

SWIFT INTRO : STRUCT & METHOD

✓ Struct

- Format

```
struct Sport {  
    var name: String  
}
```

- Retrieve data

```
var tennis = Sport(name: "Tennis")  
print(tennis.name)
```

- Modify data

```
tennis.name = "Lawn tennis"
```

- Tuple vs Struct :

A tuple is effectively just a struct without a name, like an anonymous struct.

However, tuples have a problem: they are great for one-off use, but they can be annoying to use again and again

Ex:

using struct

```
func authenticate(_ user: User) { ... }  
func showProfile(for user: User) { ... }  
func signOut(_ user: User) { ... }
```

using tuple

```
func authenticate(_ user: (name: String, age: Int, city: String)) { ... }
func showProfile(for user: (name: String, age: Int, city: String)) { ... }
func signOut(_ user: (name: String, age: Int, city: String)) { ... }
```

✓ Struct properties:

- 2 type of properties:
 - + stored property
 - + computed property : a property that runs code to figure out its value.

```
struct Sport {
    var name: String
    var isOlympicSport: Bool
    var olympicStatus: String {
        if isOlympicSport {
            return "\(name) is an Olympic sport"
        } else {
            return "\(name) is not an Olympic sport"
        }
    }
}
```

← stored properties

← computed property

```
let chessBoxing = Sport(name: "Chessboxing", isOlympicSport: false)
print(chessBoxing.olympicStatus)
```

stored property	computed property
where a value is stashed away in some memory to be used later	where a value is recomputed every time it's called

	it effectively just a function call that happens to belong to your struct
USAGE	USAGE
if you regularly read the property when its value hasn't changed, then using a stored property will be much faster	if your property is read very rarely, then using a computed property saves you from having to calculate its value and store it somewhere
	use computed property when your property's value relies on the values of your other properties

– Property observer (**didSet**)

Property observers let you run code before or after any property changes.

Ex:

Print a message every time **amount** changes, and we can use a **didSet** property observer for that.

```
struct Progress {
    var task: String
    var amount: Int {
        didSet {
            print("\(task) is now \(amount)% complete")
        }
    }
}
```

NOTE: You can also use **willSet** to take action *before* a property changes, but that is rarely used.

✓ Methods

Functions inside structs are called ***methods***. Using keyword **func**

```
struct City {  
    var population: Int  
  
    func collectTaxes() -> Int {  
        return population * 1000  
    }  
}
```

```
let london = City(population: 9_000_000)  
london.collectTaxes()
```

– Mutating method:

When you *want* to change a property inside a method, you need to mark it using the **mutating** keyword, like this:

```
struct Person {  
    var name: String  
  
    mutating func makeAnonymous() {  
        name = "Anonymous"  
    }  
}
```

NOTE: A method that is *not* marked as mutating cannot call a mutating function – you must mark them both as mutating.

– **String properties and methods:**

Strings are structs – they have their own methods and properties we can use to query and manipulate the string.

Ex:

String properties such as count,...

String methods such as String.uppercased(), String.sorted(),...

****Almost all of Swift's core types are implemented as structs, including strings, integers, arrays, dictionaries, and even Booleans**

– **Array properties and methods:**

Arrays are also structs, which means they too have their own methods and properties we can use to query and manipulate the array.

Ex:

Array properties such as count,...

Array methods such as Array.append(), Array.firstIndex() ,
Array.sorted(),...

****Swift lets us read array values using myArray[3], we can't do the same with strings – myString[3] is invalid.**

✓ Initializers

By default, we will write after defining struct

```
var user = User(username: "twostraws")
```

We can provide our own initializer to replace the default one.

```
struct User {  
    var username: String  
  
    init() {  
        username = "Anonymous"  
        print("Creating a new user!")  
    }  
}
```

```
var user = User()  
user.username = "twostraws"
```

You *don't* write **func** before initializers, but you *do* need to make sure all properties have a value before the initializer ends.

- Swift give us 2 type of initializer :
 1. default memberwise initializer (WITHOUT init())

```
struct Employee {  
    var name: String  
    var yearsActive = 0  
}  
  
let roslin = Employee(name: "Laura Roslin")  
let adama = Employee(name: "William Adama", yearsActive: 45)
```

2. Custom initializer (init())

With this, we are no longer allowed to use default memberwise initializer

```
struct Employee {  
    var name: String  
    var yearsActive = 0  
  
    init() {  
        self.name = "Anonymous"  
        print("Creating an anonymous employee...")  
    }  
}
```

NOTE: **We can have both custom initializer & default memberwise initializer. by putting init() to extension of the struct

```

struct Employee {
    var name: String
    var yearsActive = 0
}
    put init() in struct's extension to be able to use both initializer
    ↙
extension Employee {
    init() {
        self.name = "Anonymous"
        print("Creating an anonymous employee...")
    }
}

// creating a named employee now works
let roslin = Employee(name: "Laura Roslin")
                Default memberwise initializer

// as does creating an anonymous employee
let anon = Employee() Custom initializer

```

✓ Referring to current instance (SELF)

Inside methods you get a special constant called **self**, which points to whatever instance of the struct is currently being used.


```
struct Student {  
    var name: String  
    var bestFriend: String  
  
    init(name: String, bestFriend: String) {  
        print("Enrolling \(name) in class...")  
        self.name = name  
        self.bestFriend = bestFriend  
    }  
}
```



Lazy property

We add the **lazy** keyword to struct's property Swift will only create the lazy_property_ when it's first accessed

```

struct Person {
    var name: String
    var familyTree = FamilyTree()
    init(name: String) {
        self.name = name
    }
}

var ed = Person(name: "Ed")

```

we can define familyTree as lazy property :
 lazy var familyTree = FamilyTree()
 so that Swift only create it if we call it for the 1st time

Pros:

- lazy properties let us delay the creation of a property until it's actually used (like a computed property)
- *Unlike* a computed property they store the result that gets calculated, so that subsequent accesses to the property don't redo the work

=> provide less performance when they aren't used (because their code is never run)

=> provide extra performance when they are used repeatedly (because their value is cached.)

Cons:

- lazy properties can accidentally produce work where you don't expect it

Ex: Building a game and access a complex lazy property for the 1st time
 => causing game to slow down. So, it's much better to do slow work up front and get it out of the way.

- Lazy properties always store their result, which might either be

unnecessary (because you aren't going to use it again) or be pointless (because it needs to be recalculated frequently)

- lazy properties change the underlying object they are attached to, you can't use them on constant structs

–


✓ Static properties & methods

To share specific properties and methods across all instances of the struct by declaring them as *static*.

Ex:

Each time we create a new student, we'll add one to classSize

```
struct Student {  
    static var classSize = 0  
    var name: String  
  
    init(name: String) {  
        self.name = name  
        Student.classSize += 1  
    }  
}
```



** Because the **classSize** property belongs to the struct itself rather than instances of the struct, we need to read it using **Student.classSize**:

```
print(Student.classSize)
```

Usage:

- One common use for static properties and methods is to store common functionality you use across an entire app

EX: use both a static property and a static method to store some random entropy in the same struct, like this:

```
static var entropy = Int.random(in: 1...1000)

static func getEntropy() -> Int {
    entropy += 1
    return entropy
}
```

✓ Access control (Private / Public)

Access control lets you restrict which code can use properties and methods using **private**

EX: we can make their **id** be private so you can't read it from outside the struct – trying to write **ed.id** simply won't work.

```
struct Person {
    private var id: String

    init(id: String) {
        self.id = id
    }

    func identify() -> String {
        return "My social security number is \(id)"
    }
}
```

****Only methods inside **Person** can read the **id** property**
 Another common option is **public**, which lets all other code use the property or method.