Equable Protocol

For a struct all of its stored properties conform to Equatable
For an Enum all of its associated values conform to Equatable

For anything else, we implement the equal-to operator (==) as a static method of our type.

```
let yourName = "Paul"
let myName = "Gwinyai"

if yourName == myName {

}
```

```
struct User : Equatable {
    var id:Int
    var name: String
}
let user1 = User(id:1, name: "Gwinyai")
let user2 = User(id:2, name: "Paul")
if user1 == user2 {
    print
}
```

```
class UserModel{
    var id: Int
    var name: String
    init (id: String, name: String) {
        self.id = id
        self.name = name
    }
}
extension UserModel: Equatable{
    static func == (lhs: UserModel, rhs: UserModel) -> Bool{
        return lhs.id == rhs.id
    }
}
```

* creat a static method to make it equable

⇒

```
let userModel1 = UserMode(id:1, name "Gwinyai")
let userModel2 = UserModel(id:2, name: "Paul")
if userModel1 == userModel2{
    print ("the user models are the same")
} else {
    print ("the user Models are different")
}
```

Comparable protocol (will also have to conform to equatable)

⇒ Any type conforms to Comparable can be compared using the relational operators (<) (>) (<=) and (>=)
⇒ must implement the < as a static method
must implement the == as a static method

```
import Foundation

struct User{
    var id:Int
    var name: String
    var registrationDate : Date
}

extension User: Comparable{
    static func < (lhs: User, rhs: User) -> Bool{
        return lhs.registerationDate < rhs.registerationDate
    }

    static func == (lhs: User, rhs: User) -> Bool{
```

```
        return lhs.registerationDate == rhs.registerationDate
    }
}

let user1 = User(id: 1, name: "Gwinyai", registerationDate: Date.init(timeIntervalSinceNow: 5))
let user2 = User(id: 2, name: "Paul ", registerationDate: Date.init(timeIntervalSinceNow: 1))

if user1 < user2 {
    print("\(user1.name) registered before \(user2.name)")
} else {
    print("\(user2.name) registered before \(user1.name)")
}

// "Paul registered before Gwinya

var collectionOfUsers = [user1, user2]
collectionOfUsers.sort()
let newCollection = collectionOfUsers.sorted()
```

Hashable Protocol

Any type that conforms to Hashable can be used in a set or as a Dictionary key, The idea is that a type that is Hashable is unique

Set<Element>

following conditions needed:

A struct whose stored properties all conform to Hashable

An enum whose associated values all conform to Hashable

For any other custom type, implement the hash(into:) method in addition to conforming to the protocol Equatable because Hashable inherets from Equatable

Int and string already conform to hashable

- must be unique
- unordered collection

```
let studentScores = ["Gwinya": 60, "Paul": 50]
```

- usually means generatic type

```
let students : Set<String> = Set<String>(["Gwinyai", "Paul", "Gwinyai"])
```

collection, unordered, not duplicated

```
print(students.count)

class User {
    var name: String = ""
}

struct Song {
    var id: String
    var artisit : String
    var genre : String
    //var user: User
}
```

```
extension Song: Hashable{
    static func == (lhs: Song, rhs: Song) -> Bool{
        return lhs.id == rhs.id
    }
    func hash(into hasher: inout Hasher){
        hasher.combine(id)  // if more factory just add => hasher.combine(X)
    }
}

let song1 = Song(id: "1", artist: "Gwinyai", genre: "Lofi")
let song2 = Song(id: "2", artist: "Paul", genre: "Pop")
let song3 = Song(id: "1", artist: "Gwinyai", genre: "Lofi")

let musicPlaylist: Set<Song> = Set<Song>([song1, song2, song3])
print(musicPlaylist.count) //2
musicPlaylist.forEach({print($0.genre)})
```

// Generics

allow us to create reusable code

arrays can work with different types such as Int and String because an Array uses a generic type parameter called Element

Generics start with a type parameter, can simply be a letter enclosed with diamond brackets

T, U, V often used where a meanful name does not exist, type parameter is a place holder for any type

can provide type constraints to our generic code to restrict or enforce some kind of conformance on a type parameter. It's often useful to add type constrains using abstract types like Equatable and Comparable

can go a step further and provid generic where clauses. These give us the ability to create properties and methods for specific types of a type.

```
var numberArray: [Int] = [1, 2, 3, 4, 5]

var numberArray: Array<Int> = [1, 2, 3, 4, 5]

var numberDictionary: [String: Int] = ["1": 1, "2": 2]

var numberDictionary: Dictionary<String, Int> = ["1": 1, "2": 2]
// struct Dictionary<Key, Value> where Key: Hashable
                                       ↳ Generic constrains

func mergeTogether(a: String, b: String) -> [String]{
    return [a, b]
}

func mergeTogetherInts(a: Int, b: Int) -> [Int]{
    return [a, b]
}
```

=>

```
            ↑ just a generic place holder
              also U, V
func mergeTogether<T>(a: T, b: T) -> [T]{
    return [a, b]
}
            ↓ can also be
func mergeTogether<T, U>(a: T, b: U) -> [T]{
    return [a, b]
}
```

```
let mergedStrings = mergeTogether(a: "Gwinya!", b: "Pan(")
print(mergedStrings)

let mergedInts = mergeTogether(a: 1, b: 2)
print(mergedInts)

first in first out

class Queue<T>{
    var storage: [T]
    init(head: T){
        storage = [head]
    }
    func enqueue(_ element: T){      // add element to the queue
        storage.append(element)
    }
    func peek() -> T?{               // return the first element
        guard !storage.isEmpty else {return nil}
        return storage[0]
    }
    func dequeue() -> T?{
        guard !storage.isEmpty else {return nil}
        return storage.remove(at: 0)
    }
}

let ourFirstQueue = Queue(head: 2)
ourFirstQueue.enqueue(5)
ourFirstQueue.eaqueue(9)
print(ourFirstQueue.dequeue() ?? "there was nothing left")
print(ourFirstQueue.dequeue() ?? "there was nothing left")      => //  2
print(ourFirstQueue.dequeue() ?? "there was nothing left")          5
print(ourFirstQueue.dequeue() ?? "there was nothing left")          9
                                                                   "there was nothing left"


func isTheSame<T: Equatable>(a: T, b: T) -> Bool{
    return a == b
}

print(isTheSame(a: "Gwinya", b: "Pan")? "they sam": "they not same")

func changeBackgroundColor<T: UIView>(view: T){
    view.backgroundColor = UIColor.blue
}

changeBackgroundColor(view: UIButton(frame: CGRect(x:0, y:0, width: 100, height: 100)))
```

```
class Design<T: UIView>{
    let view: T
    init(view: T){
        self.view = view
    }
}

extension Design where T: UIButton{
    func changeTitle(to title: String){
        view.setTitle(title, for: .normal)
    }
}
extension Design where T: UILabel{
    func changeFont(to font: UIFont){
        view.font = font
    }
}

extension Design where T: UILabel{
    func changeFont(to font: UIFont){
        view.font = font
    }
}

let genericButton = Design(view: UIButton(frame: CGRect(x:0, y:0, width:100, height:100)))
let genericLabel = Design(view: UILabel(frame: CGRect(x:0, y:0, width:100, height:100)))

genericButton.changeTitle(to: "New Button")
genericLabel.changeFont(to: UIFont.systemFont(ofSize:16))
```