other metho
• We can use access to mark any properties and meth
or unavailable externally, as needed.
• It's possible to attach a property or methods directly
can use them without creating an instance of the struc

ting func.

trary pieces of values
me fixed data you want

omputed property

se this.”
rrent file use this.”

ly let my methods write

perties, chances are

perties and methods,

out of luck: static
rties and methods
f **School** would you

use your type’s name
lf to refer to the

Int, **Double**, **Array**
w you can recognize
method belonging to

giving it a name, then

roperties) and

u must mark it

ted properties that

vers to properties
re that some code is

ift generates one for

nt, but you must
alue by the time the
ds.
ods as being available

to a struct, so you
t.

Class

Similarities:

- You get to create and name them
- Add properties, methods, property observer and access control
- You can create custom initializers to configure new instances

Differences

1. You can make one class build upon functionality in another class (a process called **Inheritance**)
2. Swift won't generate a member-wise initializer for classes.(Either you have your own custom initializers or provide your properties with different values)
3. If you copy a initializer of a class, both copy shares the same data. (Shallow copy)
4. We can add a deinitializer to run when the final copy is destroyed. (Free up any resources we'd allocated)
5. Constant class instances can have their variable properties changed

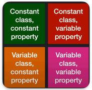
In UIKIT it is common to have long class hierarchies.

Deinitializer:

- Don't use func
- Can never take parameters or return data
- It'll Automatically call when the last copy of a class instances is destroyed.
- We never call deinitializer directly
- Struct don't have it.

Mainly scope theke ber hoilei dead oi instance.

Signpost that always points to the same user, who always has the same name.
Signpost that always points to the same user, But their name can change.
Signpost that can point to different users, But their names never change.
Signpost that can point to different users, and those users can also change their names.



Constant class, constant property	Constant class, variable property
Variable class, constant property	Variable class, variable property

• Classes have lots of things in common with structs, including the ability to have properties and methods, but there are five key differences between classes and structs.
• First, classes can inherit from other classes, which means they get access to the properties and methods of their parent class. You can optionally override methods in child classes if you want, or mark a class as being **final** to stop others subclassing it.
• Second, Swift doesn’t generate a memberwise initializer for classes, so you need to do it yourself. If a subclass has its own initializer, it must always call the parent class’s initializer at some point.
• Third, if you create a class instance then take copies of it, all those copies point back to the same instance. This means changing some data in one of the copies changes them all.
• Fourth, classes can have deinitializers that run when the last copy of one instance is destroyed.
• Finally, variable properties inside class instances can be changed regardless of whether the instance itself was created as variable.

Protocol

Protocol is bit like contracts for swift code. They let us define what kind of functionality we expect our types of support and swift will ensure those types add the required functionality that follow the rules.

Protocol is a list of bare requirements you must do at least these things.

Opaque return type

Swift provide really obscure, really complex, really important feature.
Both int and bool conform from a common swift protocol called equitable.
That means they can be compared for equality.

Extension

Already into string extension in the example.

Protocols let us define how structs, classes, and enums ought to work: what methods they should have, and what properties they should have. Swift will enforce these rules for us, so that when we say a type conforms to a protocol Swift will make sure it has all the methods and properties required by that protocol.

Extensions let us add functionality to classes, structs, and more, which is helpful for modifying types we don’t own – types that were written by Apple or someone else, for example. Methods added using extensions are indistinguishable from methods that were originally part of the type, but there is a difference for properties: extensions may not add new stored properties, only computed properties.
Extensions are also useful for organizing our *own* code, and although there are several ways of doing this I want to focus on two here: conformance grouping and purpose grouping.

protocol-oriented programming language.

Of course, the Swift developers don’t want to write this same code again and again, so they used a protocol extension: they wrote a single **`allSatisfy()`** method that works on a protocol called **Sequence**, which all arrays, sets, and dictionaries conform to. This meant the **`allSatisfy()`** method immediately became available on all those types, sharing exactly the same code.

• Protocols are like contracts for code: we specify the functions and methods that we required, and conforming types must implement them.
• Opaque return types let us hide some information in our code. That might mean we want to retain flexibility to change in the future, but also means we don’t need to write out gigantic return types.
• Extensions let us add functionality to our own custom types, or to Swift’s built-in types. This might mean adding a method, but we can also add computed properties.
• Protocol extensions let us add functionality to many types all at once – we can add properties and methods to a protocol, and all conforming types get access to them.

Optional

Swift likes to be predictable.
Any data type can be optional in Swift:
• An integer might be 0, -1, 500, or any other range o
• An *optional* integer might be all the regular integer might not exist.
• A string might be “Hello”, it might be the complete be “” – an empty string.
• An *optional* string might be any regular string value
• A custom **User** struct could contain all sorts of pro
• An *optional* **User** struct could contain all those san

```
var number: Int? = nil//optional In
anything.

//unwrapped data/optional data must
if let unwrappedNumber = number {
    print(square(number: unwrappedN
}
```

Guard

`guard` is designed exactly for this style of programm help:

1 If you use **guard** to check a function’s i require you to use **return** if the check fails.
2 If the check passes and the optional yo you can use it *after* the **guard** code finishes.

```
let lightsaberColor: String? = "green"
let color = lightsaberColor ?? "blue"
```

Nil coalescing

Nil coalescing lets us attempt to unwrap an optional, optional contains nil. This is extraordinarily useful in are a great feature it’s usually better to have a non-c rather than a “might be a string, might be nil” – and n that.

```
let savedData = loadSavedMessage() ?? ""
let savedData = first() ?? second() ?? ""
```

Optional Training

"If the optional has a value inside, unwrap it then...."

You’ll find `try?` is mainly used in three places:

1 In combination with `guard let` to exit returns `nil`.
2 In combination with nil coalescing to atter value on failure.
3 When calling any throwing function witho genuinely don’t care if it succeeded or not – maybe you analytics to a server, for example.

of numbers.
 values, but also might be **nil** – it

works of Shakespeare, or it might

e, but also might be **nil**.
perties that describe a user.
ne properties, or not exist at all.

```
t. it could be
```

```
 use if let
```

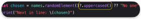
```
umber))
```

hing, and in fact does two things to

nputs are valid, Swift will always

u're unwrapping has a value inside,

, but provide a default value if the
Swift, because although optionals
optional – to have a real string
nil coalescing is a great way to get



```
et chosen = names.randomElement()?.uppercased() ?? "No one"
rint("Next in line: \(chosen)")
```

the current function if the `try?` call

mpt something or provide a default

out a return value, when you
're writing to a log file or sending

Let's recap what we learned:

- Optionals let us represent the absence of data, which means we're able to say "this integer has no value" – that's different from a fixed number such as 0.
- As a result, everything that *isn't* optional definitely has a value inside, even if that's just an empty string.
- Unwrapping an optional is the process of looking inside a box to see what it contains: if there's a value inside it's sent back for use, otherwise there will be `nil` inside.
- We can use `if let` to run some code if the optional has a value, or `guard let` to run some code if the optional *doesn't* have a value – but with `guard` we must always exit the function afterwards.
- The nil coalescing operator, `??`, unwraps and returns an optional's value, or uses a default value instead.
- Optional chaining lets us read an optional inside another optional with a convenient syntax.
- If a function might throw errors, you can convert it into an optional using `try?` – you'll either get back the function's return value, or `nil` if an error is thrown.

Optionals are second only to closures when it comes to language features folks struggle to learn, but I promise after a few months you'll wonder how you could live without them!