what learned

*best animation for transparent a view

```
blackView.alpha = 0

            UIView.animate(withDuration:
0.5, animations: {
                blackView.alpha = 1
            })
```

*when we change orientation of device some size of cells in collection view get messy or get wrong sizes to fix this we use inavalidatLayout in willTransition method:

```
override func willTransition(to
newCollection: UITraitCollection, with
coordinator:
UIViewControllerTransitionCoordinator) {


coordinator.animate(alongsideTransition: {_
in

self.collectionView.collectionViewLayout.in
validateLayout()
        }, completion: nil)

    }
```

∗ **Make custom delegation weak:**

```
class A {
   weak var delegate: myCustomDelegation?
   //making delegate as weak to break the retain
cycle
}
```

*In Swift there are two *very* common implementations of decorator pattern: **Extensions** and **Delegation**.

*UICollectionViewLayoutAttributes

A layout object that manages the layout-related attributes for a given item in a collection view.
*Layout objects create instances of this class when asked to do so by the collection view. In turn, the collection view uses the layout information to position cells and supplementary views inside its bounds.
* one of the method we have to override is collectionViewContentSize method to certain the content size of whole collection view:

```
override var collectionViewContentSize:
```

```
CGSize{
    return CGSize(width: width, height: 
contentHeight)
  }
```

* the other method to override is prepareLayout method this is called layout operation gone take a place

———————————

we can use Generics for TableViews for decrease tasks like register, reload and another things and we can add more functionality like infinite scroll and Avoid repeating:

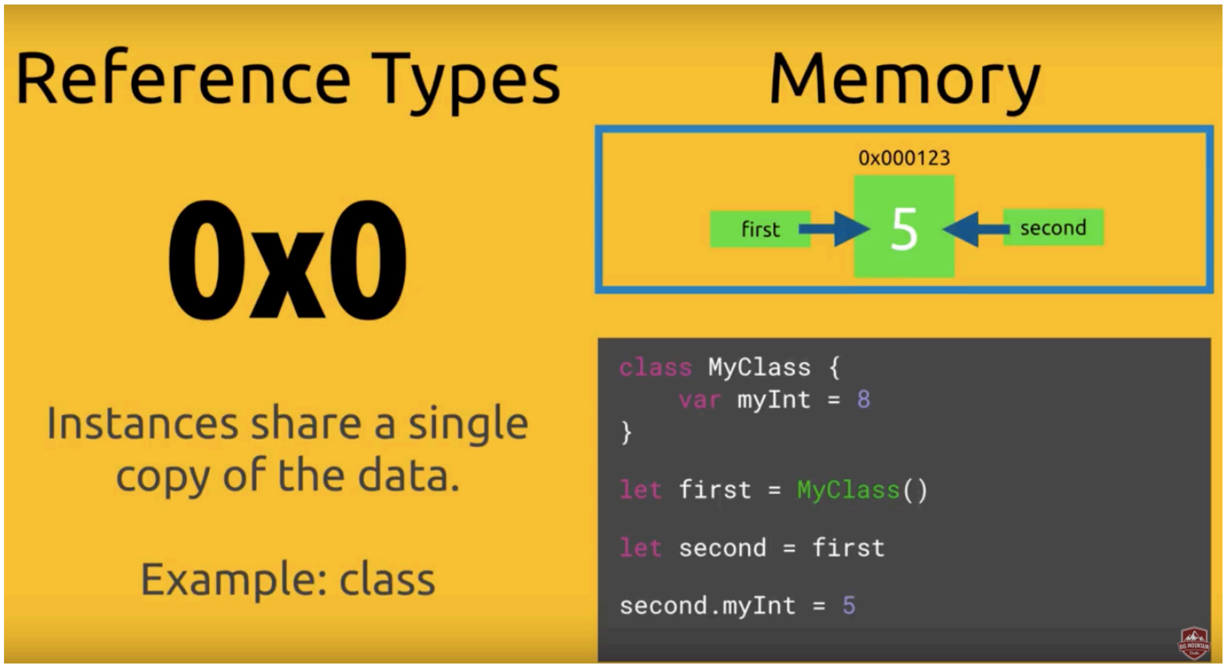
```
class BaseTableViewController<T: 
GenericCell<U>, U> : 
UITableViewController {

    let cellIdentifier = ""
    var items = [U]()
    var rowHeight: CGFloat = 50

    override func viewDidLoad() {
        super.viewDidLoad()

        cellIdentifier = 
NSStringFromClass(T.self)
```

```
        tableView.register(T.self,
forCellReuseIdentifier:
cellIdentifier)
    }

    override func tableView(_
tableView: UITableView,
numberOfRowsInSection section: Int) ->
Int {
        return items.count
    }

    override func tableView(_
tableView: UITableView, cellForRowAt
indexPath: IndexPath) ->
UITableViewCell {

        let cell =
tableView.dequeueReusableCell(withIden
tifier: cellIdentifier, for:
indexPath) as! GenericCell
        cell.item =
items[indexPath.item]
        return cell
    }

    func tableView(_ tableView:
```

```
UITableView, heightForRowAt indexPath:
IndexPath) -> CGFloat {
        return rowHeight
    }
```

using very easy and fast:

```
class UserCell: GenericCell<User> {
    override var item: User?{
        didSet{
            textLabel?.text =
item?.name
        }
    }
}

class UsersTableViewController:
BaseTableViewController<UserCell,
User> {

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.rowHeight = 120
        reloadData()
    }

    // dont need to impelement
datasource Delegate and another any
more and if its need we just override
them.
}
```



Reference Types

0x0

Instances share a single copy of the data.

Example: class

Memory

```
class MyClass {
    var myInt = 8
}

let first = MyClass()

let second = first

second.myInt = 5
```

the swift ARC (Automatic Reference Counting) for manage memory
ARC: in this system references to object get counted when count equal zero the object will remove
* a weak or unowned references dose not

increment the reference count.
* difference weak and unowned: weak-> child is optional for parent and when parent removed child will removed, unowned -> child is not optional and definitely exist all the time but when parent remove, it will removed
* weak for optionals and unowned for forces
* unowned references has to have a value

— — — — — — — — — —

*closures are reference types and they have its own object and own address memory
* we can pass data to closure:

```
var clouser = { [label = UILabel(),
str = "text", age = 23] in
        label.text = "\(str):\(age)"
    }
```

because in above code label has strongrefrence we set it weak:

```
var clouser = { [weak label =
UILabel(), str = "text", age = 23] in
        label?.text = "\(str):\(age)"
    }
```

in closures when we use self we can use unowned reference because self never won't be nil

```
    var clouser = { [unowned self] in
        print("\(self) is unowned never be nil")
    }
```

———————————————

*we can use deinit() method to check objects like view controllers are removed from memory
* when we dismiss ViewController its not removed form memory
* when we use observers in swift we have to removeObserver to remove from memory
* the child class exist as long as the parent exist
*we can limit protocol adoption to class types and not structures or enumerations, by adding the AnyObject protocol to a protocols inheritance list:

```
protocol myProtocol: AnyObject {
    func doSomthing()
}
```

— — — — — — — — — — — — — — — — — — — —

Lazy variables that aren't closure properties are not retained by anything so there is no need to use `[unowned self]` here:

```
class Test {
    lazy var tableView: UITableView = {

        let tableView = UITableView(frame:
self.view.bounds, style: .plain)
        tableView.delegate = self
        tableView.dataSource = self
        return tableView
    }()
}
```

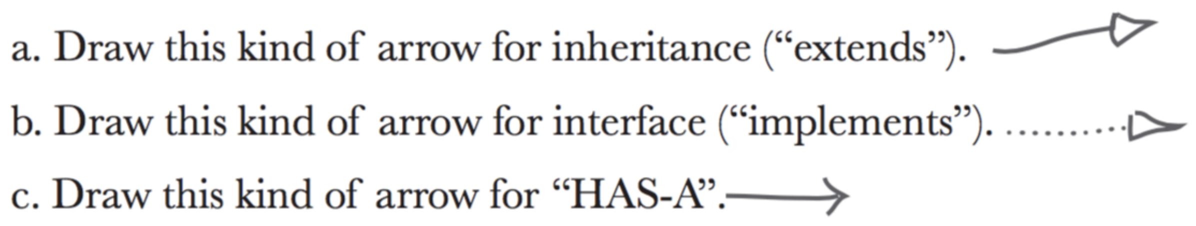
— — — — — — — — — — — — — — — — — —

design principles:

- identify aspect of your application that vary and separate them from what stays the same.
- **program to an interface not an implementation**
- favor composition over inheritance (composition better than inheritance: HAS-A can be better than IS-A)
- strive for loosely coupled designs between objects that interact.
- class should be open for extension, but closed for modification.(open close principle)
- depend upon abstraction. do not depend upon concrete classes.

- principle of less knowledge - talk only to your immediate friends. (this purpose of fecade)
- hollywood principle: don’t call us, we’ll call you

* hollywood principle says: high-level modules not depending on low levels.
* the strategy pattern: defines family of algorithm, encapsulates each one and makes them interchangeable specially at runtime. strategy lets the algorithm vary independently from clients that use it. **it works for runtime**

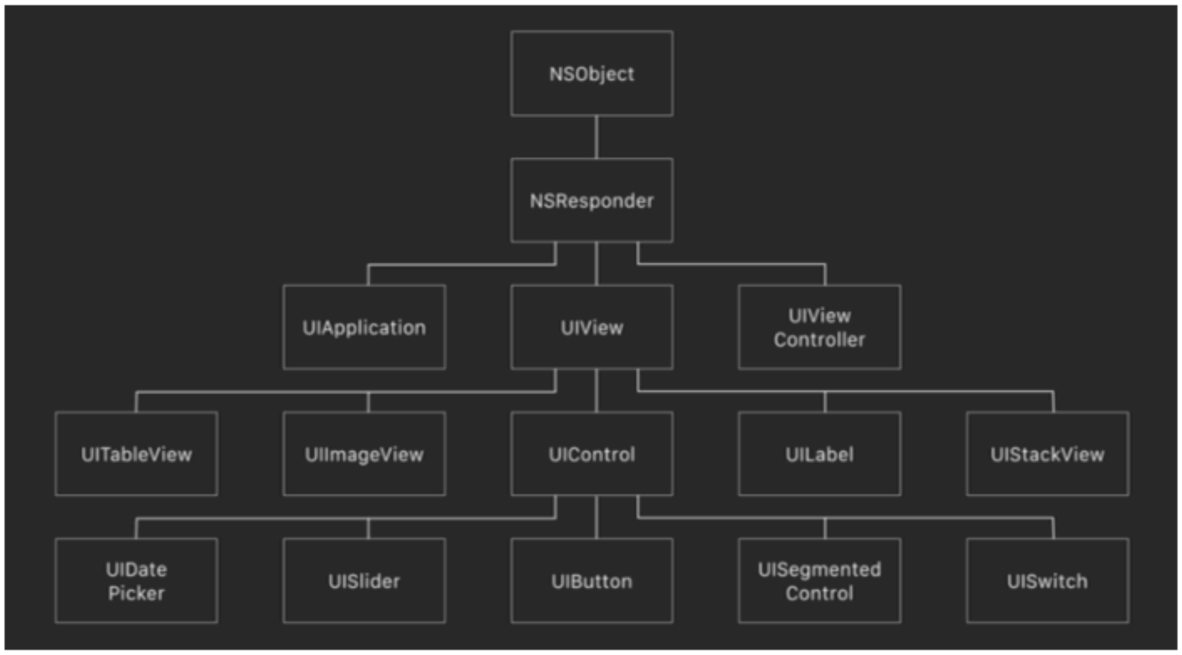


a. Draw this kind of arrow for inheritance (“extends”).

b. Draw this kind of arrow for interface (“implements”).

c. Draw this kind of arrow for “HAS-A”.

*** Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.



2015 WWDC_Hideous Structure

*solid

```
//in the name of God
//
//solid:
//S->SRP : Single Responsibility
//o->OCP: open-closed principle
//L->LSP: Liskov substitution principle
جایگزینی
//I->ISP: Interface segregation
principle  تفکیک
//D->DIP: dependency Inversion principle
اینورتر
//
//* priciple: کلی  قاعده  -اصل
```

```
//*decoupling: جدا کردن, مجزا کردن,
تفکیک کردن, سوا کردن

// for Solid ask yourself
// -how many responsibilities does this
class have?
// -Do i need to change this class to
extend its expected behaviore?
// -Is this subclass substitutable for
its parent?
// -Is this clint exposed to
functionality that it does/should not
use?
// -what parts of this class change
frequntly?
// -Does this class depand on concrete
dependencies?
// -does my class need THIS many
imports?

//s: هر ماژول نرم افزاری میبایست تنها یک
دلیل برای تغییر داشته باشد
//o: ماژول‌های نرم افزار باید برای تغییرات
بسته و برای توسعه باز باشند
//l: زیر کلاسها باید بتوانند جایگزین نوع
پایه‌ی خود باشند
//isp :  کلاینتها نباید وابسته به متدهایی
باشند که آنها را پیاده سازی نمی‌کنند.
//d : ماژول‌های سطح بالا نباید به ماژولهای
```

```
سطح پایین وابسته باشند، هر دو باید به
انتزاعات وابسته باشند. انتزاعات نباید
وابسته به جزئیات باشند، بلکه جزئیات باید
وابسته به انتزاعات باشند
```

GRASP:

```
//GRASP : General Responsibility
Assignment Software
/*
ین اصول به بررسی نحوه تقسیم وظایف بین
کلاس‌ها و مشارکت اشیاء برای به انجام رساندن
یک مسئولیت می‌پردازند. اینکه هر کلاس در
ساختار نرم افزار چه وظیفه‌ای دارد و چگونه
با کلاس‌های دیگر مشارکت میکند تا یک عملکرد
به سیستم اضافه گردد. این اصول به چند بخش
:تقسیم می‌شوند

( Controller ) کنترلر
( Creator ) ایجاد کننده
( High Cohesion ) انسجام قوی
( Indirection ) واسطه گری
( Information Expert ) دانای اطلاعات
( Low Coupling ) اتصال ضعیف
( Polymorphism ) چند ریختی
Protected ) حفاظت از تاثیر تغییرات
( Variations
( Pure Fabrication ) مصنوع خالص
```

[redacted]

بنابراین IOC می‌گوید که:

1- کلاس اصلی (یا همان Parent) نباید به صورت مستقیم وابسته به کلاس‌های دیگر باشد.

2- رابطه‌ی بین کلاس‌ها باید بر مبنای تعریف کلاس‌های abstract باشد (و یا استفاده از interface ها).

تزریق وابستگی یا Dependency injection

برای پیاده سازی IOC از روش تزریق وابستگی یا dependency injection استفاده می‌شود که می‌تواند بر اساس setter injection ، constructor injection و یا interface-based injection باشد و به صورت خلاصه پیاده سازی یک شیء را از مرحله‌ی ساخت وهله‌ای از آن مجزا و ایزوله می‌سازد.

مزایای تزریق وابستگی‌ها:

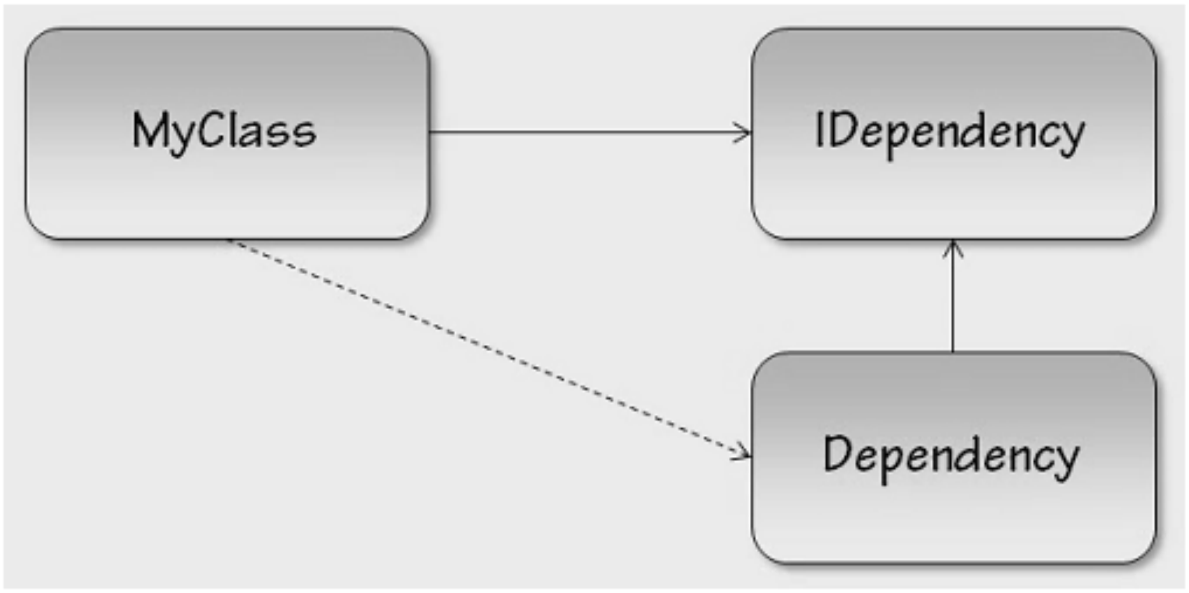
1- گره خوردگی اشیاء را حذف می‌کند.

2- اشیاء و برنامه را انعطاف پذیرتر کرده و اعمال تغییرات به آن‌ها ساده‌تر می‌شود.

معکوس سازی کنترل (Inversion of Control) الگویی است که نحوه پیاده سازی اصل معکوس سازی وابستگی‌ها (Dependency inversion principle) را بیان می‌کند.

تزریق وابستگی‌ها یا DI چیست؟

تزریق وابستگی‌ها یکی از انواع IoC بوده که در آن ایجاد و انقیاد (binding) یک وابستگی، در خارج از کلاسی که به آن نیاز دارد صورت می‌گیرد. روش‌های متفاوتی برای ارائه این وابستگی وهله سازی شده در خارج از کلاس به آن وجود دارد که در ادامه مورد بررسی قرار خواهند گرفت.



سؤال: بین IoC و DIP چه تفاوتی وجود دارد؟

در DIP (قسمت قبل) به این نتیجه رسیدیم که یک ماژول سطح بالاتر نباید به جزئیات پیاده سازی‌های ماژولی سطح پایین‌تر وابسته باشد. هر دوی این‌ها باید بر اساس Abstraction با یکدیگر ارتباط برقرار کنند. IoC روشی است که این Abstraction را فراهم می‌کند. در DIP فقط نگران این هستیم که ماژول‌های موجود در لایه‌های مختلف برنامه به یکدیگر وابسته نباشند اما بیان نکردیم که چگونه.

DIP: Dependency Inversion

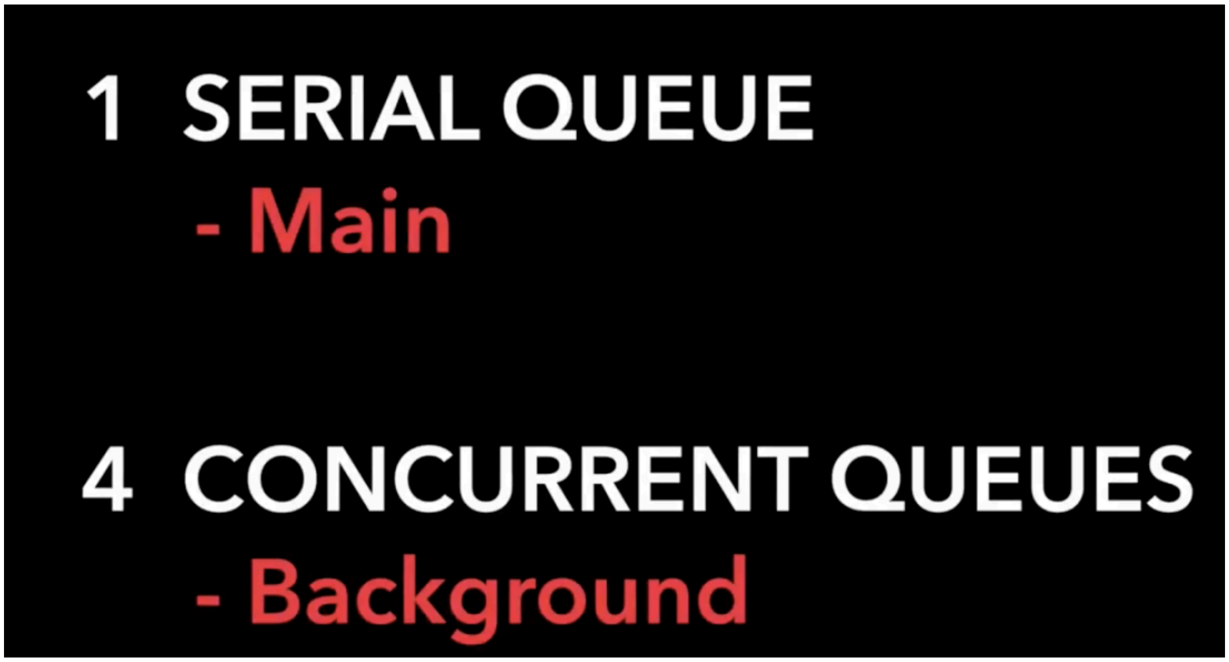
*تعریف یک اینترفیس تنها زمانی ارزش خواهد داشت که چندین پیاده سازی از آن ارائه شود.

class vs struct?
class reference type, struct value type
structs can not have inheritance

Concurency:

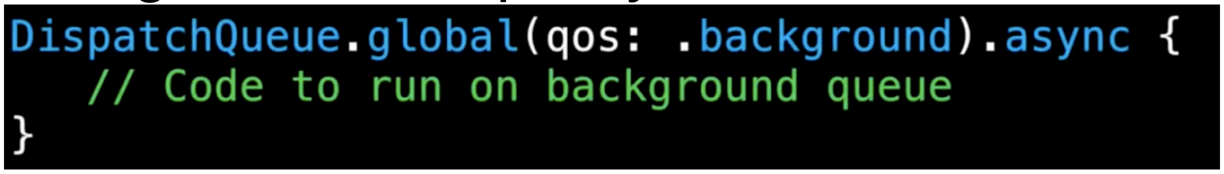
* doing some tasks at the same time
* for example iPhone use A10 quad core CPUs that can handle tasks in the same time.
* queue : First in First out: FIFO
* 2 types Queue: serial & concurrent
* serial: first task excited completely the second task executed completely then third and so on. -> Predictable,
* concurrent: tasks execute without waiting for each other.

queues in iOS: we have 2 types of queue :


1 SERIAL QUEUE
- Main

4 CONCURRENT QUEUES
- Background

*we can use this code to do tasks in background with quality of service:

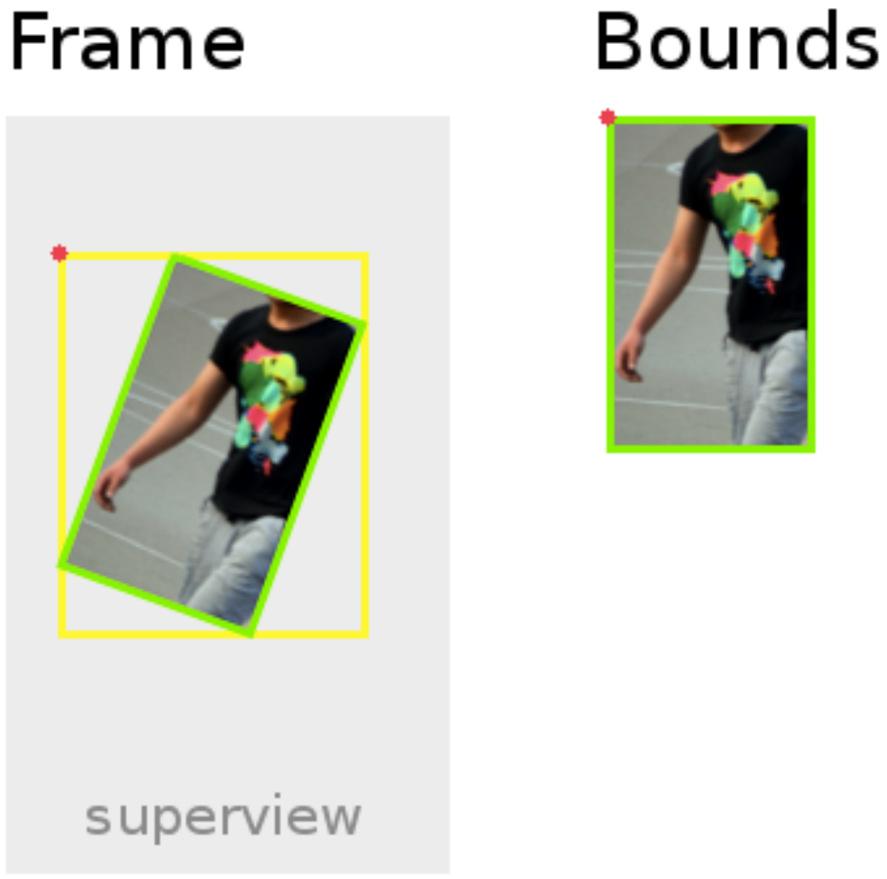

```
DispatchQueue.global(qos: .background).async {
   // Code to run on background queue
}
```

Difference Between frame and Bounds:

frame: is a position relative to parent view.
bound: is a position relative to won coordinate system.

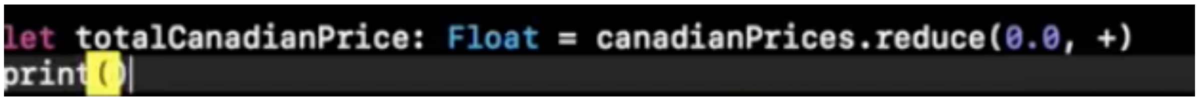
whats deference : when we rotate or transform the view its frame will be change , x , y changed , width and height will be changed and not equal to its bounds.

bounds and frame are very similar until transform and rotate.



* map function for manipulate data like multiply, decrease, change

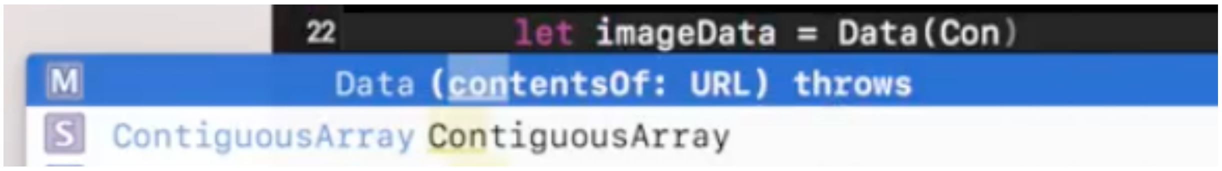
* reduce function for combine all of values to one like taking sum up all data, total, or average ..:



```
let totalCanadianPrice: Float = canadianPrices.reduce(0.0, +)
print()
```

*delegation pattern: is one to one communication pattern
* Notification and observers are is one to many pattern

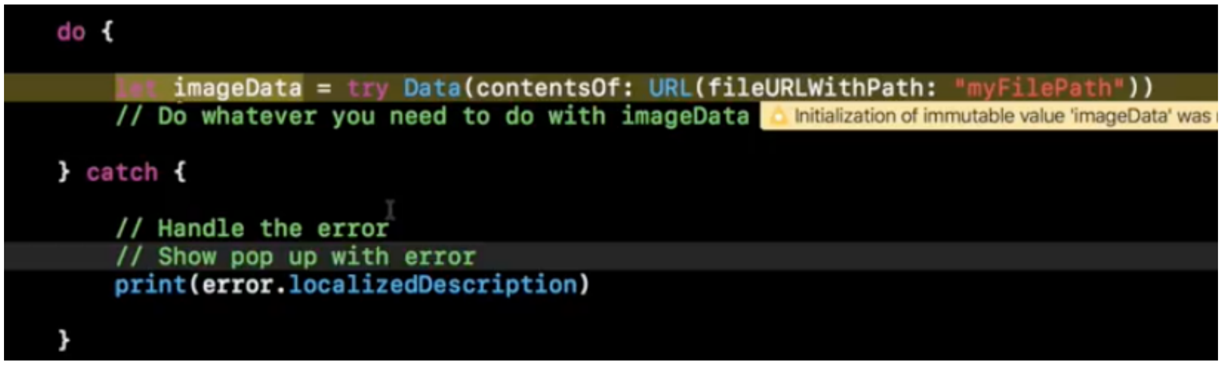
* when we have function that throws we need to try catch and we have to use do catch block:



```
let imageData = Data(Con)
```

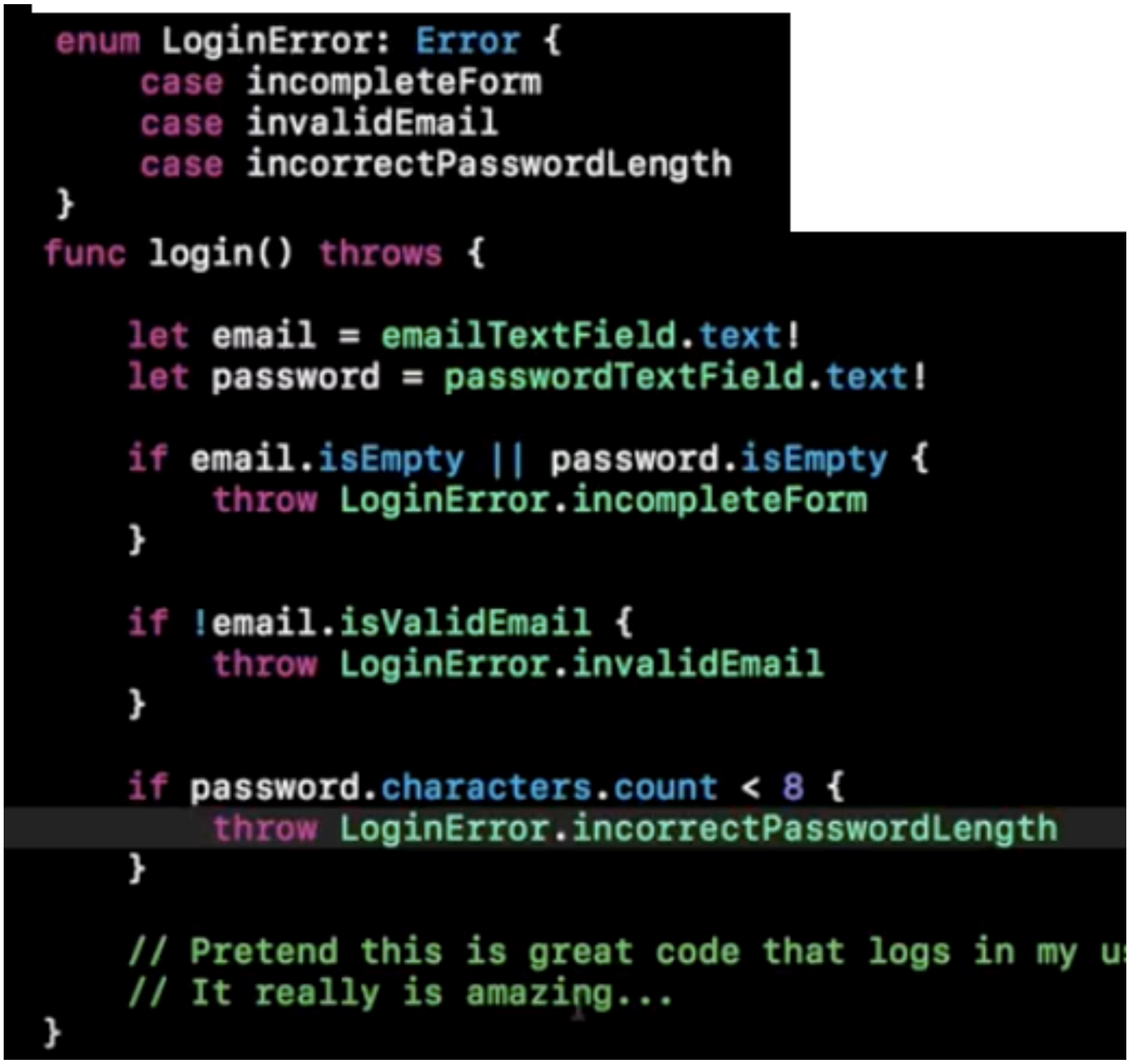
M
Data (contentsOf: URL) throws
S
ContiguousArray ContiguousArray

do catch block



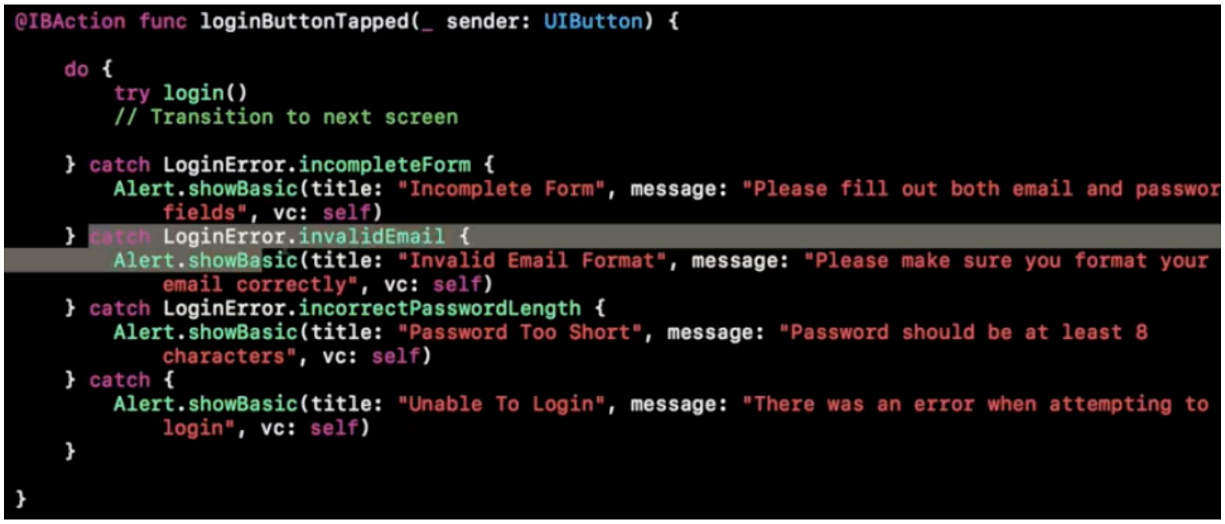
```
do {

    let imageData = try Data(contentsOf: URL(fileURLWithPath: "myFilePath"))
    // Do whatever you need to do with imageData   Initialization of immutable value 'imageData' was

} catch {

    // Handle the error
    // Show pop up with error
    print(error.localizedDescription)

}
```

* function that throws:



```
enum LoginError: Error {
     case incompleteForm
     case invalidEmail
     case incorrectPasswordLength
}
func login() throws {

    let email = emailTextField.text!
    let password = passwordTextField.text!

    if email.isEmpty || password.isEmpty {
        throw LoginError.incompleteForm
    }

    if !email.isValidEmail {
        throw LoginError.invalidEmail
    }

    if password.characters.count < 8 {
        throw LoginError.incorrectPasswordLength
    }

    // Pretend this is great code that logs in my u
    // It really is amazing...
}
```

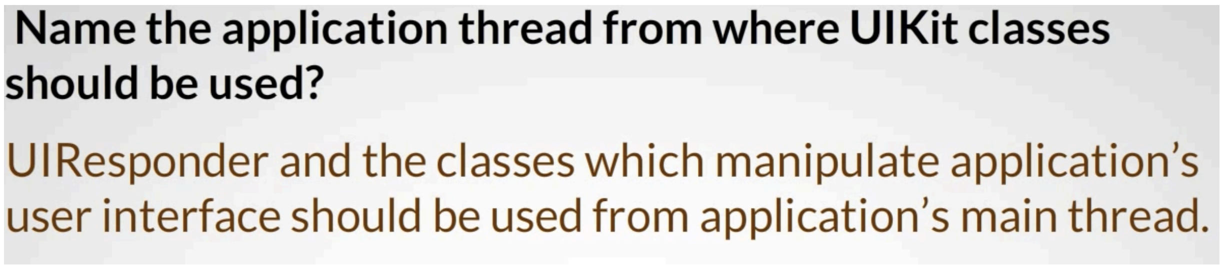
and catch this function:



```
@IBAction func loginButtonTapped(_ sender: UIButton) {

    do {
        try login()
        // Transition to next screen

    } catch LoginError.incompleteForm {
        Alert.showBasic(title: "Incomplete Form", message: "Please fill out both email and passwor
            fields", vc: self)
    } catch LoginError.invalidEmail {
        Alert.showBasic(title: "Invalid Email Format", message: "Please make sure you format your
            email correctly", vc: self)
    } catch LoginError.incorrectPasswordLength {
        Alert.showBasic(title: "Password Too Short", message: "Password should be at least 8
            characters", vc: self)
    } catch {
        Alert.showBasic(title: "Unable To Login", message: "There was an error when attempting to
            login", vc: self)
    }

}
```

defrence between framework, literary and third party?

UIKit is an framework



Name the application thread from where UIKit classes should be used?

UIResponder and the classes which manipulate application's user interface should be used from application's main thread.