

IOS İşletim Sisteminde Uygulama Geliştirme

Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 04/05/2017

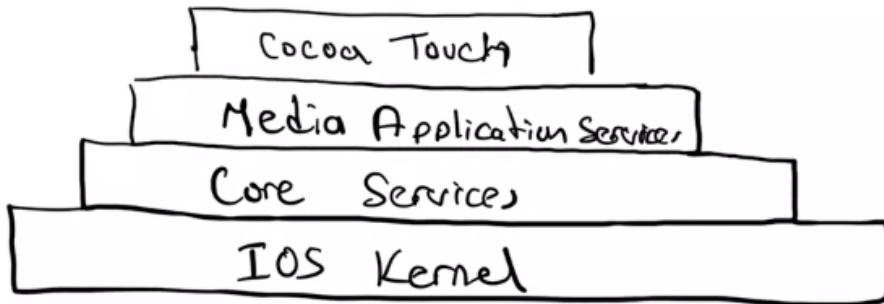
Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabılır.

Cocoa Touch Ortamı

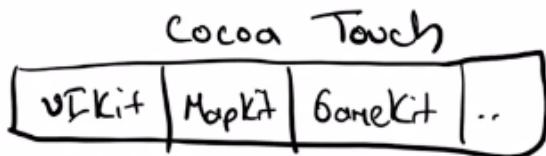
Bilindiği gibi Cocoa (Cocoa Framework) Apple'ın temel GUI uygulama geliştirme ortamıdır. Bu ortam Steve Jobs'ın Apple'dan ayrıldığı dönemde NeXT firması tarafından geliştirilmiştir. Daha sonra Mac OS'in 10 versiyonlarıyla (Mac OS X) birlikte Mac dünyasına sokulmuştur. Cocoa ortamının kaynak kodları Objective-C ile gerçekleştirilmiştir. Ancak bugün bu ortam hem Objective-C ile hem de Swift ile kullanılabilmektedir. Zaten Swift de sırı bu yüzden belli miktarda Objective-C özelliklerini barındırmaktadır.

Cocoa Touch Cocoa ortamının mobil cihazların için uyarlanmış halidir. Cocoa ile Cocoa Touch arasındaki ilişki örneğin .NET ile .NET Compact Framework arasındaki ilişkiye benzetilebilir. Cocoa Touch bugün IPhone, IPad, IPod, IWatch ve Apple TV cihazlarının programlanmasıında yoğun olarak kullanılmaktadır.

Cocoa Touch aslında yüksek seviyeli bir ortam sunmaktadır. Bu ortam daha alçak seviyeli ortamların üzerine oturtulmuştur:



Cocoa Touch ortamı da aslında alt sistemlerden ya da kütüphanelerden oluşmaktadır.



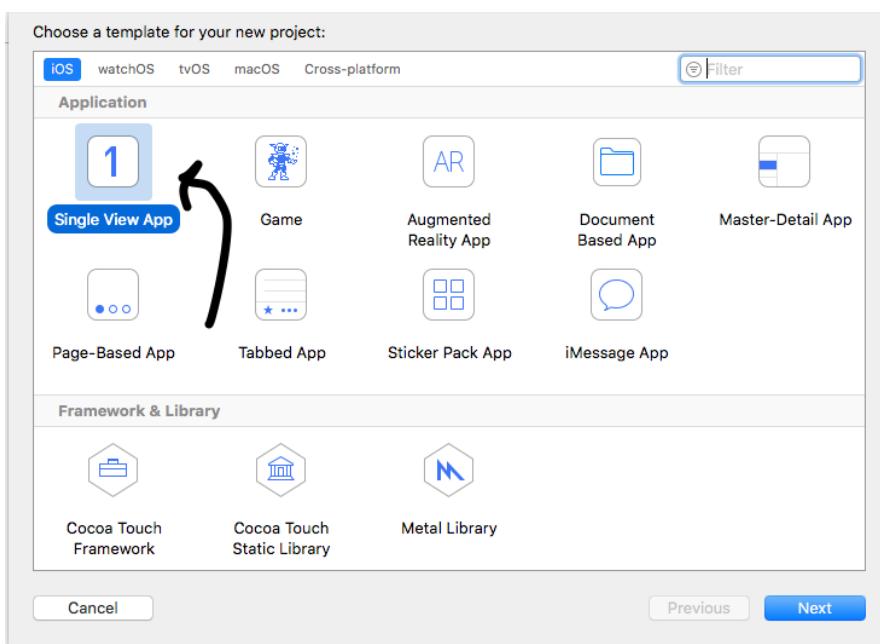
Şüphesiz Cocoa Touch ortamın en önemli alt sistemi "UIKit" denilen kütüphanedir. Bu kütüphanedeki sınıfların isimleri hep UIXXX biçiminde isimlendirilmiştir.

İskelet iOS Uygulaması ve XCode IDE'sinin Temel Özellikleri

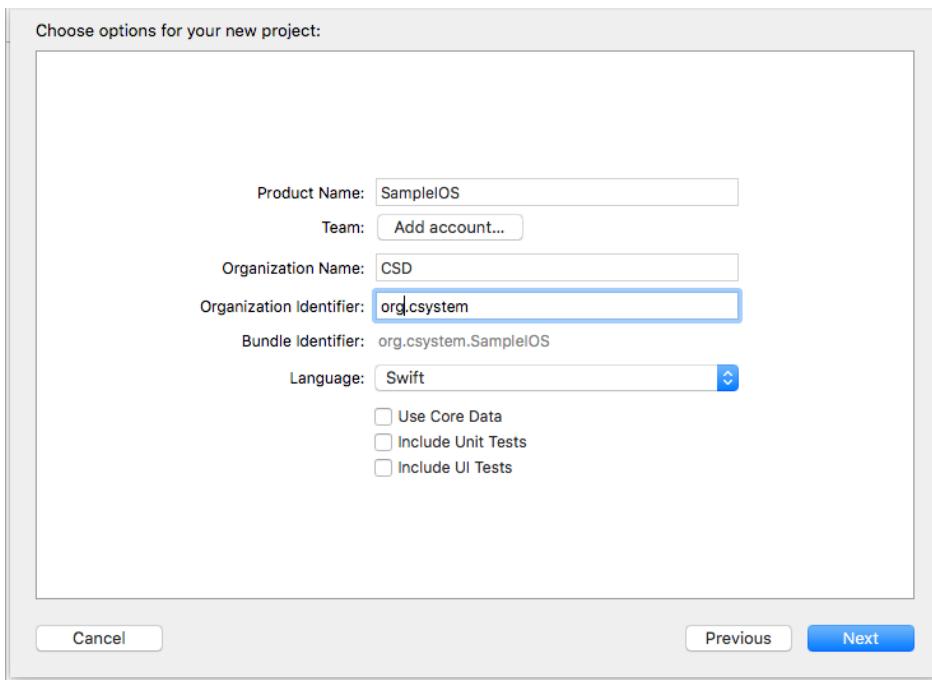
XCode IDE'si sürükle bırak tarzı pek çok özelliği ile hızlı uygulama geliştirme için kullanılmaktadır. Aslında mutlak anlamda iOS'ta uygulama geliştirme için bir IDE'ye gereksinim yoksa da pratikte XCode olmadan iOS uygulaması geliştirme rastlanan bir durum değildir.

Kursumuzdaki çalışma sırasında mümkünse hem XCode hem de mobil cihanlarımızdaki iOS sürümünü en güncel halde tutmalıyız. Örneğin iOS 12.0 için en güncel XCode sürümü 10.0'dır. Özellikle minör versiyonların uyusmasına dikkat edilmelidir. Yani iOS 12.X için olması gereken XCode 10.X biçimindedir.

İskelet bir iOS projesi için File/New Project seçilir. Burada iOS sekmesine gelinir ve buradan "Single View App" seçilir.



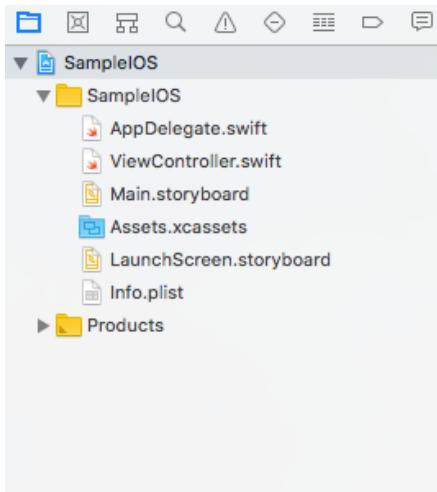
Bundan sonra karşımıza aşağıdaki ekran gelecektir:



Her projede onu betimleyen bir "Product Name" vardır. Bu isim geliştireceğimiz uygulamayı betimleyen herhangi bir isim olabilir. "Organization Identifier" ters domain ismi biçiminde verilmelidir. Ancak böyle bir domain'in bulunması zorunlu değildir. Örneklerimizde biz bunu "org.system" biçiminde vereceğiz. Programlama Dilinin Swift olduğuna dikkat ediniz.

Bundan sonra bize projenin hangi dizinde saklanacağı sorulmaktadır.

Bir IOS projesi yaratıldığında tipik bazı dosyalar IDE tarafından oluşturulmaktadır.



Uzantısı .swift olan dosyalar Swift kaynak dosyalarıdır. Uzantısı .storyboard olan dosyalar ise UI dosyalarıdır. .storyboard uzantılı UI dosyaları aslında XML barındırmaktadır. Bu dosyaların içerisindeki XML yükleme sırasında okunup UI elemanlarına dönüştürülmektedir. Bu anlamda çalışma biçimi Qt ortamına benzetilebilir. Tabii aslında hiç .storyboard dosyası kullanmadan programalama yoluyla da görsel arayüz oluşturulabilir. Biz bunun nasıl yapılacağı hakkında daha sonra fikirler vereceğiz. Ancak temel çalışma biçimimiz .storyboard dosyalarını kullanmak biçiminde olacaktır.

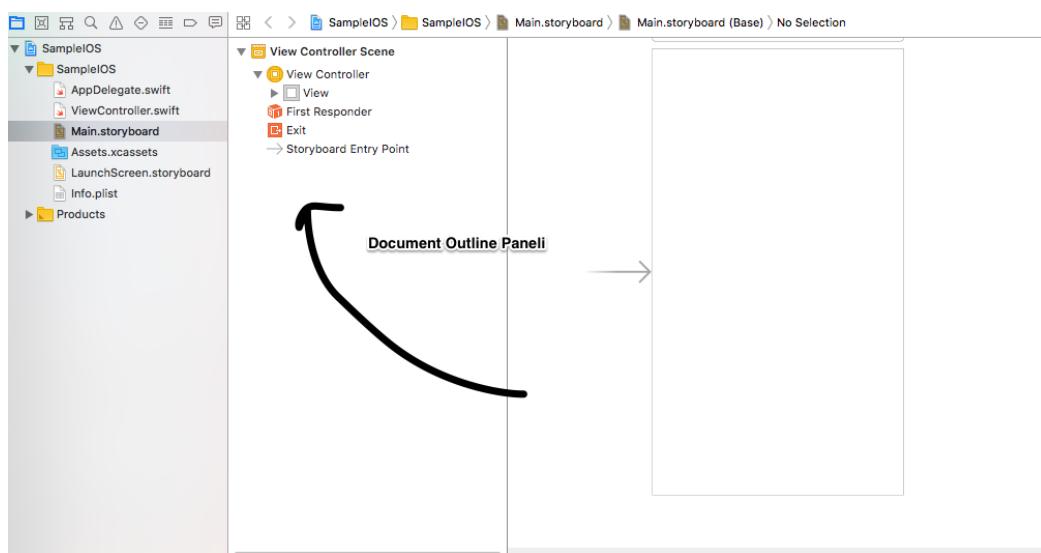
Proje yaratıldığında iki storyboard dosyası olduğunu görüporsunuz: Main.storyboard ve LaunchScreen.storyboard dosyaları. Main.storyboard dosyası ana ekranın UI tasarımını temsil etmektedir. LaunchScreen.storyboard dosyası ise uygulama yüklenirken görüntülenecek öğeyi temsil eder.

XCode IOS uygulamaları belli bir süreden sonra MVC (Model-View-Controller) tarzı bir çalışmaya geçmiştir. MVC mimarisinde Model projenin verilerini, View görsel tarafını ve Controller da verilerle görselliği birbirine bağlayan öğeyi temsil etmektedir. Bu mimarinin IOS'taki kullanımı ileride ele alınacaktır.

XCode'da bir proje açıkken pencerenin solundaki panele "Dolaşım (Navigation) Paneli" denilmektedir. Buradaki en önemli iki sekme "project navigator" sekmesi ve "issue navigator" sekmesidir. Project Navigator sekmesi projedeki dosyaları, Issue Navigator sekmesi ise hata mesajlarını göstermektedir. Diğer sekmeler yeri geldikçe açıklanacaktır.

XCode'un görsel arayüz oluşturmada kullanılan kısmına "Arayüz Oluşturucu (Interface Builder)" denilmektedir. Bazı isimlendirmelerde IB öneki "Interface Builder" sözcüklerinden kısaltma için kullanılmaktadır.

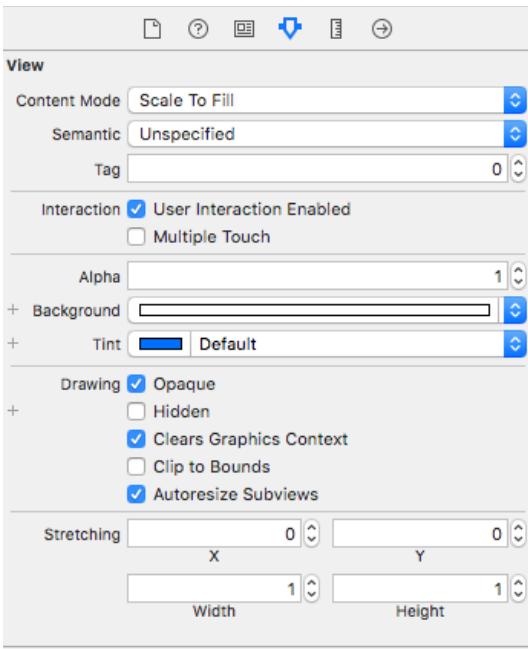
Interface Builder'daki solda bulunan panele "Document Outline" paneli denilmektedir.



Bu panel görsel öğelerin seçilmesi için sık kullanılmaktadır. Panelde arayüzü oluşturan görsel elemanlar hiyerarşik bir biçimde listelenmektedir. Programcı isterse ilgili görsel elemanı buradan da seçebilir. Bu panel aşağıdaki simge ile kapatılıp açılabilir.

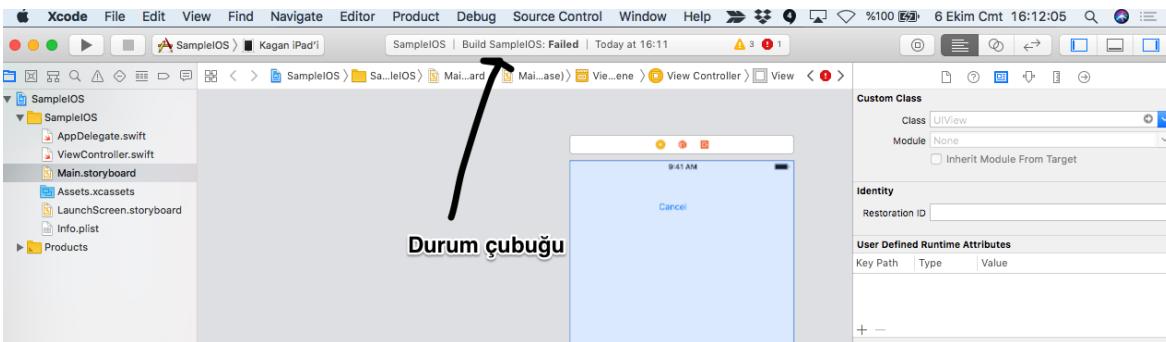


XCode IOS projesinde ekranın en sağındaki panele "Inspector" paneli denmektedir. Bu panelde çeşitli sekmeler vardır.

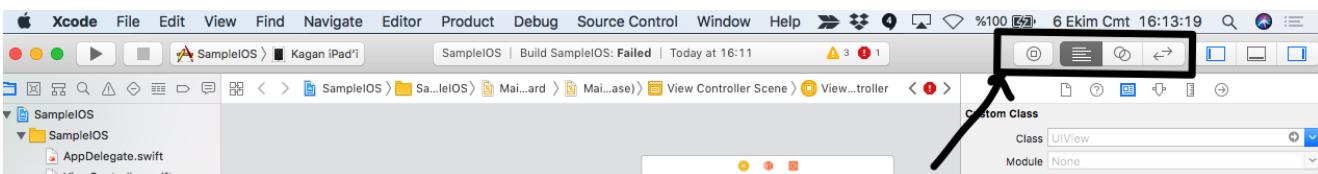


Buradaki sekmelerden en önemlileri "Attributes Inspector", "Size Inspector" ve "Quick Help Inspector" isimli sekmelerdir. Attribute Inspector o anda seçilmiş olan görsel öğelerin özniteliklerini değiştirmekte kullanılır.

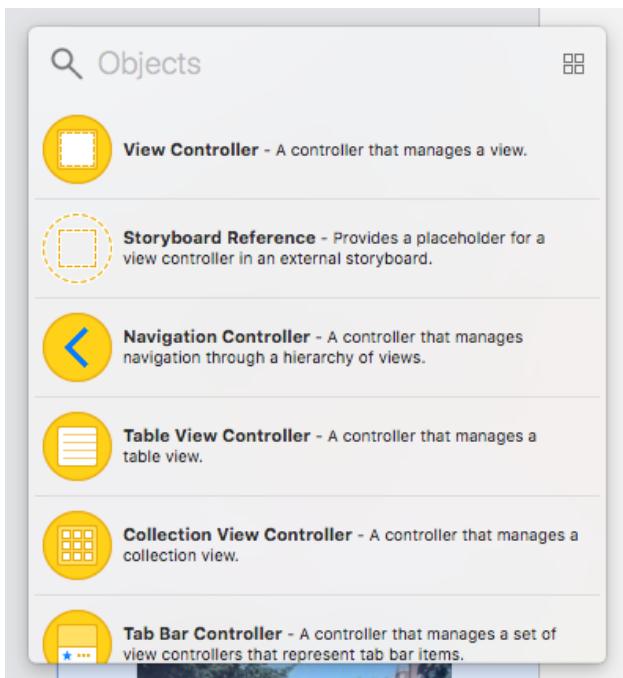
Yukarıdaki çubuğun ortasına "durum çubuğu (status bar)" denilmektedir.



Burada uygulanmanın içinde bulunduğu durum hakkında bilgi verilmektedir. Durum çubuğunun sağında yan yana dört düğme bulunmaktadır:



En soldaki düğme "nesne kütüphanesinin (object library)" açılıp kapatılmasını sağlamaktadır. Nesne Kütüphanesi görsel öğelerin bulunduğu bir listeden oluşmaktadır. Burada istenilen görsel öğe seçilerek View üzerine yerleştirilebilmektedir.



Üst bölümdeki diğer düğme içi içe geçmiş iki çember görüntüsünün bulunduğu düğmedir. Buna "Yardımcı Editor (Asistant Editor)" denilmektedir. Yardımcı editör bizim tasarıma ilişkin View ekranı ile kod dosyasını beraber görmemizi sağlar. Bu sayede geliştirici görsel öğelerle Controller sınıfı arasında bağlantılar kurabilmektedir.

UIKit İçerisindeki Bazı Temel Sınıflar

Bazı sınıflar IOS programlamasında dolaylı olarak çok sık kullanılmaktadır. Burada bu sınıflar temel olarak açıklanacaktır.

UIColor Sınıfı

UIColor Cocoa'da renk belirtmek için kullanılan temel bir sınıfıdır. Pek çok UI nesnesi bizden çeşitli biçimlerde UIColor nesnesi istemektedir. Her ne kadar XCode IB (Interface Builder) arayüzünde renkler görsel biçimde belirlenebiliyorsa da programlama yoluyla renk belirtmek için mecburen bu sınıfın faydalılmaktadır.

UIColor sınıfında renk çok çeşitli biçimlerde belirtilebilmektedir. Sınıfın init(red:green:blue:alpha) parametreli init metodu bizden renki RGBA olarak almaktadır. Bu metodun parametreleri 0 ile 1 arasında Float değerlerden oluşmaktadır.

```
let redColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
```

Sınıfın bir grup hesaplanmış property elemanı bize UIColor türünden ilgili renkleri vermektedir. Örneğin:

```
red  
green  
blue  
blue  
brown  
cyan  
orange  
purple  
...
```

Böylece bir metot bizden UIColor nesnesi istediğiinde biz ona doğrudan UIColor.red, UIColor.green, UIColor.blue gibi değerleri verebiliriz.

CGPoint Yapı

CGPoint sınıfı bir pixel2in koordinatlarını tutmak için kullanılmaktadır. Yapının init(x:y:) metodu bizden x ve y değerlerini alarak nesneyi oluşturur. Örneğin:

```
let point = CGPoint(x: 10, y: 20)
```

Benzer biçimde bu değerler yine sınıfın x eve y isimli hesaplanmış property elemanlarıyla elde edilebilmektedir.

CGSize Yapısı

Bu yapı da genişlik-yükseklik bilgisini tutmak için kullanılmaktadır. Yapının init(width:height) metodu bizden genişlik yükseklik bilgisini alarak nesneyi oluşturur. İsterske biz de bu değerleri yine yapının width ve height isimli hesaplanmış property elemanlarıyla geri alabiliriz. Örneğin:

```
let size = CGSize(width: 10, height: 5)
```

CGRect Sınıf

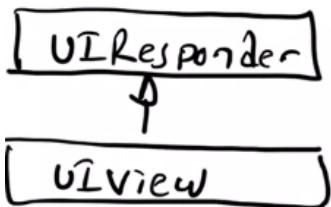
iOS'ta dikdörtgensel alanlar CGRect isimli sınıfla temsil edilmektedir. Bu sınıfın init(x:y:width:height) metodu bizden dikdörtgenin sol üst köşe koordinatlarıyla genişlik ve yüksekliğini ister. Örneğin:

```
let rect = CGRect(x: 10, y: 10, width: 100, height: 200)
```

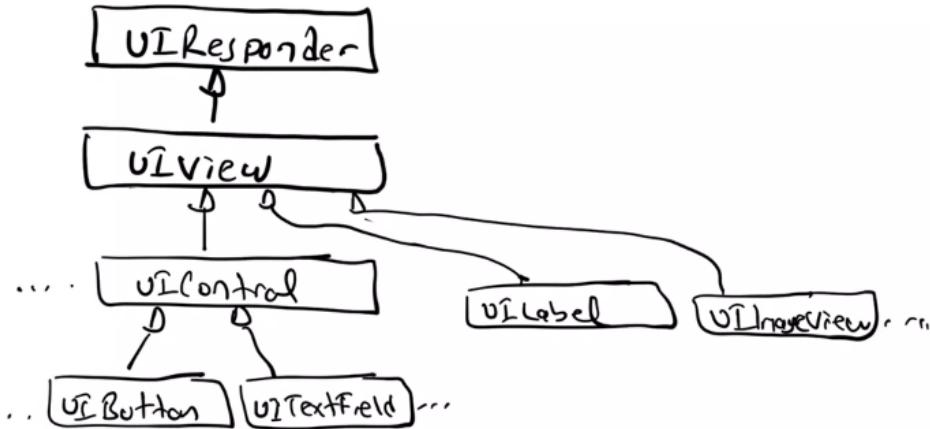
CGRect sınıfının width ve height isimli hesaplanmış property elemanları bize dikdörtgenin genişlik ve yükseklik değerlerini verir.

UIKit İçerisindeki Pencere Hiyerarşisi

UIKit'teki pencere kavramı UIView sınıfıyla temsil edilmiştir. Bir view bağımsız bir çizim bölgesi oluşturmaktadır. UIButton gibi, UILabel gibi bütün görsel öğeler dolaylı olarak UIView sınıfından türetilmiş durumdadır. UIView sınıfı UIResponder isimli sınıfından türetilmiş durumdadır. UIResponder sınıfı temel event sistemini oluşturan sınıfır. Böylece UIView sınıfı çeşitli olaylara tepki verebilmektedir.



UIControl isimli sınıf UIView sınıfından türetilmiştir. Tüm görsel elemanların bazı ortak özellikleri bu sınıfta toplanmıştır. Bu sınıf UIView sınıfının görsel öğeler için biraz özelleştirilmiş bir biçimini sunmaktadır. Nihayet pek çok görsel öğe aslında UIControl sınıfından türetilmiş durumdadır. Tabii tüm görsel öğeler UIControl sınıfından türetilmiş değildir. Bzları doğrudan UIView sınıfından türetilmiştir.



UIView Nesneleriyle Sıfırdan Programlama Yoluyla Alt Pencere Oluşturulması

Biz programlama yoluyla sıfırdan bir View nesnesi yaratıp onu ana pencereye yerleştirebiliriz. Bu işlem en aşağı seviyede bir alt pencere oluşturma işlemidir. Bu sağlamak için View nesnesi View sınıfının CGrect parametrelü init(frame:) metoduyla oluşturulur. Sonra da üst pencereye bağlanır. Yaratılan view'nun zemin rengi üst pencere zemin rengiyle aynı olmaktadır. Bu nedenle bizim yeni yaratılan view'nun zemin rengini farkedilebilsin diye değiştirmemiz uygun olur. View'nun zemin rengi UIView sınıfının UIColor türünden backgroundColor property'si ilke değiştirilmektedir. Bir view hangi pencerenin içerisinde görüntüleneceğse o pencereye ilişkin view'nun SubView listesine eklenmesi gereklidir. Bu işlem View sınıfının addSubView(_:) metoduyla yapılabilmektedir. Örneğin:

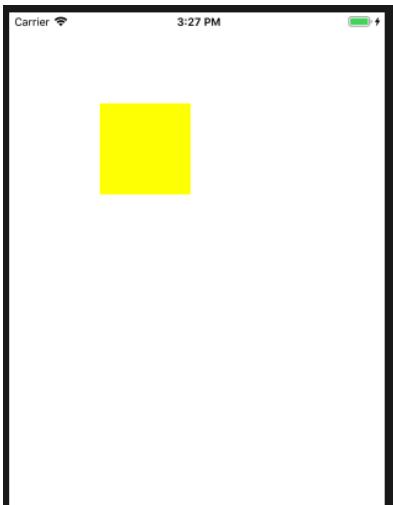
```

override func viewDidLoad()
{
    super.viewDidLoad()

    let myview = UIView(frame: CGRect(x: 100, y: 100, width: 100, height: 100))
    myview.backgroundColor = UIColor.yellow
    self.view.addSubview(myview)
}

```

Görüntü aşağıdaki gibi olacaktır:



UIView sınıfının subviews isimli property elemanı bize o view'nun alt view'larını, superview isimli property elemanı üst view'sunu bize vermektedir. Bu property elemanlar let biçimindedir. Şimdi ana view'ya iki view daha ekleyelim:

```

override func viewDidLoad()
{
    super.viewDidLoad()

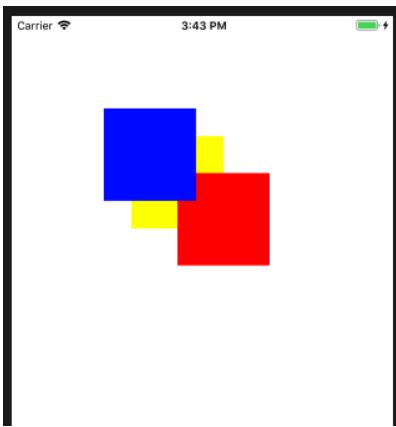
    let myview1 = UIView(frame: CGRect(x: 130, y: 130, width: 100, height: 100))
    myview1.backgroundColor = UIColor.yellow
    self.view.addSubview(myview1)

    let myview2 = UIView(frame: CGRect(x: 180, y: 170, width: 100, height: 100))
    myview2.backgroundColor = UIColor.red
    self.view.addSubview(myview2)

    let myview3 = UIView(frame: CGRect(x: 100, y: 100, width: 100, height: 100))
    myview3.backgroundColor = UIColor.blue
    self.view.addSubview(myview3)
}

```

Buradaki görüntü de şöyle olacaktır:



Aynı view'nun alt view'ları söz konusu olduğunda sonra yaratılmış olan her zaman daha yukarıda görüntülenir. (Çünkü Cocoa tarafından daha sonra çizilmektedir.) Biz alt view'ların istediğimiz sırada çizilmesini sağlamak için eklenme sırasını değiştirmeliyiz. UIView sınıfının insertSubview isimli metodları view'yu belli bir view'nun önüne ya da arkasına insert edebilmektedir. insertSubview(_:aboveSubview:) metodu view'yu diğer alt view'nun sonrasında (yani yukarısına) insertSubview(_:belowSubview:) metodu da öncesine yerleştirmektedir. Örneğin:

```

override func viewDidLoad()
{
    super.viewDidLoad()

    let myview1 = UIView(frame: CGRect(x: 130, y: 130, width: 100, height: 100))
    myview1.backgroundColor = UIColor.yellow

    let myview2 = UIView(frame: CGRect(x: 180, y: 170, width: 100, height: 100))
    myview2.backgroundColor = UIColor.red

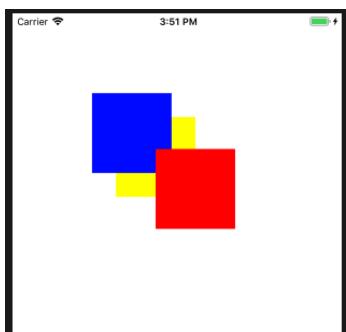
    let myview3 = UIView(frame: CGRect(x: 100, y: 100, width: 100, height: 100))
    myview3.backgroundColor = UIColor.blue

    self.view.addSubview(myview1)
    self.view.addSubview(myview2)
    self.view.insertSubview(myview3, belowSubview: myview2)
}

```

}

Görüntü şöyle olacaktır:



Bir alt view'nun da alt view'ları olabilir. Alt view'ların orijinleri kendi üst view'larının sol-üst köşesidir. Yakın bir geçmişe kadar bir alt view üst view'sunun sınırları dışına çıkamıyordu. Fakat artık çıkabilmektedir. Alt view'lar her zaman üst view'ların çizim olarak yukarısında görüntülenmektedir. Örneğin:

```
override func viewDidLoad()
{
    super.viewDidLoad()

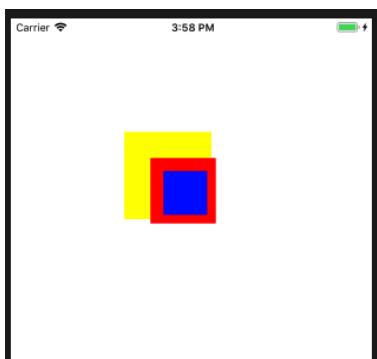
    let myview1 = UIView(frame: CGRect(x: 130, y: 130, width: 100, height: 100))
    myview1.backgroundColor = UIColor.yellow

    let myview2 = UIView(frame: CGRect(x: 30, y: 30, width: 75, height: 75))
    myview2.backgroundColor = UIColor.red

    let myview3 = UIView(frame: CGRect(x: 15, y: 15, width: 50, height: 50))
    myview3.backgroundColor = UIColor.blue

    self.view.addSubview(myview1)
    myview1.addSubview(myview2)
    myview2.addSubview(myview3)
}
```

Görüntü şu biçimde olacaktır:



View'lar birbirlerinden bağımsız çizim mekanizmasına sahiptir. Yani her view kendi çizimini bağımsız olarak kendisi yapabilmektedir.

Cocoa'nın standart birtakım görsel öğeleri (UIButton, UITextField gibi) aslında başka görsel öğeler kullanılarak da yazılmış olabilmektedir. Örneğin bir UIButton nesnesi kendi içerisinde alt view olarak UILabel ve UIImageView kullanmaktadır. Yani bazı kontroller bu bakımdan kompozit yapıdadır.

UIView sınıfının diğer önemli property'leri ve sınıfları ileride ele alınacaktır.

View Sınıfında Touch Event'lerinin İşlenmesi

Anımsanacağı gibi UIView sınıfı UIResponder sınıfından türetilmiş durumdadır. Gerçekten de event işlemleri aslında UIResponder sınıfı tarafından yapılmaktadır. Başka bir deyişle sistem bir event gerçekleştiğinde UIResponder sınıfının çeşitli metodlarını çağırır. Bu metodlar view sınıflarında override edilirse view sınıfı bunları işleme şansına sahip olmaktadır. Cocoa Touch'ta mesaj işlemesi ayrıntıları olan bir konudur. Biz burada yalnızca şimdilik bir fikir oluşması için touch event'lerine gözdireceğiz.

Mobil aygıtlarda uzun süredir "multi touch" işlemler yapılmaktadır. Multitouch aynı anda birden fazla parmağın ekran'a tıklanmasıyla oluşturulmaktadır. Oysa masaüstü sistemlerde genellikle tek bir fare bulunmaktadır. Bu yönyle mobil cihazlar masaüstü sistemlerden biraz farklılık göstermektedir.

UIResponder sınıfında üç touch metodu vardır:

```
func touchesBegan(Set<UITouch>, with: UIEvent?)  
func touchesMoved(Set<UITouch>, with: UIEvent?)  
func touchesEnded(Set<UITouch>, with: UIEvent?)
```

Metotların birinci parametreleri oluşan touch olayına ilişkin ayrıntıları içeren UITouch türünden kümedir. Aynı anda birden fazla parmakla touch işlemi yapılabildiği için burada bu nesneler bir collection olarak verilmektedir. with parametreleri oluşan event hakkında bilgiler vermektedir. touchesBegan metodu touch olayı başladığında, touchesEnden metodu da bittiğinde çağrılmaktadır. touchesMoved ise parmak basılı halde hareket ettirildiğinde çağrılr. Bu olaylar diğer framework'lerdeki MouseDown, MouseUp ve MouseMove mesajlarına benzetilebilir. Örneğin biz UIView sınıfından MyView sınıfını türetip bu sınıfta yukarıdaki üç metodu override ederek söz konusu olaylar kendi view'muz üzerinde gerçekleştiğinde bu override etmiş olduğumuz metodlar çalıştırılacaktır.

```
import UIKit  
  
class ViewController: UIViewController {  
  
    override func viewDidLoad()  
    {  
        super.viewDidLoad()  
  
        let myview = MyView(frame: CGRect(x: 50, y: 150, width: 150, height: 150))  
        myview.backgroundColor = UIColor.yellow  
        self.view.addSubview(myview)  
    }  
}  
  
class MyView : UIView  
{  
    required init?(coder aDecoder: NSCoder)  
    {  
        super.init(coder: aDecoder)  
    }  
  
    override init(frame rect: CGRect)  
    {
```

```

        super.init(frame: rect)
    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        print("began")
    }

    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
        print("ended")
    }

    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        print("moved")
    }
}

```

Burada en düşük seviyede touch event'lerinin işlenmesini yüzeysel olarak gördük. Bu event'lerin ayrıntıları ileride ele alınacaktır. Aslında tüm görsel öğeler bu event'leri işleyerek işlevselliklerini gerçekleştirmektedir. Ancak bazı görsel öğeler bu event'lerin sonucu olarak daha yüksek seviyeli event mekanizmaları oluşturabilmektedir.

UIControl Sınıfı

iOS'ta görsel elemanların çoğu doğrudan IView sınıfından değil, UIView sınıfından türetilmiş olan UIControl sınıfından türetilmiş durumdadır. UIControl sınıfı pek çok kontrol için bazı ortak elemanları bulundurmaktadır. Biz bu burada UIControl sınıfının önemli bazı elemanlarını tanıtabiliriz.

- Sınıfın isEnabled property'si kontrolün etkin (enabled) durumda olup olmadığını bize verir ve aynı zamanda kontrolü etin ve pasif duruma geçirir.
 - Sınıfın isSelected property'si benzer biçimde kontrolü "selected" moda geçirmekte kullanılmaktadır. Bazı kontroller "selected" moda geçtiğinde daha değişik görünürlür.
 - Yine kontrollerin diğer görüntü modu da "highlighted" moddur. isHighlighted metodu ile kontrol highlighted moda geçirilebilmektedir.
 - UIControl sınıfının en önemli elemanlarından biri addTarget isimli metottur. UIControl sınıfından türetilmiş tüm kontrol sınıflarında bu metot bulunmaktadır ve event'lere yanıt vermek için kullanılmaktadır. addTarget metodunun parametrik yapısı şöyledir:
- ```
func addTarget(_ target: Any?, action: Selector, for controlEvents: UIControl.Event)
```
- Metodun birinci parametresi bir olay gerçekleştiğinde yönlendirilecek sınıfı belirtmektedir. Bu parametre genellikle ilgili view sınıfı olur. İkinci parametre olay karşısında çağrılacak metodu belirtmektedir. Ancak bu metot Selector denilen bir türle istenmektedir. Selector içerisinde metot tutan özel sınıflardır. Her ne kadar Swift'te metodlar zaten birinci sınıf vatandaş gibi kolaylıkla metodlara parametre olarak geçiriliyorsa da maalesef Objective-C uyumu için olaylara yanıtta kullanılan metodlar Selector denilen özel nesnelerle ifade edilmektedir. İzleyen paragrafta selector kullanımı üzerinde durulacaktır. on parametre oluşan olayı belirtmektedir. Yani hangi olay gerçekleştiğinde bu selektör metodu çağrılacaktır? İşte olaylar UIControl sınıfının içerisindeki Event isimli bir yapıyla temsil edilmektedir. Bu yapının bazı static öznitelikleri bize Event nesneleri vermektedir. Bunların önemli olanları şunlardır:

touchDown

```
touchDownRepeat
touchDragInside
touchDragOutside
touchDragEnter
touchDragExit
touchUpInside
touchUpOutside
touchCancel
```

Aslında UIControl sınıfı UIResponder sınıfından gelen temel dokunma mesajlarını basitleştirerek daha kolay kullanılır hale getirmiştir. İşte buradaki bu touchXXX olayları daha basit hale getirilmiş dokunma mesajlarıdır.

Yukarıda da belirtildiği gibi selektör kavramı aslında Objective-C uyumu için Swift'te de kullanılmaktadır. Cocoa kütüphanesi Objective-C'de yazıldığından bazı durumlarda böylesi uyum koruma çabaları görülmektedir. Bir selektör metodu başına @objc özniteligi getirilerek bildirilir. Örneğin:

```
@objc func onButtonClicked(sender: UIButton)
{
 //....
}
```

Bizden selektör isteyen metodlarda biz selektörü #selector(metot ismi) biçiminde vermemeliyiz. Başka bir deyişle Selector nesneleri #selector(metot ismi) biçiminde oluşturulmaktadır. Selektör sınıfın statik metodu olabileceği gibi örnek metodu daolabilir. Örnek metodlar bir referansla, statik metodlar da sınıf ismiyle nokta operatörü kullanılarak belirtilirler. Tabii biz sınıf içerisinde doğrudan metodun ismini yazabiliriz. Bu durumda metot static olmayan bir metot olarak değerlendirilir. Şimdi biz bir UIControl nesnesine (örneğin bu UIButton olabilir) ile addAction kullanarak metot ekleyelim:

```
buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)
...
@objc func onButtonOk(sender: UIButton)
{
 //...
}
```

## UIKit'teki Önemli Kontroller

Bu bölümde UIKit içerisinde GUI arayüzüni oluşturmakta kullanılan önemli görsel öğeler açıklanacaktır. Bu görsel öğelerin hepsi doğrudan ya da dolaylı olarak UIControl sınıfından türetilmiştir. IOS programcılar bunları genellikle IB (Interface Builder) kullanarak görsel biçimde oluşturuyor olsalar da siz denemelerde bu kontrolleri hem sıfırdan kodlama yoluyla hem de IB yoluyla oluşturup test etmelisiniz.

### Etiket Kontrolü ve UILabel Sınıfı

Ekranda bir yazı oluşturmanın en kolay yolu UILabel sınıfıyla temsil edilen etiket kontrolü kullanmaktadır. Bir UILabel nesnesi IB kullanılarak görsel olarak ya da UIView sınıfından gelen init(frame:) metoduyla yaratılabilir. Metot parametre olarak CGRect almaktadır. UILabel ile yaratılan etiketin zemin rengini yine istersek UIView sınıfından gelen backgroundColor isimli UIColor türünden property ile değiştirebiliriz. Default zemin rengi transparandır. Yani arkadaki pencerenin rengi gözükecek biçimdedir. Etiketin üzerindeki yazı String? türünden text property'si ile alınıp değiştirilebilmektedir. Örneğin:

```
class ViewController: UIViewController {
 var labelName: UILabel!

 override func viewDidLoad() {
```

```

super.viewDidLoad()
// Do any additional setup after loading the view, typically from a nib.

labelName = UILabel(frame: CGRect(x: 10, y: 100, width: 200, height: 50))
labelName.backgroundColor = UIColor.yellow
labelName.text = "Kaan Aslan"
view.addSubview(labelName)
}
}

```

Etiket kontrolü için sınıfın veri elemanın UILabel! türünden alındığını dikkat ediniz. (Swift'te sınıfın seçeneksel olmayan veri elemanlarına bildirim sırasında ya da init metodunda değer atamak zorunda olduğunu anımsayınız. Bu nedenle sınıfın örnek özniteliği olan kontrollerin seçeneksel olarak bildirilmesi daha uygundur.)

Etiket üzerindeki yazının rengi UILabel sınıfındaki UIColor! türünden textColor property'si ile alınıp değiştirilebilir. Etiket highlighted moda geçirilebilir. Bunun için UILabel sınıfının UIControl sınıfından gelen Bool türden isHighlighted property'si true yapılmalıdır. Etiket highlighted moda geçirildiğinde artık yazı olarak onun highlightedTextColor property'si ile belirtilen yazı görüntülenecektir.

Etiket üzerindeki yazıyı hizalayabiliriz. Bunun için UILabel sınıfının NSTextAlignment enum'u türünden textAlignment isimli property'si kullanılmaktadır. NSTextAlignment içerisinde left, right, center ve justified elemanları bulunan bir enum türüdür. Örneğin:

```
labelName.textAlignment = .center
```

Yazı için bir gölgelendirme UILabel sınıfının shadowColor isimli property'si ile verilebilmektedir. Ancak bu işlemden önce programcının shadowOffset property'si ile gölgelendirme offset'ini belirmesi gereklidir. Örneğin:

```
labelTest.textColor = UIColor(red: 155, green: 0, blue: 25, alpha: 255)
labelTest.shadowOffset = CGSize(width: 2, height: 2)
```

Etikette görüntülenecek satır sayısı default olarak 1'dir. Ancak biz sınıfın numberOfLines isimli property'si ile bu sayıyı fazlalaştıralım. Örneğin:

```
labelTest.numberOfLines = 4
```

UILabel sınıfının lineBreakMode isimli property elemanı sığmayan yazılarla satırın nasıl kesileceğini belirlemekte kullanılır. Bu property NSLineBreakMode isimli enum türündendir. Bu enum türünün önemli elemanları şunlardır:

```
byWordWrapping
byCharWrapping
byClipping
byTruncatingHead
byTruncatingTail
byTruncatingMiddle
```

UILabel sınıfının adjustFontSizeToFitWidth isimli property'si True yapılrsa yazının tamamı gözküçük biçimde yazının fontu otomatik olarak küçültülür. minimumScaleFactor ise 0 ile 1 arasında bir değer almaktadır. Yazının en fazla sığdırma için yüzde kaç küçültüleceğini belirtir. minimumScaleFactor property'si ancak adjustFontSizeToFitWidth property'si True ise anlamlıdır. Örneğin:

```
labelTest.adjustsFontSizeToFitWidth = true
labelTest.minimumScaleFactor = 0.7
```

Burada yazının hepsini sığdırmak için istenildiği kadar değil ancak 0.7 oranında küçültme yapılması istenmiştir.

Etiket üzerindeki yazının fontu UILabel sınıfındaki UIFont türünden font isimli property ile değiştirilebilir. Örneğin:

```
labelTest.font = UIFont(name: "Arial", size: 30)
```

Pekiyi etiket üzerindeki bir yazının bir bölümü bold, bir bölümü italic vs. olabilir mi? Yani biz yazıyı bu anlamda formatlayabilir miyiz? İşte Cocoa'da yazı formatlamak için NSAttributedString isimli bir sınıf kullanılmaktadır. UILabel sınıfının AttributedText isimli property'si NSAttributedString türündendir.

NSAttributedString nesnesi kendi içersinde bir yazı ve bir sözlük nesnesi içermektedir. Nesne sınıfın init(string:attributes:) metodu ile yaratılır. Metodun birinci parametresi formatlanacak yazıyı belirtir. İkinci parametresi ise bir sözlük nesnesidir. Öyle ki bu sözlük nesnesinin anhtarları NSAttributedString.Key isimli yapı türünden, değeri ise anahtarla uyumlu türdendir. Teknik anlamda buradaki sözlük [NSAttributedString.Key: Any] türündendir. Sözlüğün anahtarlarının aynı türden olduğuna ama değerinin anahtara göre değişik türlerde olduğuna (bu durumda mecburen değer Any türü olarak belirtilmiştir) dikkat ediniz. Örneğin:

```
let attr = [NSAttributedString.Key.font: UIFont(name: "Arial", size: 40)]
labelTest.attributedText = NSAttributedString(string: "Ankara", attributes: attr)
```

Biz sözlüğümüzü çeşitli özelliklerle doldurup NSAttributedString metoduna parametre olarak geçiririz. Böylece yazının çeşitli kısımlarının değişik bir biçimde görüntülenmesini sağlayabiliyoruz. Ancak yazı görsel özellikleri (fontu gibi, rengi gibi) zaten ilgili sınıfların bazı property'leriyle değiştirilebilmektedir. Ancak örneğin yazıyı altı çizili (underline) yapmak için bir property yoktur. Bunu NSAttributedString ile gerçekleştirebiliriz.

### Tek Satırlı Edit Kontrolü ve UITextField Sınıfı

Kullanıcıdan yazı almak için CocoaTouch'ta UITextField isimli sınıf kullanılmaktadır. Bu sınıf tek satırlı edit kontrolü oluşturmaktadır.

UITextField nesnesini programlama yoluyla yine UIView sınıfından gelen init(frame:) metodu ile CGRect vererek yaratabiliriz. Yine biz edit alanın zemin rengini UIView sınıfından gelen backgroundColor property'si ile değiştirebiliriz. Yine tipki etiket kontrolünde olduğu gibi edit kontrolünde de text property'si edit alanındaki yazıyı elde etmekte ve set etmeye kullanılır. Yine yazının rengi textColor property'si ile ayarlanabilmektedir.

UITextField sınıfının placeHolder isimli String? türünden property elemanı henüz edit alanına yazı girilmemişken görüntülenecek ipucu yazısını belirtir. Örneğin:

```
textFieldTest.placeholder = "Adınızı soyadınızı giriniz"
```

Ayrıca bir de attributedPlaceHolder isimli NSAttributedString alan bir ipucu yazısı da bulunmaktadır.

Edit alanının zemin rengine bir resim de girilebilir. Bunun için sınıfın background property'si kullanılır. Bu property UIImage? türündendir. Benzer biçimde kontrol disable durumdayken disabledBackground property'si ile ayrı bir resim gösterilebilmektedir.

Sınıfın borderStyle isimli property elemanı UITextFiled.BorderStyle isimi enum türündendir. Bu enum türünün elemanları da şunlardır:

none  
line  
bezel

roundedRect

Sınıfın clearViewButton isimli property elemanı ViewMode isimli enum türündendir. Bu enum türünün elemanları şunlardır:

```
never
whileEditing
unlessEditing
always
```

Bu property edit alanında bu enum değerine bağlı olarak küçük bir x düğmesinin çıkışını sağlar.

Tıpkı IUILabel sınıfında olduğu gibi UITextField sınıfında da bir font property'si bulunmaktadır.

UITextField sınıfının allowsEditingTextAttributes isimli property elemanı edit alanı içerisinde bir seçim yapıldığında yukarıda çıkan çubukta Bold/Italic gibi öğelerin görüntülenmesini sağlar.

UITextField sınıfının textAlignment property'si tamamen UILabel sınıfındaki gibidir.

UITextField nesnesi çeşitli bazı eylemlerde karşı event oluşturabilmektedir. Yine bu olayların işlenmesi UIControl sınıfından gelen addTarget metoduyla yapılmaktadır. Tabii IB'de bunun görsel olarak yapılması çok daha kolaydır. Şüphesiz en çok kullanılan UITextField event'i editingChanged isimli event'tir. Bu event edit alanındaki yazdı herhangi bir değişiklik olduğunda tetiklenmektedir. Örneğin:

```
textFieldTest.addTarget(self, action: #selector(onMyEditingChange), for: .editingChanged)
..
@objc func onMyEditingChange(sender: Any)
{
 //...
}
```

### Çok Satırlı Edit Kontrolü ve UITextView Sınıfı

UIKit kütüphanesinde çok satırlı edit kontorlu UITextView isimli sınıfta temsil edilmiştir. Bu sınıf türünden nesne yine diğerlerinde olduğu gibi UIView sınıfından gelen init(frame:) metoduyla yaratılabilir. Bu sınıf UIControl sınıfından türetilmemiştir.

UITextView sınıfında textAlinment, textColor ve font property'leri diğer kontrollerdeki gibidir. Edit alanı içerisindeki bilgi yine text property'si yoluyla alınıp set edilebilir. Sınıfın isEditable property'si edit alanının readonly olup olmayacağıni belirlemekte kullanılır. Benzer biçimde isSelectable property'si de edit alanı içerisindeki yazının seçilip seçilemeyeceğini belirlemekte kullanılır.

### Düğmeler ve UIButton Sınıfı

Düğmeler şüphesiz en çok kullanılan UI nesnelerindendir. UIButton sınıfıyla temsil edilmişlerdir. UIButton sınıfı da UIControl sınıfından türetilmiş durumdadır.

Programlama yoluyla düğme yaratırken UIButton sınıfının init metotları kullanılmaktadır. Nesne init(frame:) ya da init(type:) metoduyla düğme türü belirtilerek yaratılabilir. Örneğin:

```
buttonOk = UIButton(type: .system)
```

Default düğme .custom türüyle yaratılmaktadır.

Düğme yaratılırken verilen type şunlardan biri olabilir:

```
custom
system
detailedDisclosure
infoLight
infoDark
contactAdd
plain
roundedRect
```

Düğmenin üzerindeki yazı setTitle metoduyla değiştirilebilir. Bu metodun parametrik yapısı şöyledir:

```
func setTitle(_ title: String?, for state: UIControl.State)
```

Metodun birinci parametresi düğme üzerinde görüntülenecek yazıyı, ikinci parametresi ise bu yazının düğmenin hangi durumunda görüntüleneceğini belirtir.

UIButton sınıfının setTitleColor metodu ile düğme üzerindeki yazının rengi değiştirilebilir. Metodun parametrik yapısı şöyledir:

```
setTitleColor(_:for:)
```

Bir düğmenin zeminine bir resim yerleştirebiliriz. Bunun için UIButton sınıfının setBackgroundImage(\_:for:) metodunu kullanılmaktadır.

Diğer kontrollerde olduğu gibi UIButton kontrolü de UIControl sınıfından gelen isEnabled property'si yoluyla "disabled" ya da "enabled" durum getirilebilir. Yine UIControl sınıfından gelen addTarget metodu ile biz kontrole tıklandığından belli bir işlem yapabiliriz.

## Switch Kontrolü ve UISwitch Sınıfı

UISwitch kontrolü bir çeşit on/off kontrolü görevini yapmaktadır. UISwitch sınıfıyla temsil edilmiştir. UISwitch sınıfı UIControl sınıfından türetilmiş durumdadır. Kontrol init(frame:) metodu ile yaratılabilir. Örneğin:

```
uiswitch = UISwitch(frame: CGRect(x: 10, y: 100, width: 100, height: 50))
```

Sınıfın isOn property elemanı programlama yoluyla switch'in açık mı yoksa kapalı olduğunu true/false olarak set ve get etmekte kullanılır. Örneğin:

```
import UIKit

class ViewController: UIViewController {
 var uiswitch: UISwitch!
 var buttonOk: UIButton!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 uiswitch = UISwitch(frame: CGRect(x: 10, y: 100, width: 100, height: 50))
 uiswitch.isOn = true
 self.view.addSubview(uiswitch)
```

```

 buttonOk = UIButton(type: .system)
 buttonOk.frame = CGRect(x: 10, y: 200, width: 100, height: 100)
 buttonOk.setTitle("Ok", for: .normal)
 buttonOk.backgroundColor = .yellow
 buttonOk.addTarget(self, action: #selector(onButtonClick), for: .touchUpInside)
 self.view.addSubview(buttonOk)
 }

@objc
func onButtonClick()
{
 print(self.uiswitch.isOn ? "Açık" : "Kapalı")
}
}

```

Carrier 2:10 PM

Switch State: On



## Slider Kontrolü ve UISlider Sınıfı

Slider belli bir olgu için düzey belirlemekte kullanılmaktadır. Kontrol UISlider sınıfı ile temsil edilmektedir. Bu sınıf da UIControl sınıfından türetilmiş durumdadır. Nesne yine UIView sınıfından gelen init(frame:) metodu ile yaratılabilir. Örneğin:

```

slider = UISlider(frame: CGRect(x: 10, y: 100, width: 200, height: 50))
self.view.addSubview(slider)

```

Slider'ın default durumda en düşük değeri 0, en yüksek 1'dir. Yürüteç konumlandırıldığındanFloat tründne value property'si oransal değerini alır. Biz istersek value değerini programlama yoluyla setValue metodunu ile değiştirebiliriz. Slider'ın en küçük ve en büyük değerleri istenirse minValue ve maxValue property'leri ile alınıp değiştirilebilir.

Slider'ın yürütücü hareket ettirildiğinde valueChanged isimi event oluşmaktadır. Biz de bunu programlama yoluyla addTarget metodunu ile set edebiliriz. valueChanged event'i yürütücü her hareket ettirildiğinde tetiklenmektedir. Ancak sınıfın isContinuous property'si false yapılrsa yalnızca yürütücü bırakıldığından bu event tetiklenir. Ancak value property'si programlama yoluyla değiştirildiğinde valueChanged event'i tetiklenmemektedir.

Örneğin:

```

import UIKit

class ViewController: UIViewController {
 var slider: UISlider!
 var labelValue: UILabel!
 var textField: UITextField!
 var buttonOk: UIButton!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 slider = UISlider(frame: CGRect(x: 10, y: 100, width: 390, height: 50))
 slider.maximumValue = 100
 slider.addTarget(self, action: #selector(onValueChanged), for: .valueChanged)
 //slider.isContinuous = false

 self.view.addSubview(slider)

 labelValue = UILabel(frame: CGRect(x: 190, y: 70, width: 100, height: 30))
 labelValue.text = "0"
 self.view.addSubview(labelValue)

 textField = UITextField(frame: CGRect(x: 10, y: 200, width: 100, height: 30))
 self.view.addSubview(textField)
 textField.borderStyle = .roundedRect

 buttonOk = UIButton(type: .system)
 buttonOk.frame = CGRect(x: 10, y: 250, width: 100, height: 100)
 buttonOk.setTitle("Ok", for: .normal)
 buttonOk.backgroundColor = .yellow
 buttonOk.addTarget(self, action: #selector(onButtonClick), for: .touchUpInside)
 self.view.addSubview(buttonOk)
 }

 @objc
 func onValueChanged()
 {
 labelValue.text = String(Int(round(slider.value)))
 }

 @objc
 func onButtonClick()
 {
 if textField.text != ""
 slider.value = Float(textField.text!)!
 labelValue.text = String(Int(round(slider.value)))
 }
}

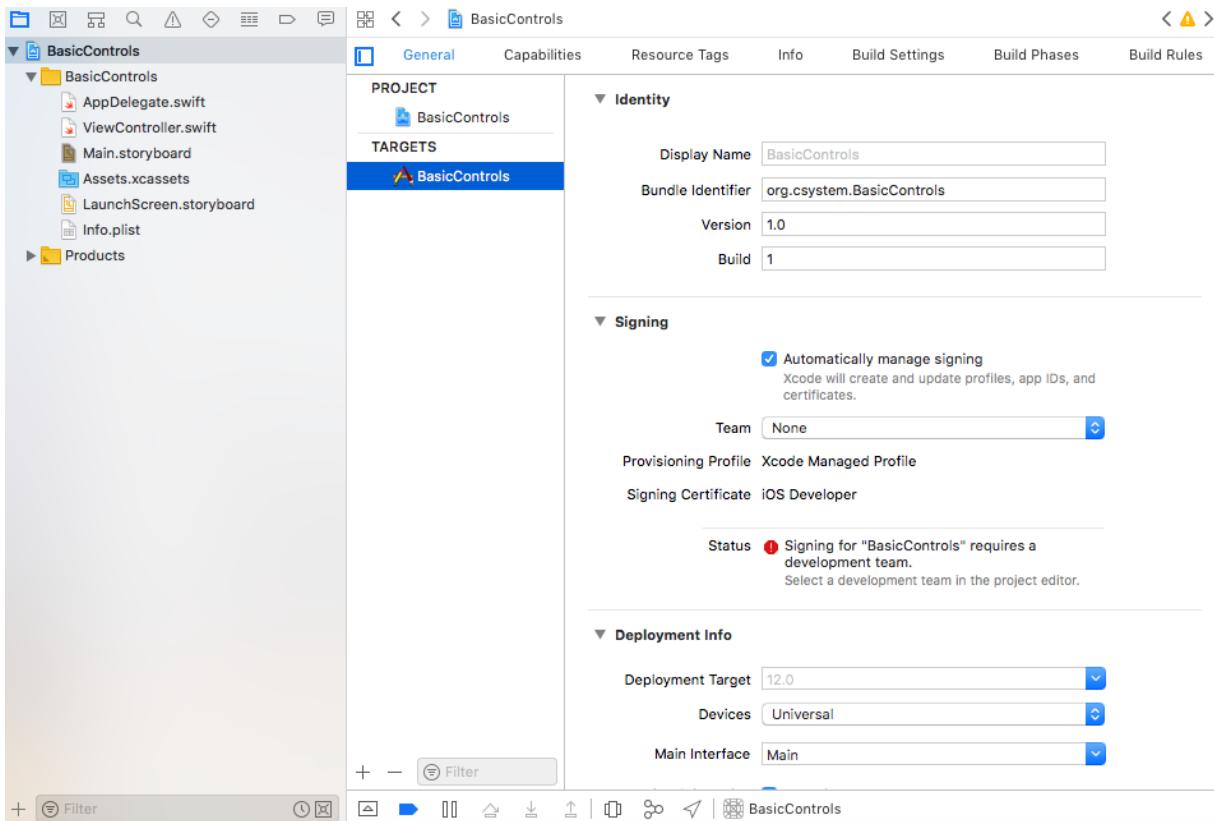
```

## IOS Programlarının Gerçek Aygıta Yüklenmesi

Eskiden IOS programları test amaçlı olsa bile yalnızca AppStore'dan yüklenebiliyordu. (AppStore program yerleştirmek için ise ücret ödemek gerekiyordu.) Ancak artık biz IOS programlarını gerçek aygıta yükleyip test edebilmekteyiz. Bu işlem adım adım şöyle sağlanmaktadır:

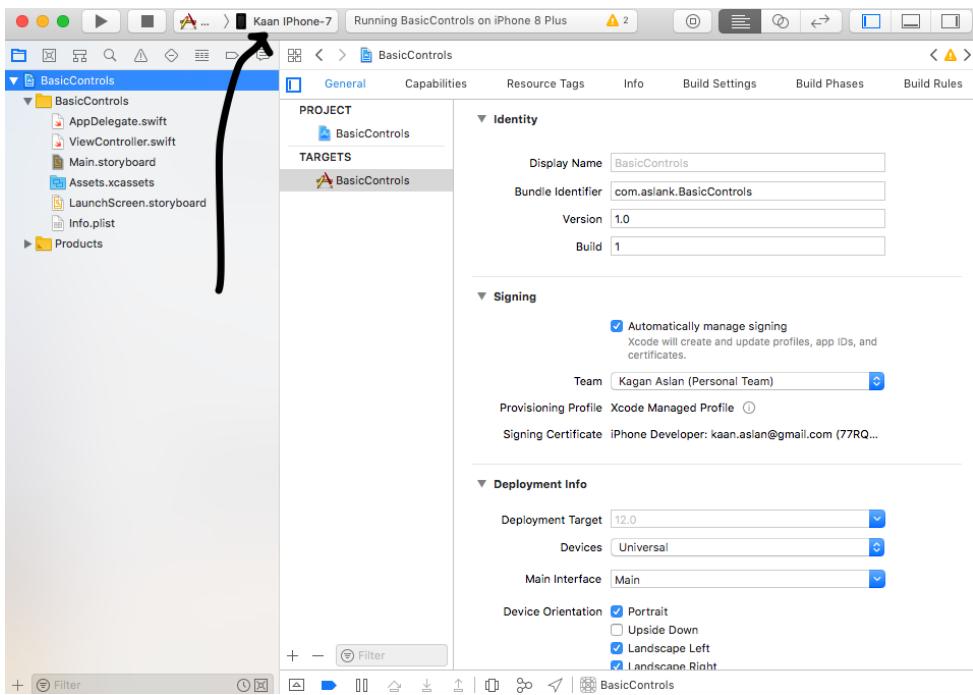
1) IOS aygıtı geliştirmenin yapıldığı bilgisayara USB portundan bağlanır. (Aslında aynı işlem wireless network ile de yapılabilmektedir.)

2) Project Navigator'da projenin üzerine gelinerek Project Target seçilir:



3) Bundle Identifier'daki ismin AppStore genelinde tek (unique) olması gerekmektedir. Buradaki isim ters ismiyle (reverse domain name) ile başlatılmalı ve istenildiği gibi devam ettirilmelidir. Bundan sonra "Signing" bölümünde "Team" seçilmesi gerekmektedir. Ancak "Team" seçmeden önce yüklü bir hesabın bulunması gerekmektedir. Buradaki hesap paralı AppStore hesabı değildir. Herhangi bir AppleId hesabı olabilir. Bunun için öncelikle "Add an Account" yapmak gerekebilir.

4) Team seçildikten sonra hedef aygit ayarlanmasıdır. Bunun için aşağıdaki düğme kullanılabilir:



5) Artık proje çalıştırıldığında otomatik olarak hedef paket aygıta aktarılacaktır. Bu işlem sırasında bizden Mac bilgisayarımızın parolası istenmektedir. Burada "Her Zaman İzin Ver" seçilmelidir.

### **UISegmentedControl Kullanımı**

Bu görsel eleman bir çwşit radio button görevini yapmaktadır. Kontrolde tab'lar vardır. Bu tablardan yalnızca biri seçilebilmektedir. Kontrolün kullanımı şöyledir:

1) UISegmentedControl sınıfı türünden bir nesne yaratılır. Örneğin:

```
segmentedControl = UISegmentedControl(frame: CGRect(x: 10, y: 50, width: 200, height: 30))
segmentedControl.backgroundColor = .yellow
self.view.addSubview(segmentedControl)
```

2) Kontrole segmentler insertSegment(withTitle:at:animated:) metoduyla eklenir. Örneğin:

```
segmentedControl.insertSegment(withTitle: "Apple", at: 0, animated: true)
segmentedControl.insertSegment(withTitle: "Banana", at: 1, animated: true)
segmentedControl.insertSegment(withTitle: "Apricot", at: 2, animated: true)
```

3) Seçilmiş olan segmentin indeksi selectedSegmentIndex isimli metotla elde edilir. Eğer hiçbir segment seçili değilse bu property bize -1 verir.

4) Kontrolün baz property'leri bazı ayarlamaları yapmak için kullanılmaktadır. isMomentary isimli Bool türden property kontrolü kısa sürecekli moda getirir. Normal olarak segmentler aynı boyuttadır. Ancak biz istersek sınıfın setWidth(\_:forSegmentAt:) metoduyla belli bir segmentin genişliğini ayarlayabiliyoruz. Ya da herhangi bir segmentin genişliğini widthForSegment(at:) metodu ile elde edebiliriz. Örneğin:

```
segmentedControl.setWidth(100, forSegmentAt: 0)
```

Sınıfın apportionsSegmentWidthsByContent isimi Bool türden property'si yazıların uzunluğuna göre segmentlerin genişliğinin otomatik ayarlanması amacıyla kullanılmaktadır. Kontrolün tintColor isimli property elemanı segment seçildiğindeki seçim rengini ayarlamakta kullanılır. Aslında bu property UIView sınıfından gelmektedir. Yine sınıfın

`removeSegment(a:animated:)` isimli metodu belli bir segmenti silmek için, `removeAllSegments` isimli metodu da tüm segmentleri silmek için kullanılmaktadır. `numberOfSegments` isimli read-only property bize segment sayısını verir.

Sınıfın `valueChanged` isimli bir event'i her segment seçildiğinde tetiklenmektedir.

Kontrol kullanımına ilişkin örnek şöyledir:

```
import UIKit

class ViewController: UIViewController {

 var segmentedControl: UISegmentedControl!
 var buttonOk: UIButton!

 @IBOutlet weak var segmentedControlIB: UISegmentedControl!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 segmentedControl = UISegmentedControl(frame: CGRect(x: 10, y: 50, width: 400, height: 30))
 segmentedControl.backgroundColor = .yellow
 segmentedControl.insertSegment(withTitle: "Pineapple", at: 0, animated: true)
 segmentedControl.insertSegment(withTitle: "Banana", at: 1, animated: true)
 segmentedControl.insertSegment(withTitle: "Apricot", at: 2, animated: true)
 segmentedControl.insertSegment(withTitle: "Watermelon", at: 3, animated: true)
 //segmentedControl.isMomentary = true
 segmentedControl.addTarget(self, action: #selector(onValueChanged), for: .valueChanged)
 //segmentedControl.setWidth(100, forSegmentAt: 0)
 segmentedControl.apportionsSegmentWidthsByContent = true
 self.view.addSubview(segmentedControl)

 buttonOk = UIButton(frame: CGRect(x: 10, y: 100, width: 100, height: 100))
 buttonOk.setTitle("Ok", for: .normal)
 buttonOk.backgroundColor = .yellow
 buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)
 buttonOk.setTitleColor(.black, for: .normal)

 self.view.addSubview(buttonOk)
 }

 @objc func onButtonOk(_ sender: Any)
 {
 print(segmentedControl.selectedSegmentIndex)
 }

 @objc func onValueChanged(_ sender: Any)
 {
 print(segmentedControl.selectedSegmentIndex)
 }
}
```

## UIStepper Kullanımı

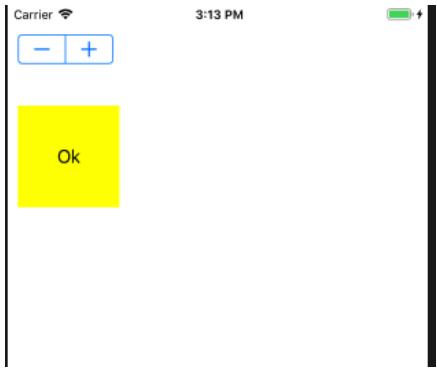
UIStepper diğer platformlardaki up/down counter'a benzemektedir. Kontrol şöyle kullanılmaktadır:

1) UIStepper türünden bir nesne yaratılır. Örneğin:

```
stepper = UIStepper(frame: CGRect(x: 10, y: 30, width: 100, height: 30))
self.view.addSubview(stepper)

buttonOk = UIButton(frame: CGRect(x: 10, y: 100, width: 100, height: 100))
buttonOk.setTitle("Ok", for: .normal)
buttonOk.backgroundColor = .yellow
buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)
buttonOk.setTitleColor(.black, for: .normal)

self.view.addSubview(buttonOk)
```



2) Sınıfın value isimli property'si o andaki sayacın değerini alıp set etmekte kullanılır. Sınıfın isContinuous isimli Bool türden property'si "+" ya da "-" simbolünde parmak basılı tutulduğunda sürekli artım ya da eksiltim (typematic) uygulamakta kullanılır. Bu property default olarak true durumdadır. Sınıfın minValue, maxValue ve stepValue property'leri sırasıyla sayacın alabileceği en küçük değeri, en büyük değeri ve adım değerini belirtir. Sayaç bu değerlerin dışında olamaz. Ancak wraps property'si true ise (default durumu false biçimdedir) sarmalama işlemi yapılır.

3) UIStepper kontrolü için de valueChanged event'i işlenebilir. Böylece değer her değiştiğinde değişmiş olan değer bir yerde görüntülenebilmektedir.

Aşağıda UIStepper kullanımına ilişkin bir örnek görülmektedir:

```
// ViewController.swift
// BasicControls
//
// Created by Kaan Aslan on 20.10.2018.
// Copyright © 2018 CSD. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

 var stepper: UIStepper!
 var buttonOk: UIButton!
 var stepperLabel: UILabel!

 override func viewDidLoad()
 {
```

```

super.viewDidLoad()

stepper = UISStepper(frame: CGRect(x: 10, y: 40, width: 100, height: 30))
stepper.isContinuous = true
stepper.addTarget(self, action: #selector(onValueChanged), for: .valueChanged)
self.view.addSubview(stepper)

stepperLabel = UILabel(frame: CGRect(x: 115, y: 40, width: 100, height: 30))
stepperLabel.text = "0"
self.view.addSubview(stepperLabel)

buttonOk = UIButton(frame: CGRect(x: 10, y: 100, width: 100, height: 100))
buttonOk.setTitle("Ok", for: .normal)
buttonOk.backgroundColor = .yellow
buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)
buttonOk.setTitleColor(.black, for: .normal)

self.view.addSubview(buttonOk)
}

@objc func onButtonOk(_ sender: Any)
{
 print(stepper.value)
}

@objc func onValueChanged(_ sender: Any)
{
 stepperLabel.text = String(Int(stepper.value))
}
}

```

## UIProgressView Kullanımı

UIProgressView kontrolü bir sürecin ne süredir devam ettiğini ve sonlanması için ne kadar görelî zaman gerektiğini görüntülemek amacıyla kullanılmaktadır. Kontrolün kullanımı şöyledir:

1) UIProgressView nesnesi yaratılarak konumlandırılır. Örneğin:

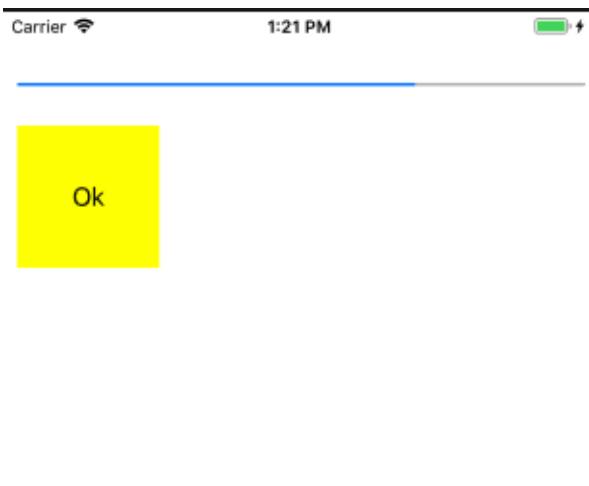
```

progressView = UIProgressView(frame: CGRect(x: 10, y: 50, width: 400, height: 30))
self.view.addSubview(progressView)

```

2) ProgressView sınıfının progress isimli Float türden property elemanı ile biz çubuğuun ne kadarının dolu olacağını belirleyebiliriz ve doluluk değerini alabiliriz. Buradaki minimum değer 0, maksimum değer 1'dir ve bu değerler değiştirilememektedir. Örneğin:

```
progressView.progress = 0.70
```



Dolum rengi UIView sınıfından gelen tintColor property'si ile değiştirilebilir. Şüphesiz dolumun oransal olarak artırılması başka bir süreç içerisinde ara ara yapılmalıdır. Örneğin bir kopyalama işleminde kopyalacak öğelerin sayısı ile bir orantı kurularak progress değeri azar azar artırılabilir.

### **UIImageView Kullanımı**

Bu kontrol diğer framework'lerde "PictureBox" kontrolüne benzetilebilir. Kontrolün amacı bir resim göstermektir. Resimler UIImage isimli sınıfla temsil edilmişlerdir. UIImage sınıfı ana pek çok dosya formatını desteklemektedir. UIImageView bizden görüntülenecek resmi UIImage nesnesi olarak istemektedir. UIImage kontrolü şöyle kullanılır:

1) Öncelikle görüntülenecek resim bir UIImage nesnesi biçiminde oluşturulur. UIImage sınıfının init metodlarında ilgili resmi belirterek nesneyi oluşturabiliyoruz. UIImage sınıfının pek çok init metodu olsa da en önemli ikisi şunlardır:

```
init?(named: String)
init?(contentsOfFile path: String)
```

Birinci metot bizden bundle'daki resim ismini istemektedir. Resimler XCode'da Assets kısmına aktarıldıktan sonra bu isim named etiketli init ile kullanılarak yükleme yapılmaktadır. Buradaki isimde dosya uzantısı belirtmeye de gerek yoktur. Halbuki ikinci metotta biz resim dosyasının (bu dosya değişik formatlara sahip olabilir) yol ifadesini vererek resmi yükleriz. Örneğin:

```
let image = UIImage(named: "AbbeyRoad")
```

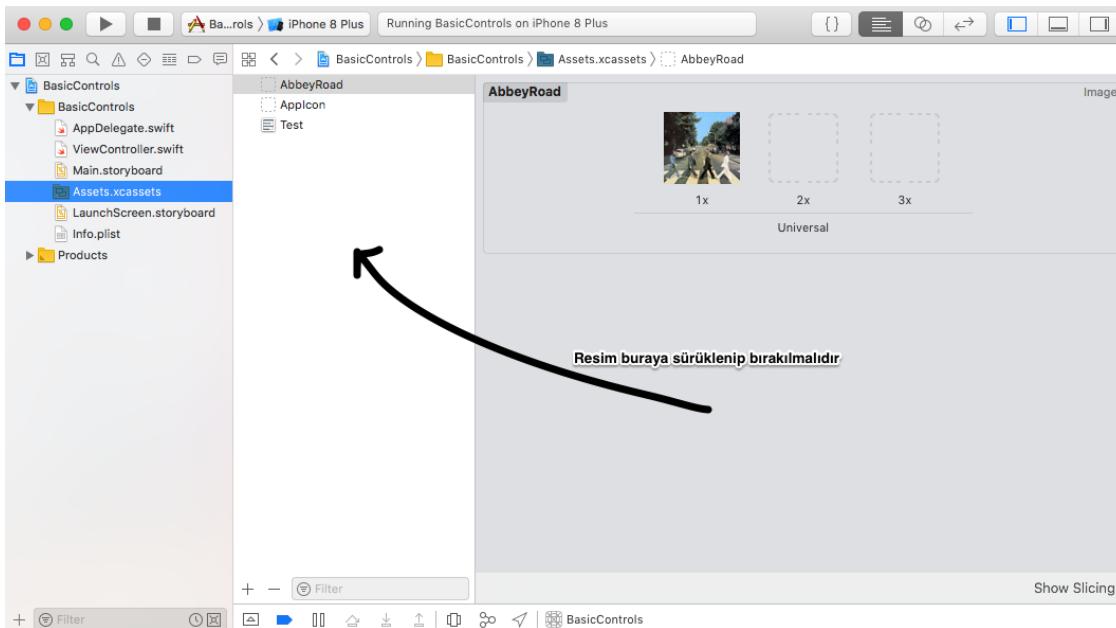
2) UIImageView nesnesi aşağıdaki yaratılır:

```
imageView = UIImageView(frame: CGRect(x: 10, y: 20, width: 400, height: 400))
```

3) Yaratılan UIImageView nesnesinin image isimli property elemanına UIImage nesnesi atanır. Örneğin:

```
imageView = UIImageView(frame: CGRect(x: 10, y: 20, width: 400, height: 400))
imageView.backgroundColor = .yellow
imageView.image = UIImage(named: "AbbeyRoad")
```

Burada UIImage nesnesinin resim olarak "bundle'daki isimle" yaratıldığına dikkat ediniz. Bunun için önce resmin XCode projesindeki Assets.xcassettes dizinine sürükleyip bırakılması gerekmektedir.



4) Bundle'a resim yerleştirirken orada 1x, 2x ve 3x biçiminde slotların olduğuna dikkat ediniz. iPhone ve iPad cihazları tarihsel olarak şimdiye kadar üç farklı çözünürlükle piyasaya sürülmüştür. Bunlar "normal", "retina" ve "süper retina" çözünürlükleridir. İşte bu çözünürlükler burada 1x, 2x ve 3x slotlarıyla ifade edilmektedir. Programcının da her resim için üç ayrı çözünürlükte resim üretmesi ve bu slotlara yerlestirmesi tavsiye edilmektedir. Uygulama AppStore'e yerleştirilecekse bu slotların dolu olması gereklidir. Tabii biz aynı resmi bu üç slota da yerlestirebiliriz. Ancak arzu edilen durum bu değildir.

5) Gösterilecek resim kontrolün çerçevisinden büyü ya da küçük olabilir. Bu durumda bir hizalama ve boyutlandırma söz konusu olacaktır. Bunun için view sınıfından gelen contentMode isimli property kullanılır. Bu property UIView.contentMode isimli bir enum türündendir. Bu enum türünün elemanları şunlardır:

```
scaleToFill
scaleAspectFit
scaleAspectFill
center
top
bottom
left
right
topLeft
topRight
bottomLeft
bottomRight
```

Kontrol yaratıldığında contentMode default olarak scaleToFit biçimindedir. Aşağıda UIImageView yaratımına ve kullanıma ilişkin bir örnek görüyorsunuz:

```
import UIKit

class ViewController: UIViewController {
 var imageView: UIImageView!

 override func viewDidLoad()
 {
 super.viewDidLoad()
```

```

 imageView = UIImageView(frame: CGRect(x: 10, y: 20, width: 400, height: 400))
 imageView.backgroundColor = .yellow
 imageView.image = UIImage(contentsOfFile: "/Users/KaanAslan/Dropbox/Shared/Kurslar/Swift-
IOS-Nisan-2018/Src/BasicControls/BasicControls/AbbeyRoad.jpg")
 imageView.contentMode = .center

 self.view.addSubview(imageView)
 }
}

```

## UIPickerView Kullanımı

Bu kontrol tekerlekSEL bir görünümle seçim yapmak amacıyla kullanılmaktadır. Bu tekerlekLere "component" denilmektedir. Kontrol birden fazla "component"ti içerebilmektedir. UIPickerView kullanımı biraz zahmetlidir. Çünkü kullanım sırasında bir sınıfın oluşturulması (ya da oluşturulmuş mevcut bir sınıf da kullanılabilir) Bu sınıfın iki protokolÜ desteklemesi gerekmektedir. Kontrolün kullanımı sırasıyla şu adımlardan geçilerek yapılmaktadır:

1) Öncelikle bir sınıf yaratıp UIPickerViewDelegate ve UIPickerViewDataSource isimli protokollerİ bu sınıfTa desteklememiz gereklİ. Aslında bir sınıf yaratmak yerine zaten yaratılmış olan ViewController sınıfı da bu amaçla kullanılabilir. Örneğin:

```

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }
 //...
}

```

2) Şimdi bu iki protokolÜ destekledeğimizden zorunlu olarak bulundurmamız gereken bazı metotları yazacağz. Bunlardan ilki kontrolümüzde kaç tekerleğin (component'in) bulunacağını belirten aşağıdaki metottur:

```
func numberOfComponents(in pickerView: UIPickerView) -> Int
```

Bu metot UIPickerViewDataSource protokolünden gelmektedir. Bu metotta programcı tekerlek sayısına geri dönmeli dir. Örneğin:

```
func numberOfComponents(in pickerView: UIPickerView) -> Int
{
 return 1
}
```

Burada kontrolümüzde bir tane tekerlek bulunacaktır. Metodun parametresi hangi UIPickerView nesnesi için bu istekte bulunulduğunu belirtmektedir.

Bundan sonra aşağıdaki metodu yazarak tekerleklerdeki eleman sayısını belirlememiz gereklİ. Bu metot ise UIPickerViewDelegate protolünden gelmektedir

```
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
```

Göründüğü gibi metdun iki parametresi vardır. Birinci parametre bu metodun hangi UIPickerView nesnesi için çağrıldığını belirtir. İkinci parametre ise tekerlek numarasını belirtmektedir. Örneğin kontrolümüzde üç ayrı tekerlek varsa bu metot kontrol tarafından üç kez 0, 1 ve 2 tekerlek numaralarıyla çağrılacaktır. Örneğimizde tek tekerlek olduğu

İçin burada biz tek bir değerle geri döneriz. Tekerleklerde gösterilecek yazıların bir dizide olduğunu düşünelim. Bu durumda biz bu dizinin uzunlu ile geri donebiliriz:

```
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
{
 return fruits.count
}
```

Buradaki fruits dizisi sınıf içerisinde bildirilmiştir.

Şimdi de tekerlekteki bir elemanın yazısının belirlenmesi için aşağıdaki metodun yapılması gerekmektedir:

```
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?
```

Bu metodun birinci parametresi yine hangi UIPickerView nesnesi için bu işlemin yapılacağını belirtmektedir. İkinci parametre ilgili tekerliğin hangi elemanı için yazı döndürüleceğini belirtmektedir. Üçüncü parametre de hangi tekerlek için bu işlemin yapılacağını belirtmektedir. Örneğin:

3) Artık UIPickerView nesnemizi yaratabiliriz. Nedeni yarattıktan sonra UIPickerView sınıfının delegate ve dataSource property'lerine bu protokollerini destekleyen sınıf nesnelerini atamamız gereklidir. Örneğin:

```
pickerView = UIPickerView(frame: CGRect(x: 10, y: 30, width: 200, height: 50))
pickerView.dataSource = self
pickerView.delegate = self
```

Tek tekerlikli UIPickerView için aşağıdaki örnek verilebilir:

```
import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {

 var fruits: [String]!
 var pickerView: UIPickerView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 fruits = ["Banana", "Apple", "Apricot", "Mellan", "Pineapple", "Orange"]

 pickerView = UIPickerView(frame: CGRect(x: 10, y: 30, width: 200, height: 100))
 pickerView.dataSource = self
 pickerView.delegate = self

 self.view.addSubview(pickerView)
 }

 func numberOfComponents(in pickerView: UIPickerView) -> Int
 {
 return 1
 }

 func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
 {
 return fruits.count
 }

 func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?
 {
 return fruits[row]
 }
}
```

```

 }

 func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int)
-> String?
{
 return fruits[row]
}
}

```

4) UIPickerView'dan seçilen elemanın satır numarası sınıfın selectedRow(inComponent:) metodundan elde edilebilir. Bu metodun parametresi tekerlek numarasını (component'in numarasını) bizden almaktadır. Tabii tek tekerlek varsa bu parametreyi dikkate almamıza gerek olmaz. Örneğin:

```

@objc
func onButtonOk(sender: UIButton)
{
 let row = pickerView.selectedRow(inComponent: 0)
 print(fruits[row])
}

```

Tek tekerlekli basit uygulama bir düğmeyi de içerecek biçimde aşağıdaki gibi yazılabilir:

```

import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {

 var fruits: [String]!
 var pickerView: UIPickerView!
 var buttonOk: UIButton!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 fruits = ["Banana", "Apple", "Apricot", "Mellan", "Pineapple", "Orange"]

 pickerView = UIPickerView(frame: CGRect(x: 10, y: 50, width: 100, height: 100))
 pickerView.dataSource = self
 pickerView.delegate = self
 self.view.addSubview(pickerView)

 buttonOk = UIButton(frame: CGRect(x: 10, y: 200, width: 100, height: 100))
 buttonOk.setTitleColor(.black, for: .normal)
 buttonOk.setTitle("Ok", for: .normal)
 buttonOk.backgroundColor = .yellow
 buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)

 self.view.addSubview(buttonOk)
 }

 @objc
 func onButtonOk(sender: UIButton)
 {
 let row = pickerView.selectedRow(inComponent: 0)
 print(fruits[row])
 }

 func numberOfComponents(in pickerView: UIPickerView) -> Int

```

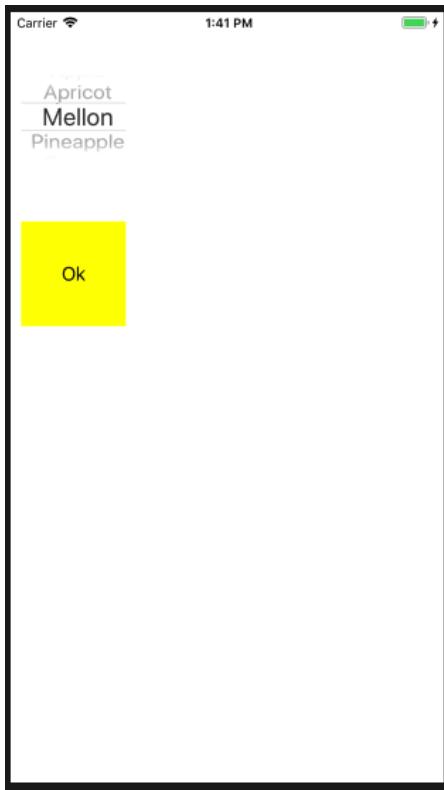
```

 {
 return 1
 }

 func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
 {
 return fruits.count
 }

 func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?
 {
 return fruits[row]
 }
}

```



Şimdi de birden fazla tekerlek içeren bir uygulma yapalım:

```

import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {

 let fruits = ["Banana", "Apple", "Apricot", "Mellan", "Pineapple", "Orange"]
 let colors = ["Red", "Green", "Blue", "White", "Magenta", "Brown", "Gray"]
 let cars = ["Mercedes", "Volkswagen", "Audi", "BMW", "Ford", "Renault", "Toyota", "Chevrolet"]

 var pickerView: UIPickerView!
 var buttonOk: UIButton!

 override func viewDidLoad()
 {
 super.viewDidLoad()

```

```

pickerView = UIPickerView(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
pickerView.dataSource = self
pickerView.delegate = self
self.view.addSubview(pickerView)

buttonOk = UIButton(frame: CGRect(x: 10, y: 200, width: 100, height: 100))
buttonOk.setTitleColor(.black, for: .normal)
buttonOk.setTitle("Ok", for: .normal)
buttonOk.backgroundColor = .yellow
buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)

self.view.addSubview(buttonOk)
}

@objc
func onButtonOk(sender: UIButton)
{
 let row1 = pickerView.selectedRow(inComponent: 0)
 let row2 = pickerView.selectedRow(inComponent: 1)
 let row3 = pickerView.selectedRow(inComponent: 2)

 print(fruits[row1], colors[row2], cars[row3])
}

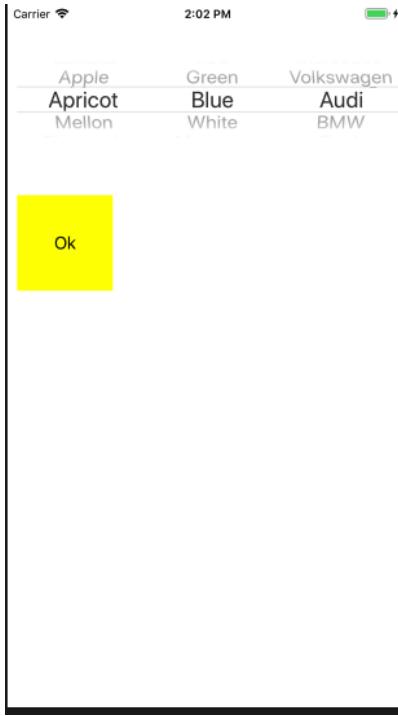
func numberOfComponents(in pickerView: UIPickerView) -> Int
{
 return 3
}

func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
{
 switch component {
 case 0:
 return fruits.count
 case 1:
 return colors.count
 case 2:
 return cars.count
 default:
 return 0
 }
}

func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?
{
 switch component {
 case 0:
 return fruits[row]
 case 1:
 return colors[row]
 case 2:
 return cars[row]
 default:
 return nil
 }
}

```

```
}
```



PickerView kontrolünü IB'de görsel olarak da oluşturabiliriz. Ancak maalesef bu kontrol kod yazmayı gerektirdiği için kod kısmının otomatize edilmesi mümkün olmamaktadır.

### UIDatePicker Kullanımı

Kullanıcıdan bir tarih seçmesini istemek için UIDatePicker nesnesi tercih edilmektedir. Bu nesne aslında UIPickerView kullanılarak gerçekleştirılmıştır. Dolayısıyla kontrolde tarih semek için üç tane tekerlek bulunur. Kontrolün kullanımı şöyledir:

1) UIDatePicker türünden bir nesne yaratılır ve konumlandırma yapılır. Örneğin:

```
datePicker = UIDatePicker(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
self.view.addSubview(datePicker)
```

Default durumda kontrol hem tarihi hem de zamanı göstermektedir. Kontrolün bu davranışını UIDatePicker sınıfının datePickerMode property'si ile değiştirilebilir. Bu property Mode isimli enum türündendir. Bu enum türünün elemanları şunlardır:

```
time
date
dateAndTime (default)
countDownTimer
```

Örneğin:

```
datePicker = UIDatePicker(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
datePicker.datePickerMode = .date
self.view.addSubview(datePicker)
```

Bu örnekte UIDatePicker nesnesi tarih alacak biçimde ayarlanmıştır.

2) Kontrolün içerisindeki tarih ve zaman sınıfının date isimli property'si ile alınabilir. ve set edilebilir. Ancak Cocoa'daki Date yapısı maalesef kendi başına işleme sokulacak biçimde organize edilmemiştir. Genellikle programcılar bu Date nesnesini oluşturabilmek ve içerisindeki bilgiyi labilmek için yardımcı sınıflar kullanırlar.

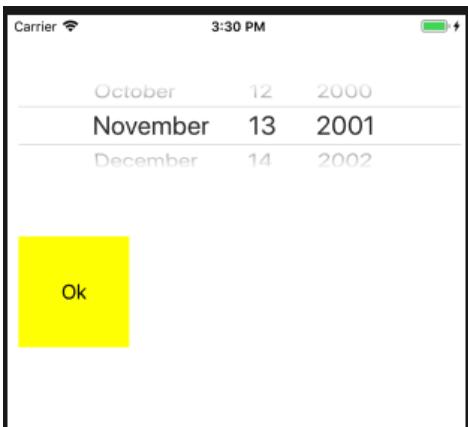
Bir tarih bilgisini oluşturabilmek için Date sınıfının init metotları çok yeterli değildir. Genelikle tarih bilgileri DateFormatter denilen yardımcı bir sınıf yoluyla oluşturulmaktadır. Programcı bir DateFormatter nesnesi yaratır. Sonra dateFormat property'si ile yazışal olarak bir biçim belirler. Biçim belirleme işlemi bir string'le yapılmaktadır. Bu string'te y, M, d, h, m, s gibi harfler tarih ve zamanın bileşenleri için kullanılır. Daha sonra da DateFormatter sınıfının date isimli metoduyla verilen tarih Date sınıfına dönüştürülür, string metoduyla da söz konusu tarih yazıya dönüştürilmektedir. date metodu bizden dönüştürülecek tarihi yazı olarak, string metodu da Date nesnesi olarak almaktadır. Örneğin:

```
datePicker = UIDatePicker(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
datePicker.datePickerMode = .date
self.view.addSubview(datePicker)

let dateFormatter = DateFormatter()
dateFormatter.dateFormat = "dd/MM/yyyy"
let date = dateFormatter.date(from: "13/12/2001")

datePicker.date = date!
```

Burada biz "13/12/2001" tarihini DateFormatter sınıfından faydalananarak Date sınıfına dönüştürdük. Bu tarihi UIDatePicker sınıfının date property'sine atarsak kontrol bu tarihle açılacaktır. Default durumda kontrol geçerli tarih ve zamanla açılmaktadır.



Şimdi de düğmeye basıldığında kontrol içerisindeki tarihi alıp yazdırma isteyelim. Bunun için UIDatePicker sınıfının date property'si kullanılır. Date nesnesi olarak verilen tarihi yazıya dönüştürmek için DateFormatter sınıfının string metodu kullanılmaktadır. Örneğin:

```
@objc
func onButtonOk(sender: UIButton)
{
 let dateFormatter = DateFormatter()
 dateFormatter.dateFormat = "dd/MM/yyyy"
 let str = dateFormatter.string(from: datePicker.date)
 print(str)
}
```

Date nesnesinin içerisindeki bileşenleri tek tek nasıl alabiliriz? . Maalesef Date yapısı bize bileşenleri doğrudan vermemektedir. Tabii bu Date nesnesini yazıya dönüştürüp string işlemleriyle bunları almak mümkündür. Ancak bu yöntem fazlaca işlem yoğunluğu gerektirmektedir. İşte bunun için Calendar yapısı kullanılmaktadır.

Calendar türünden nesne tipik olarak sınıfın static current isimli property'si ile yaratılır. Daha sonra bu nesne ile sınıfın örnek component isimli metod çağrılar. component metodunun birinci parametresi hangi bilşenin elde edileceğini belirten bir enum türündendir. İkinci parametre ise bileşenei elde edilecek olan Date nesnesini belirtir. Örneğin:

```
@objc
func onButtonOk(sender: UIButton)
{
 let calendar = Calendar.current
 let day = calendar.component(.day, from: datePicker.date)
 let month = calendar.component(.month, from: datePicker.date)
 let year = calendar.component(.year, from: datePicker.date)

 print("\(day)/\(month)/\(year)")
}
```

Ayrıca bir Date nesnesi DateComponents ve Calendar yapısı yoluyla da oluşturulabilmektedir. Programcı önce bir DateComponents nesnesi oluşturur. Tarih ve zaman bileşenlerini bu nesneye yerleştirir. Sonra Calendar sınıfının date(from:) metodu ile DateComponents nesnesini vererek Date nesnesi alabilir. Örneğin:

```
datePicker = UIDatePicker(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
datePicker.datePickerMode = .date
self.view.addSubview(datePicker)

var dc = DateComponents()

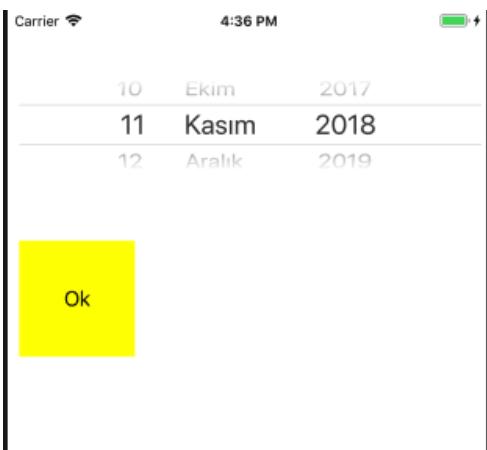
dc.day = 13
dc.month = 12
dc.year = 2001

let calendar = Calendar.current
let date = calendar.date(from: dc)

datePicker.date = date!
```

3) UIDatePicker sınıfının locale isimli property'yi Locale isimli yapı türündendir. Biz bu property'yi örneğin Locale("str") biçiminde set edersek bu durumda kontrol bize tarihi Türkçe gösterecektir. Örneğin:

```
datePicker = UIDatePicker(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
datePicker.datePickerMode = .date
datePicker.locale = Locale(identifier: "tr")
self.view.addSubview(datePicker)
```



Aşağıda kontrolün kullanımına ilişkin bir örnek görüyorsunuz:

```
import UIKit

class ViewController: UIViewController {
 var datePicker: UIDatePicker!

 var buttonOk: UIButton!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 datePicker = UIDatePicker(frame: CGRect(x: 10, y: 50, width: 400, height: 100))
 datePicker.datePickerMode = .date
 datePicker.locale = Locale(identifier: "tr")
 self.view.addSubview(datePicker)

 buttonOk = UIButton(frame: CGRect(x: 10, y: 200, width: 100, height: 100))
 buttonOk.setTitleColor(.black, for: .normal)
 buttonOk.setTitle("Ok", for: .normal)
 buttonOk.backgroundColor = .yellow
 buttonOk.addTarget(self, action: #selector(onButtonOk), for: .touchUpInside)

 self.view.addSubview(buttonOk)
 }

 @objc
 func onButtonOk(sender: UIButton)
 {
 let calendar = Calendar.current
 let day = calendar.component(.day, from: datePicker.date)
 let month = calendar.component(.month, from: datePicker.date)
 let year = calendar.component(.year, from: datePicker.date)

 print("\(day)/\(month)/\(year)")
 }
}
```

UIDatepicker sınıfının ayrıntılı kullanımı için dokümanlar incelenebilir.

### UIActivityIndicatorView Kullanımı

UIActivityIndicatorView programın o anda belli bir meşguliyetinin olduğunu göstermek için kullanılan basit bir view

penceresidir. Diğer framework'lerde de buna benzer bir kontrol bulunmaktadır. UIActivityIndicatorView sınıfı kullanılmaktadır:

1) UIActivityIndicatorView sınıfı cinsinden bir nesne yaratılır ve konumlandırma yapılır. Örneğin:

```
import UIKit

class ViewController: UIViewController {

 var ai: UIActivityIndicatorView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 ai = UIActivityIndicatorView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))

 ai.backgroundColor = .yellow

 self.view.addSubview(ai)
 }
}
```

2) View'nun style property'si genel görünümü belirlemek için kullanılmaktadır. Bu property aşağıdaki değerlerden birini almaktadır:

```
whiteLarge
white
gray
```

Yine sınıfın color isimli property elemanı genel olarak nesnenin rengini belirlemekte kullanılmaktadır.

3) Aktivasyonu başlatmak için sınıfın startAnimating metodu çağrılır. Aktivasyonu sonlandırmak için ise stopAnimating metodu çağrılır. Aktivasyon yokken pencerenin görüntülenip görüntülenmeyeceği sınıfın Bool türden hidesWhenStopped property'si ile belirlenmektedir. Bu property default olarak True biçimdedir.

```
import UIKit

class ViewController: UIViewController {

 var ai: UIActivityIndicatorView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 ai = UIActivityIndicatorView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))

 ai.style = .whiteLarge
 ai.hidesWhenStopped = false
 ai.color = .blue
 self.view.addSubview(ai)
 }
}
```

## UIScrollView Kullanımı

Telefon gibi mobil aygıtlar küçük bir ekranı sahiptir. Ancak bize çok sayıda kontrolün ekranı yerleştirilmesi gerekebilir. Örneğin 100 tane düğmeyi küçük bir ekranı yerleştirememiz. Yerleştirsek bile bunlar çok küçük olacağından kullanım zorluğu olacaktır. İşte UIScrollView kontrolü başka kontrolleri içeren ve onların scroll edilmesini sağlayan bir kontroldür. Bu kontrol şöyle kullanılmaktadır:

- 1) UIScrollView sınıfı türünden bir nesne oluşturulur ve konumlandırılır. Genellikle bu nesne ana ekranın boyutu kadar büyük olacak biçimde oluşturulmaktadır.

```
override func viewDidLoad()
{
 super.viewDidLoad()

 scrollView = UIScrollView(frame: self.view.bounds)
 scrollView.backgroundColor = .yellow
 self.view.addSubview(scrollView)
}
```

- 2) UIScrollView'ine kontroller yerleştirilir. Yani yerleştirilecek kontroller UIScrollView nesnesinin alt penceresi (sub view) gibi olmalıdır. Örneğin:

```
scrollView = UIScrollView(frame: self.view.bounds)
scrollView.backgroundColor = .yellow
self.view.addSubview(scrollView)

for i in 0..<100 {
 let label = UILabel()
 label.text = String(i)
 label.frame = CGRect(x: 10, y: i * 20, width: 100, height: 15)
 scrollView.addSubview(label)
}
```

Burada 100 tane UILabel nesnesi UIScrollView'ine yerleştirilmiştir.

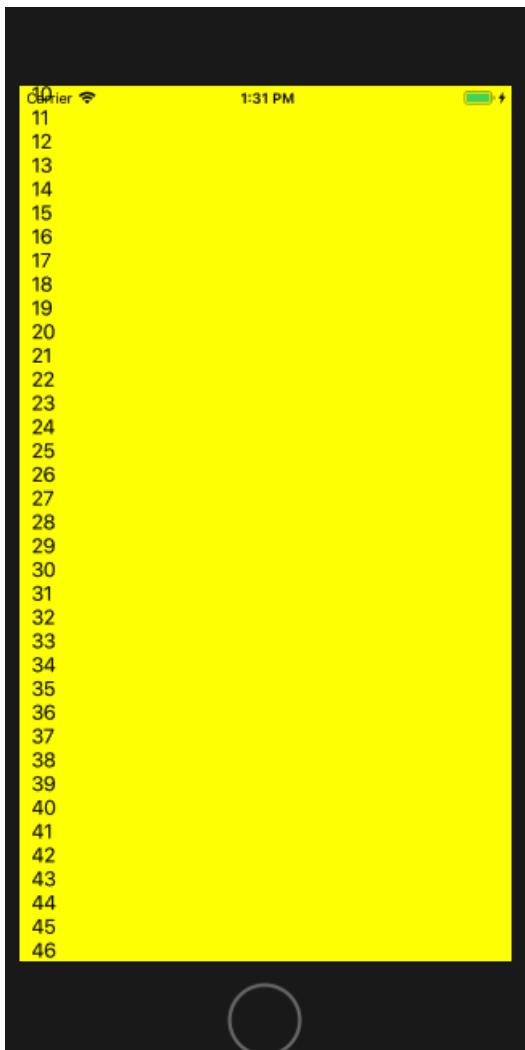
- 3) Scroll alanı UIScrollView sınıfının contentSize isimli property'si ile set edilmektedir. Bu property CGSize türündendir ve genişlik-yükseklik değerlerini almaktadır. contentSize adet olarak UIScrollView nesnesinin sanal boyutunu belirtmektedir. Bu boyutun tam da scroll edilecek miktar kadar olması anlamlıdır. Örneğin:

```
override func viewDidLoad()
{
 super.viewDidLoad()

 scrollView = UIScrollView(frame: self.view.bounds)
 scrollView.backgroundColor = .yellow
 self.view.addSubview(scrollView)

 for i in 0..<100 {
 let label = UILabel()
 label.text = String(i)
 label.frame = CGRect(x: 10, y: i * 20, width: 100, height: 15)
 scrollView.addSubview(label)
 }
 scrollView.contentSize = CGSize(width: scrollView.bounds.width, height: 100 * 20)
}
```

Örneğimizde yatay içerik alanı (contentSize) tam UIScrollView kontrolünün genişliği kadar verilmiştir. Aslında daha geniş de olabilirdi. Ancak içerik olmadıktan sonra geniş bir içerik alanı (contentSize) oluşturmak iyi bir teknik değildir.



Yukarıdaki örneğin kodu şöyledir:

```
import UIKit

class ViewController: UIViewController {
 var scrollView: UIScrollView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 scrollView = UIScrollView(frame: self.view.bounds)
 scrollView.backgroundColor = .yellow
 self.view.addSubview(scrollView)

 for i in 0..<100 {
 let label = UILabel()
 label.text = String(i)
 label.frame = CGRect(x: 10, y: i * 20, width: 100, height: 15)
 scrollView.addSubview(label)
 }
 }
}
```

```

 scrollView.contentSize = CGSize(width: scrollView.bounds.width * 2, height: 100 * 20)
 }
}

```

**Sınıf Çalışması:** Her satırda 10 tane olacak biçimde 100 x 100 lik 200 düğmeyi bir UIScrollView içerisinde yaratınız ve düğmelerin üzerinde 1'den başlayarak sayılar yazılı olsun. Düğmeler arasında yatayda ve düşeyde 20 pixel boşluk bırakılmalıdır. Aynı zamanda yukarıdan, aşağıdan, soldan ve sağdan 20 pixel boşluk olmalıdır.

Çözüm:

```

import UIKit

class ViewController: UIViewController {
 var scrollView: UIScrollView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 scrollView = UIScrollView(frame: self.view.bounds)
 self.view.addSubview(scrollView)
 var left, top, count: Int
 top = 20
 count = 1
 for _ in 0..<20 {
 left = 20
 for _ in 0..<10 {
 let button = UIButton(type: .system)
 button.frame = CGRect(x: left, y: top, width: 100, height: 100)
 button.backgroundColor = .green
 button.addTarget(self, action: #selector(onButtonClicked(sender:)), for:
.touchUpInside)
 left += 120
 button.setTitle(String(count), for: .normal)
 count += 1
 scrollView.addSubview(button)
 }
 top += 120
 }
 scrollView.contentSize = CGSize(width: 1220, height: 2420)
 }

 @objc
 func onButtonClicked(sender: UIButton)
 {
 print(sender.titleLabel!.text!)
 }
}

```

Aslında UIScrollView içeresine tek bir kontrol yerleştirip onu da scroll edebiliriz. Örneğin bir UIScrollView içeresine bir UITableView ya da bir UIImageView yerleştirebiliriz. Aşağıdaki örnekte büyük bir resim bir UIImageView içeresine, UIImageView nesnesi de UIScrollView içeresine yerleştirilmiştir. Böylece resmin her yerri scroll edilerek görüntülenebilmektedir:

```

class ViewController: UIViewController {
 var scrollView: UIScrollView!

```

```

override func viewDidLoad()
{
 super.viewDidLoad()

 scrollView = UIScrollView(frame: self.view.bounds)
 self.view.addSubview(scrollView)
 let imageView = UIImageView(image: UIImage(contentsOfFile:
"/Users/KaanAslan/Dropbox/Shared/Kurslar/Swift-IOS-Nisan-
2018/Src/BasicControls/BasicControls/AbbeyRoad.jpg"))
 scrollView.addSubview(imageView)
 scrollView.contentSize = imageView.image!.size
}

@objc
func onButtonClicked(sender: UIButton)
{
 print(sender.titleLabel!.text!)
}
}

```



## UITableView Kullanımı

UITableView IOS uygulamalarında en fazla kullanılan kontrollerden biridir. Bu kontrolü diğer framework'lerdeki "listview" ya da gelişmiş bir "listbox" kontrolüne benzetebiliriz. Gerçekten de IOS'ta ne zaman bir grup öğe liste biçiminde görüntülenecek olsa UITableView kontrolü kullanılmaktadır. Apple IOS'un temel uygulamalarında da UITableView kontrolünü çokça kullanmıştır. UITableView kontrolü oldukça genel amaçlı tasarlandığı için ayrıntılara sahiptir. Bu nedenle kontrolün kullanımının öğrenilmesi biraz zaman almaktadır.

Öncelikle tıpkı UIPickerView kontrolünde olduğu gibi UITableView kontrolünde de kontrolün göstereceği bilgiler baştan bir dizi olarak değil gerekiğinde o anda yaratılıp verilmektedir. Bunun birkaç nedenei vardır:

- 1) Uzun listelerde tüm listenin baştan kontrole verilmesi gereksiz bir bellek alanı israfına yol açabilmektedir. Oysa kontrolün göstereceği elemanı o anda ona vermek az miktarda bellek kullanılmasına yol açmaktadır.
- 2) Nesnenin gösterici öğelerin o anda oluşturulması daha esnek bir kullanımına yol açmaktadır.

Biz UITableView kontrolünde gerekiğinde gösterilecek elemanı yaratıp ona veririz. Pekiyi görsel bakımından tablonun dışına çıkan öğelere ne olmaktadır? Bu öğeler kontrol tarafından saklanıp geri basılmazlar. Kontrol her öğeyi her zaman yine bizden almaktadır. İşte bu durumda yukarı aşağı kaydırımlar sırasında aynı öğelerin defalarca yaratılıp yok edilmesinin önüne geçebilmek için kontrol görüntü dışına geçmiş olan öğeleri kendisi yok etmemekte ve onu bir kuyruk sistemine yerleştirmektedir. Böylece biz belli bir öğe görüntüleneceği zaman yeni bir öğe nesnesi yaratmak yerine kontrolden onun sakladığı bir öğeyi alabiliriz. O öğede değişiklikler yapıp onu yeniden kontrole verebiliriz. Böylece aslında çok az sayıda yaratılmış olan öğelerle binlerce öğe görüntülenebilmektedir. 8Çünkü nasıl olsa bunlar aynı anda görüntülenmemektedir.)

Bir UITableView nesnesi aslında üç biçimde oluşturulabilmektedir:

- 1) Sıfırdan mevcut bir Controller sınıfına protokol desteği vererek
- 2) Sıfırdan bir UITableViewController sınıfı oluşturarak
- 3) IB yoluya bazı işlemleri kısmen görsel biçimde yaparak

Burada bu yöntemleri tek tek inceleyeceğiz. Ancak bu kontrolün oldukça detaylı sahip olduğunu tekrar anımsatmak istiyoruz.

#### Sıfırdan Mevcut Controller Nesnesine Protokol Desteği Vererek UITableView Nesnelerinin Yaratılması

Bir UITableView nesnesi kabaca (ayrintılar göz ardı edilerek) şu aşamalardan geçilerek sıfırdan oluşturulabilir.

1) Ana Controller sınıfımızın UITableViewDataSource ve UITableViewDelegate protokollerini desteklemesini sağlamalıyız. Bu protokoller nedeniyle controller sınıfımıza üç adet metot yazmak zorundayız. Bu metotlar numberOfRowsInSection(in:), tableView(\_:numberOfRowsInSection:) ve tableView(\_:cellForRowAt:) metotlarıdır. Örneğin:

```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
 var tableView: UITableView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 tableView = UITableView(frame: self.view.bounds)
 //...
 self.view.addSubview(tableView)
 }
}
```

```

func numberOfSections(in tableView: UITableView) -> Int
{
 //...
}

func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
 //...
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
 //...
}
}

```

2) UITableView nesnesi viewDidLoad metodunda yaratılır ve konumlandırılır. Daha sonra bu nesnenin delegate ve dataSource property'lerine ilgili controller nesnesi (örneğimizde self) atanır. Örneğib:

```

var tableView: UITableView!

override func viewDidLoad()
{
 super.viewDidLoad()

 tableView = UITableView(frame: self.view.bounds)
 tableView.dataSource = self
 tableView.delegate = self
 self.view.addSubview(tableView)
}

```

3) numberOfSection(in:) metodu kontrol tarafından işin başında bir kez çağrılmaktadır. UITableView nesnesi bölgümlerden (sections) oluşturulabilir. İşte bu metotta biz kontrolümüzde kaç tane bölüm olacağını bölüm sayısı ile geri dönerek belirtiriz. Örneğin:

```

func numberOfSections(in tableView: UITableView) -> Int
{
 return 1
}

```

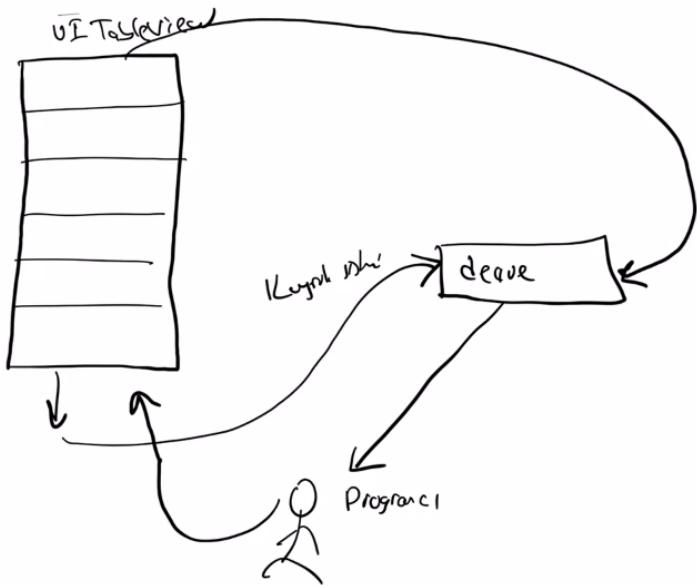
4) Her bölümde farklı sayıda satır (yani eleman) bulunabilmektedir. İşte kontrol her bölüm için bir kez tableView(\_:numberOfRowsInSection) metodunu çağırarak o bölümde kaç tane satır olacağını bize sorar. Bizim kontrolümüzde tek bir bölüm olduğuna göre bölüm sayısına bakmadan doğrudan satır sayısını ile geri dönebiliriz. Örneğin:

```

func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
 return 100
}

```

5) Şüphesiz en önemli protokol metodu tableView(\_:cellForRowAt:) isimli metottur. Bu metot UITableViewCell türünden bir nesneye geri dönürtülmelidir. Aslında UITableView kontrolünün satırları UITableViewCell nesnelerinden oluşturulmuştur. Biz de hangi satır görüntüleneceğse o satıra ilişkin UITableViewCell nesnesini yaratıp bu nesneye geri dönmemeliyiz. Tabii yukarıda da belirtildiği gibi görüntü dışına çıkan satırların UITableViewCell nesnelerini kontrol kendi içerisinde yeniden kullanılsın diye programcıya vermek üzere saklamaktadır. O halde bizim önce verimlilik açısından kontrolden zaten yaratılmış bir nesne istememiz uygun olur. Tabii işin başında henüz nesne yaratılmadığı için bize de böyle bir nesne verilmeyecektir. Bu durumda bizim gerçek UITableViewCell nesnelerini yaratmamız gereklidir.



UITableView sınıfının dequeueReusableCellWithIdentifier(withIdentifier:) metodu bize kontrolün sakladığı bir UITableViewCell nesnesini vermektedir. Eğer verilecek böyle bir nesne yoksa metot nil değerine geri döner. İşte bu durumda bizim kendimizin UITableViewCell nesnesini yaratmamız gereklidir. Metot şöyle yazılabilir:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
 var cell: UITableViewCell!

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = UITableViewCell(style: .default, reuseIdentifier: "mycell")
 cell.textLabel!.textColor = .black
 }

 cell.textLabel!.text = "TextLabel: " + String(indexPath.row)

 return cell
}
```

Tablo üzerinde değişiklik yapmak birkaç bakımdan mümkündür. Tablodak her satır UITableViewCell türündendir. Biz istersek her satırda diğer satırlardan farklı olarak değişiklikler yapabiliriz. Sayı üzerinde yapabileceğimiz önemli değişiklikler şunlardır:

- UITableViewCell sınıfının.textLabel isimli property elemanı UILabel? türündendir. Dolayısıyla biz UILabel üzerinde yapabildiğimiz her şeyi bu leman üzerinde yapabiliriz. Örneğin yazının fontunu ve rengini değiştirebiliriz:
- defuault hücrelendirmede sağ tarafta bir donatı bölümü vardır. Bu donatinın cinsi accessoryType property'si belirlenir. accesoryType property'si bir enum türündendir. Bu enum türünün elemanları şunlardır:

```
none
disclosureIndicator
detailDisclosureButton
checkmark
detailButton
```

İstenirse sağ taraf donatısı olarak istediğimiz view da kullanılabilir. Bunun için UITableViewCell sınıfının accessoryView property'si kullanılmaktadır. Örneğin biz sağ taraftaki donatılar için bir düğme yerleştirebiliriz:

```
import UIKit

class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
 var tableView: UITableView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 tableView = UITableView(frame: self.view.bounds)
 tableView.dataSource = self
 tableView.delegate = self
 self.view.addSubview(tableView)
 }

 func numberOfSections(in tableView: UITableView) -> Int
 {
 return 1
 }

 func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
 {
 return 100
 }

 func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
 {
 var cell: UITableViewCell!

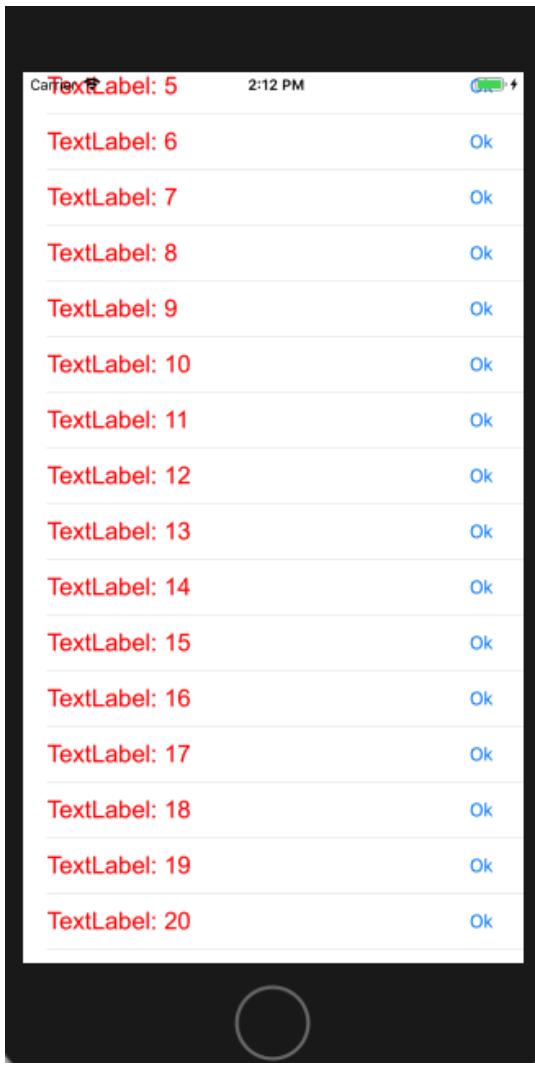
 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = UITableViewCell(style: .default, reuseIdentifier: "mycell")
 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)

 let button = UIButton(type: .system)
 button.setTitle("Ok", for: .normal)
 button.sizeToFit()

 cell.accessoryView = button
 }

 cell.textLabel!.text = "TextLabel: " + String(indexPath.row)

 return cell
 }
}
```



Genellikle uygulamalarda sağ taraftaki donatı olarak UISwitch kontrolleri yoğun biçimde kullanılmaktadır. Örneğin:

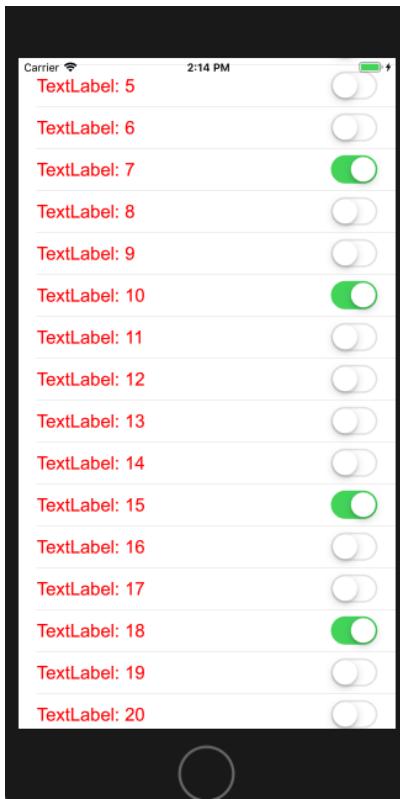
```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
 var cell: UITableViewCell!

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = UITableViewCell(style: .default, reuseIdentifier: "mycell")
 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)

 let switcher = UISwitch()
 cell.accessoryView = switcher
 }

 cell.textLabel!.text = "TextLabel: " + String(indexPath.row)

 return cell
}
```



- Satırlar üzerindeki UILabel nesnelerinin textAlignment property'lerini değiştirerek yazıların satır üzerinde hizalanması sağlanabilmektedir. Örneğin:

```
cell.textLabel!.textAlignment = .center
```

- Satırlara girinti düzeyleri verilebilmektedir. Bu girinti düzeyi girinti miktarıyla çarpılarak toplam girinti hesaplanmaktadır. Görüntü düzeyi için UITableViewCell sınıfının indentationLevel property'si ile girinti miktarı ise sınıfın indentationWidth property'si ile ayarlanmaktadır.

- Bir satır seçildiğinde satırın seçim biçimi selectionStyle isimli property ile ayarlanabilmektedir. Bu enum türünün elemanları şunlardır:

```
blue
default
gray
none
```

- Hücrelerin zemin renkleri backgroundColor property'si ile değiştirilebilmektedir İstenirse zemin için başka bir view da kullanılabilir. Bu durumda view sınıfın backgroundView property'si ile set edilir. Satır seçildiğinde selectedBackgroundView property'si ile set edilen view gösterilmektedir.

- Bir satırdaki yazı birden fazla satırdan oluşabilir. Bu durumda default olarak yalnızca tek satır görüntülenmektedir. İşte birden fazla satır görüntüleyebilmek için UITableViewCell nesnesi içerisindeki UILabel nesnesinin numberOfLines property'si set edilir.

UITableView sınıfının property'leri ise belli bir satıra yönelik değil tüm kontrole yönelik işlemlerde kullanılmaktadır. Burada önemli olanların üzerinde duracağız:

- UITableView sınıfının rowHeight property'si tüm satırların yüksekliğini belli bir değere ayarlamak için kullanılır.
- UITableView sınıfının separatorStyle ve separatorColor property'leri satırları ayıran çizgi konusunda etkili olmaktadır. separatorStyle sunlardan biri olabilir:

```
none
singleLine
```

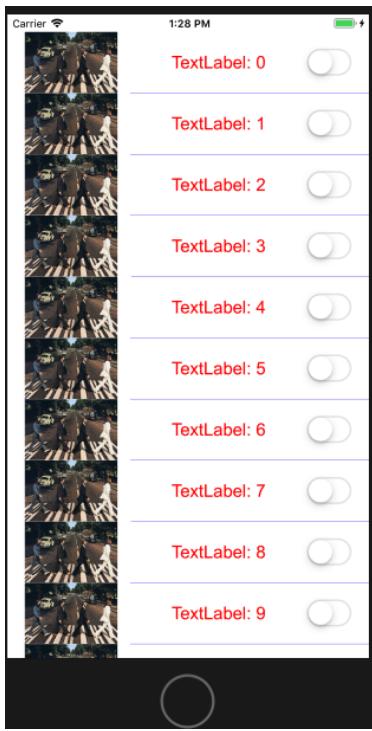
- Default durumda table view'da yalnızca tek bir satır seçilebilmektedir. Birden fazla satırın seçilebilmesi için UITableView sınıfının allowMultipleSelection isimli property'si true yapılmalıdır. İstersek kontrolün seçim yapmasını da allowSelection property'sine false değeri yerleştirek engelleyebiliriz.

- Seçilen satırlar UITableView sınıfının indexPathForSelectedRow ve indexPathForSelectedRows isimli property'lerle alınabilmektedir. indexPathForSelectedRow property'si IndexPath? türünden, indexPathForSelectedRows property'si ise [IndexPath]? türündendir.

Hücrelerin default stilinde (Yani UITableViewCell nesnesi yaratılırken style etiketi ile girdiğimiz değer) aslında satır (hücre) üzerinde yalnızca bir UILabel nesnesi ve donatı değil bir de UIImageView nesnesi de vardır. Biz bu UIImageView nesnesine UITableViewCell sınıfının imageView property'si ile erişilebilir. Bu property let biçimindedir. Fakat biz bu property'nin image elemanını değiştirebiliriz. Örneğin:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
 var cell: UITableViewCell!
 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = UITableViewCell(style: .default, reuseIdentifier: "mycell")
 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)
 cell.imageView?.image = UIImage(contentsOfFile:
 "/Users/KaanAslan/Dropbox/Shared/Kurslar/Swift-IOS-Nisan-2018/Src/BasicControls/BasicControls/AbbeyRoad.jpg")
 let switcher = UISwitch()
 switcher.sizeToFit()
 cell.accessoryView = switcher
 cell.textLabel!.numberOfLines = 5
 cell.textLabel!.textAlignment = .center
 }
 cell.textLabel!.text = "TextLabel: " + String(indexPath.row)
 return cell
}
```

Yukarıdaki örnekte her stardaki UITableViewCell nesnesinde bir tane UIImage, bir tane UILabel bir tane de donatı kısmında "accessoryView" olarak UISwitcher nesnesi vardır. Bu elemanlar satır default style ile yaratıldığında otomatik olarak nesnede bulunmaktadır ve bu elemanlara yukarıda da ele alındığı gibi UITableViewCell sınıfının.textLabel, imageView ve accessoryView property'leri ile erişilir.



Bu noktada programcıların en fazla tereddütte kaldığı noktalardan birisi menzil içerisinde giren satırı dequeReusableCell(withIdentifier:) metodu ile aldığımızda buradaki yazıların ve switcher'ların konumlarının bozulmasıdır. Bunun nedeni bu metodun kuyruktaki rastgele bir nesneyi vermesindedir. Sorunu çözmenin basit bir yolu nesneyi bu metottan aldıktan sonra onun değişimini yeniden oluşturmak olabilir.

### UITableView Verilerinin Organize Edilmesi

Yukarıdaki örneklerde biz UITableViewCell verilerini o anda satır numarasından hareketle oluşturduk. Çok sayıda tablo elemanı olduğu durumda gerçekten bu verilerin otomatik oluşturulması gerekebilir. Ancak az sayıda tablo elemanı söz konusu olduğunda bu verilen daha uygun bir biçimde dizisel olarak oluşturulması tavsiye edilir. Örneğin biz UITableView içerisinde 15 tane şehir ismini yerlestirecek olalım. Bu isimleri önce bir diziye yerleştirip daha sonra UITableViewCell nesnesinin UILabel kontrolünün yazısı olarak atayabiliriz.

UITableView kullanımında donanımları olarak UISwitch kullanıldığında bu switch'lerin on/off durumlarının izlenmesi gereklidir. Çünkü menzil dışına çıkan satırlar bize geri verildiğinde o satır rastgele bir UITableViewCell nesnesi olabilmektedir. Buradaki UISwitch nesnesi bu nesnesinin içerisinde olduğu için bunların konumları bozulabilemektedir. Peki konumu nasıl saklayıp yeniden kullanabiliriz? İşte bunun yollarından biri her satırındaki UITableViewCell nesnesinin içindeki UISwitch konumlarını saklamaktır. Buna ilişkin bir örnek aşağıda verilmiştir:

```
import UIKit

class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
 var tableView: UITableView!
 var cities: [(String, Bool)] = [("Adana", false), ("Sivas", false), ("Eskişehir", false),
 ("İstanbul", false), ("Ankara", false), ("Edirne", false), ("Sakarya", false), ("Trabzon", false),
 ("Samsun", false), ("Balıkesir", false), ("Kars", false), ("Van", false), ("Malatya", false),
 ("Hatay", false), ("İzmir", false)]

 override func viewDidLoad()
 {
 super.viewDidLoad()
```

```

 tableView = UITableView(frame: self.view.bounds)
 tableView.dataSource = self
 tableView.delegate = self
 tableView.rowHeight = 70
 tableView.separatorStyle = .singleLine
 tableView.separatorColor = .blue
 //tableView.allowsMultipleSelection = true
 self.view.addSubview(tableView)
 }

 func numberOfSections(in tableView: UITableView) -> Int
 {
 return 1
 }

 func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
 {
 return 15
 }

 func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
 {
 var cell: UITableViewCell!

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = UITableViewCell(style: .default, reuseIdentifier: "mycell")
 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)

 let switcher = UISwitch()
 switcher.tag = indexPath.row
 switcher.sizeToFit()
 cell.accessoryView = switcher
 cell.textLabel!.numberOfLines = 5
 cell.textLabel!.textAlignment = .center
 switcher.addTarget(self, action: #selector(onSwitcher(sender:)), for: .valueChanged)
 }

 cell.textLabel!.text = cities[indexPath.row].0
 cell.accessoryView!.tag = indexPath.row
 (cell.accessoryView as! UISwitch).isOn = cities[indexPath.row].1

 return cell
 }

 @objc
 func onSwitcher(sender: UISwitch)
 {
 cities[sender.tag].1 = sender.isOn
 }
}

```

Bu örnekte UITableView nesnesinin üzerinde çıkarılacak yazılar ve UISwitch kontrolünün durumu bir demet dizisinde saklanmıştır. Sonra UISwitch üzerinde on/off işlemi yapıldığında hangi switch üzerinde bu işlemin yapıldığı kaydedilmiştir. Kuyrukta boş UlViewCell nesnesi alındığında da bu bilgiler satıra yeniden yerleştirilmiştir. Kodda UIView sınıfının tag property'sinin de kullanıldığına dikkat ediniz. Bu property UInt türündendir ve kullanıcı tarafından kontrolleri

birbirinden ayırmak için istenildiği gibi kullanılabilmektedir.

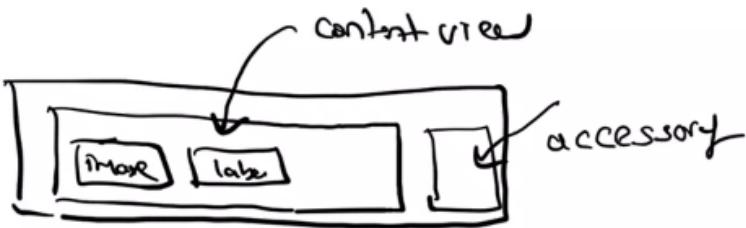
### UITableViewCell Nesnelerinin Ayrıntıları

Bir UITableViewCell nesnesi yaratılırken style adı altında default bir yapılandırma biçimini belirtilmektedir. UITableViewCell sınıfının init metodunun aşağıdaki olduğunu anımsayınız:

```
init(style: UITableViewCell.CellStyle, reuseIdentifier: String?)
```

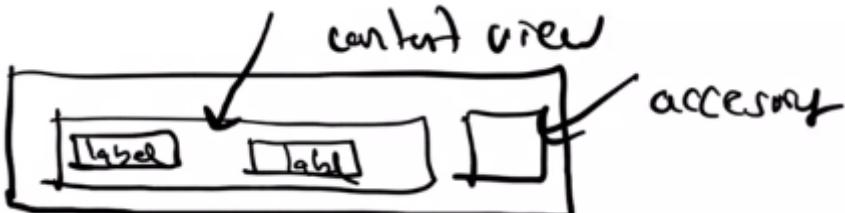
Biz şimdije kadar bu style özelliğini hep .default biçiminde verdik. İşte bu style özelliği yukarıda da belirttiğimiz gibi hücrenin default tasarım biçimini belirtmektedir. Aslında UITableViewCell nesneleri "custom" biçimde de yaratılabilmektedir. Custom yaratımda biz hücrenin içerisinde istediğimiz kontrolleri yerleştirebilmekteyiz. Öncelikle biz zaten var olan style türlerinin ne anlamına geldiğini açıklayalım.

**Default Style:** Bu konfigürasyonda UITableViewCell nesnesinin içerisinde bir content view penceresi vardır. Bu content view penceresinin içerisinde de bir UIImageView bir de UILabel nesnesi bulunur. Bu content view'nun dışında ayrıca bir de donatı (accessory) alanı bulunmaktadır:



Eğer biz nesneyi default biçimde konfigüre etmişsek içerisindeki UIImageView nesnesine imageView property'si ile erişebiliriz. Başlangıçta kontrolün içerisinde bir resim iliştirilmemiştir. Ancak programcı bu resmi iliştirebilir. Nesnenin UILabel elemanına da textLabel property'si ile erişilebilmektedir. Donatı (accessory) kısmında aslında küçük bir pencere vardır. Biz sınıfın accessoryType property'si ile oaradki view nesnesinin ne olması gerektiğini birkaç seçenek içerisinde belirtebiliriz. Eğer istersek bu donatı alanına sınıfın accessoryView property'si yoluyla istediğimiz bir kontrolü yerleştirebiliriz.

**Value1 Style:** Bu konfigürasyonda contentview içerisinde sol tarafta ve sağ tarafta iki ayrı UILabel nesnesi vardır.



Satır böyle yaratılmışsa soldaki label'a sınıfın textLabel property'si ile sağdaki label'a ise detailedTextLabel property'si ile erişilir.

**Value2 Style:** Bu konfigürasyonda da yine iki label vardır. Ancak bu label'lar Value1'in tersi olacak biçimde hizalanmıştır.

**Subtitle Style:** Burada sol tarafta üst üste iki tane label bulunur.

### UITableViewCell Sınıfından Türetme Yapmak

Bazen UITableViewCell sınıfının içehrısına başka elemanları eklemeyi isteyebiliriz. İşte durumda türetme yapmak gereklidir. Türetme gereksiniminin diğer bir nedeni de UITableView kontrolünün bazı durumlarda UITableViewCell sınıfının çokbüçümeli metodlarını çağırması yüzündendir. Biz bazı işlevleri değiştirebilmek için UITableViewCell sınıfından türetme yapıp türemişsinfta bazı sanal metodları override edebiliriz. Aşağıdaki örnekte UITableViewCell sınıfından MyTableViewCell sınıfı türetilmiştir ve UITableView kontrolünün satırları artık MyTableViewCell nesneleriyle oluşturulmuştur:

```
import UIKit

class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
 var tableView: UITableView!
 var cities: [(String, Bool)] = [("Adana", false), ("Sivas", false), ("Eskişehir", false),
 ("İstanbul", false), ("Ankara", false), ("Edirne", false), ("Sakarya", false), ("Trabzon", false),
 ("Samsun", false), ("Balıkesir", false), ("Kars", false), ("Van", false), ("Malatya", false),
 ("Hatay", false), ("İzmir", false)]

 override func viewDidLoad()
 {
 super.viewDidLoad()

 tableView = UITableView(frame: self.view.bounds)
 tableView.dataSource = self
 tableView.delegate = self
 tableView.rowHeight = 70
 tableView.separatorStyle = .singleLine
 tableView.separatorColor = .blue
 //tableView.allowsMultipleSelection = true
 self.view.addSubview(tableView)
 }

 func numberOfSections(in tableView: UITableView) -> Int
 {
 return 1
 }

 func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
 {
 return 15
 }

 func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
 {
 var cell: UITableViewCell!

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = MyTableViewCell(style: .default, reuseIdentifier: "mycell")
 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)

 let switcher = UISwitch()
 switcher.tag = indexPath.row
 switcher.sizeToFit()
 cell.accessoryView = switcher
 cell.textLabel!.numberOfLines = 5
 cell.textLabel!.textAlignment = .center
 switcher.addTarget(self, action: #selector(onSwitcher(sender:)), for: .valueChanged)
 }
 }
}
```

```

let myCell: MyTableViewCell = cell as! MyTableViewCell
myCell.textLabel!.text = cities[indexPath.row].0
myCell.accessoryView!.tag = indexPath.row
(myCell.accessoryView as! UISwitch).isOn = cities[indexPath.row].1

return cell
}

@objc
func onSwitcher(sender: UISwitch)
{
 cities[sender.tag].1 = sender.isOn
}
}

```

Böylece biz artık türemiş sınıfa farklı elemanlar ekleyebiliriz. Aynı zamanda türemiş sınıfta bazı sanal metodları override edebiliriz.

### UITableView Satırları İçin Register İşleminin Yapılması

Şimdiye kadarki örneklerimizde biz dequeueReusableCellWithIdentifier(withIdentifier:) metoduyla menzil dışına çıkan nesnelerden herhangi birini alıp onu yeniden konfigüre ederek kontrole verdik. Eğer dequeueReusableCellWithIdentifier(withIdentifier:) metodu verecek nesne bulamazsa nil değerine geri dönüktür. Çünkü bu metodun geri dönüş değeri UITableViewCell? türündendir. İşte biz bu metot nil ile geri dönüp yeni bir nesne yaratıp kontrole bu nesneyi veriyorduk. Örneğin:

```

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
 var cell: UITableViewCell!

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell")
 if cell == nil {
 cell = MyTableViewCell(style: .default, reuseIdentifier: "mycell")
 <Burada ilk kez yaratılmış olan nesnenin elemanları set ediliyor>
 }
 <Burada daha önce yaratılmış olan nesnenin de elemanları set ediliyor>

 return cell
}

```

Biz UITableView sınıfının register(\_:forCellReuseIdentifier:) isimli metodunu çağrıp bu metodun birinci parametresinde ilgili sınıfın Type nesne referansını verirsek artık dequeueReusableCellWithIdentifier(withIdentifier:for:) isimli metot eğer kuyrukta UITableViewCell nesnesi yoksa bize kendisi yaratıp bu nesneden verebilmektedir. dequeueReusableCellWithIdentifier(withIdentifier:for:) metodu bu durumda default style ile bize yeni UITableViewCell nesnesini yaratır. Eğer biz nesnenin başka bir style ile yaratılmasını istiyorsak UITableViewCell sınıfından türetme yapıp style'ı türemiş sınıfta değiştirmemiz gereklidir. Örneğin:

```

override func viewDidLoad()
{
 super.viewDidLoad()

 tableView = UITableView(frame: self.view.bounds)
 tableView.dataSource = self
 tableView.delegate = self
 tableView.rowHeight = 70
 tableView.separatorStyle = .singleLine
 tableView.separatorColor = .blue
 tableView.register(UITableViewCell.self, forCellReuseIdentifier: "mycell")
}

```

```
 self.view.addSubview(tableView)
}
```

İşte bu register işleminden sonra artık dequeReusableCell(withIdentifier:for:) metodu eğer kuyrukta boşta bir UITableViewCell nesnesi yoksa onu kendisi oluşturup bize verecektir. Örneğin:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
 var cell: UITableViewCell

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell", for: indexPath)

 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)

 let switcher = UISwitch()
 switcher.tag = indexPath.row
 switcher.sizeToFit()
 cell.accessoryView = switcher
 cell.textLabel!.numberOfLines = 5
 cell.textLabel!.textAlignment = .center
 switcher.addTarget(self, action: #selector(onSwitcher(sender:)), for: .valueChanged)

 cell.textLabel!.text = cities[indexPath.row].0
 cell.accessoryView!.tag = indexPath.row
 (cell.accessoryView as! UISwitch).isOn = cities[indexPath.row].1

 return cell
}
```

Bu kod biraz daha sade olmakla birlikte bir dezavantaja sahiptir. Biz burada dequeReusableCell(withIdentifier:for:) metodunun kuyruktaki bir nesneyi mi verdigini yoksa yeni bir nesneyi mi verdigini anlayamiyoruz. Bu nedenle yalnızca ilk kez yapılacak işlemleri de her defasında yapmak durumunda kalıyoruz. Ancak yine de programcı manuel yolla birtakım elemanları kullanarak nesnenin ilk kez oluşturulduğunu anlayabilir. Örneğin UITableViewCell nesnesinin içerisindeki UILabel kontrolünün text kısmına bakılabilir:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
 var cell: UITableViewCell

 cell = tableView.dequeueReusableCell(withIdentifier: "mycell", for: indexPath)

 if cell.textLabel!.text == nil {
 cell.textLabel!.textColor = .red
 cell.textLabel!.font = UIFont(name: "Arial", size: 20)

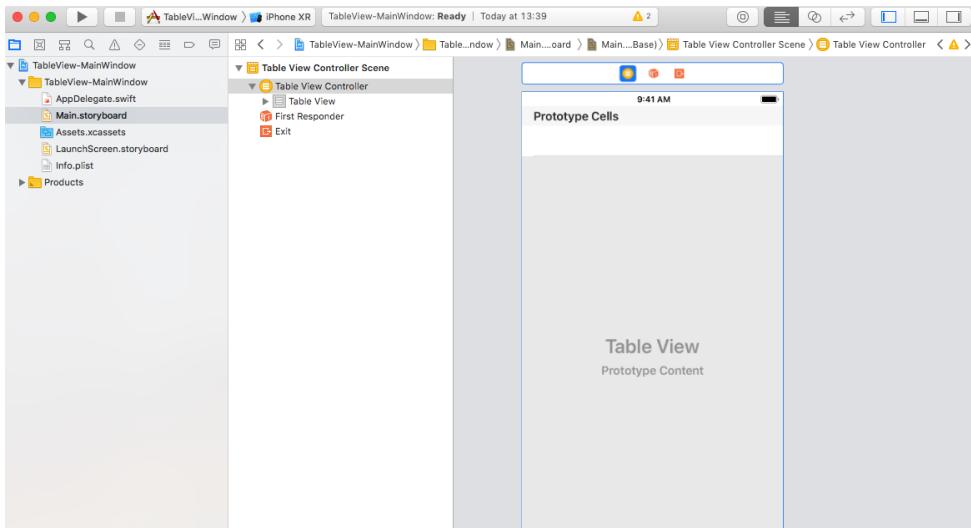
 let switcher = UISwitch()
 switcher.tag = indexPath.row
 switcher.sizeToFit()
 cell.accessoryView = switcher
 cell.textLabel!.numberOfLines = 5
 cell.textLabel!.textAlignment = .center
 switcher.addTarget(self, action: #selector(onSwitcher(sender:)), for: .valueChanged)
 }
 cell.textLabel!.text = cities[indexPath.row].0
 cell.accessoryView!.tag = indexPath.row
 (cell.accessoryView as! UISwitch).isOn = cities[indexPath.row].1
```

```
 return cell
}
```

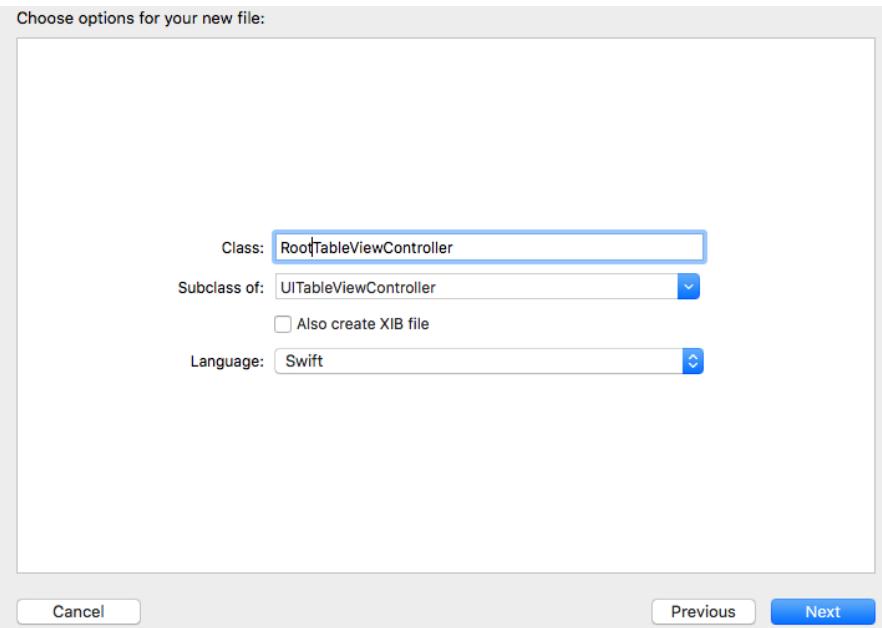
## UITableView Nesnesinin Ana Pencere Halinde Oluşturulması

Biz şimdiye kadar UITableView nesnesini ana pencerenin bir alt penceresi (Subview'su) olacak biçimde yarattık. Şimdi programımızın ana penceresinin UITableView olmasını sağlayacağız. Bu işlem adım adım şöyle yapılmaktadır:

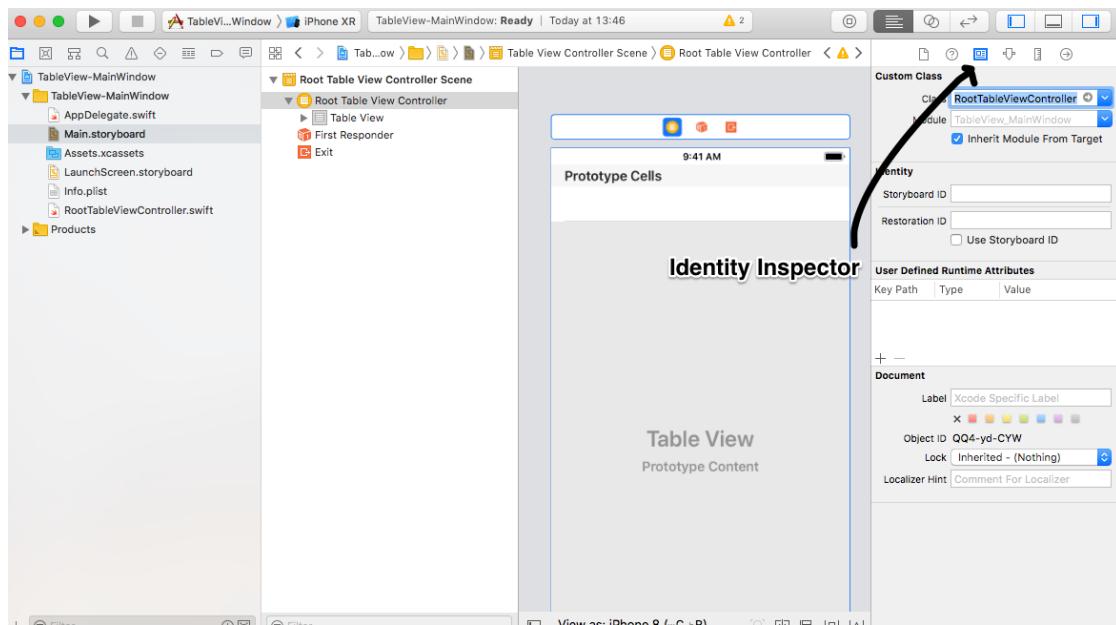
- 1) Önce bir single view app uygulaması yaratılır.
- 2) Mademki bizim ana penceremiz artık UIView sınıfından değil UITableView sınıfından oluşturulacaktır. O halde XCode'un oluşturmuş olduğu ViewController.swift dosyasının bir işlevi kalmayacaktır. Bu nedenle bu dosya silinmelidir.
- 3) Sonra Main.Storyboard dosyasını açıp buradaki View nesnesini tamamen silmek gereklidir. Öyle ki artık görüntünün canvas kısmı boş olmalıdır.
- 4) Bundan sonra Object Library'den "TableView Controller" nesnesi sürüklerek canvas'a bırakılmalıdır. Görüntü aşağıdaki gibi olacaktır:



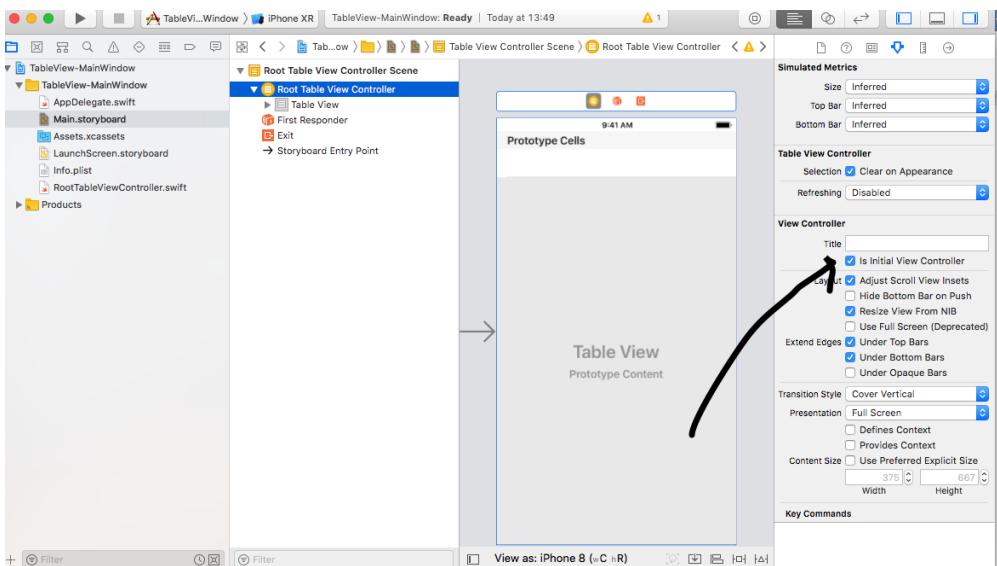
- 5) Henüz bir TableView Controller görseli storyboard'ta XML olarak olarak oluşturulmuştur ancak bunun için bir controller sınıfı yaratılmamıştır. Şimdi bizim bir controller sınıfını UITableViewController sınıfından türeterek oluşturmamız gereklidir. Bunun için New/File/Cocoa Touch Class seçilir. Sınıfıa bir isim verilir. Örneğin biz bu ismi "RootTableViewController" olarak verebiliriz.



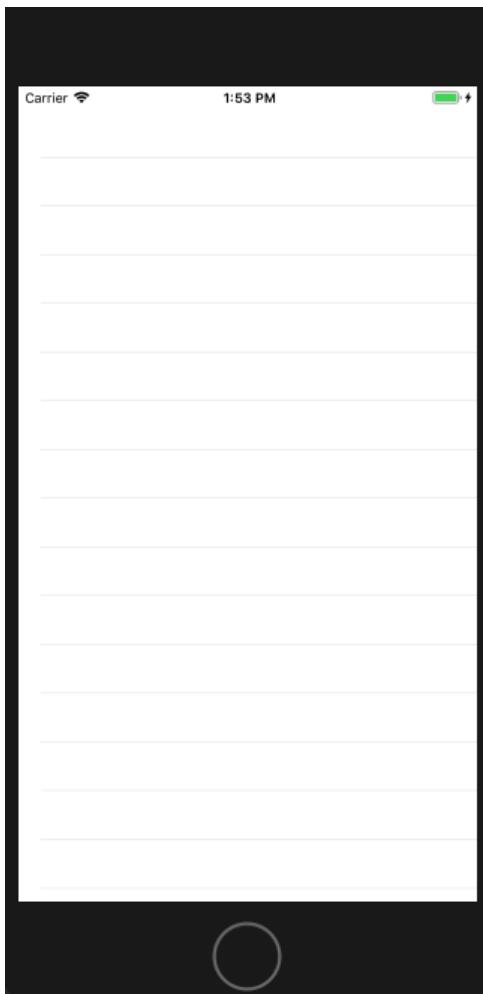
Bu işlemden sonra yaratılan sınıfın Main.storyboard'daki görsel nesneyle ilişkilendirilmesi gerekmektedir. Bunun için Main.storyboard seçilerek "inspector" panelinden "Identity Inspector" a gelinir. Orada sınıf ismi olarak yarattığımız isim girilir.



Bundan sonra "Attribute Inspector" dan söz konusu controller nesnesinin "initial controller" olmasını sağlamalıyız.

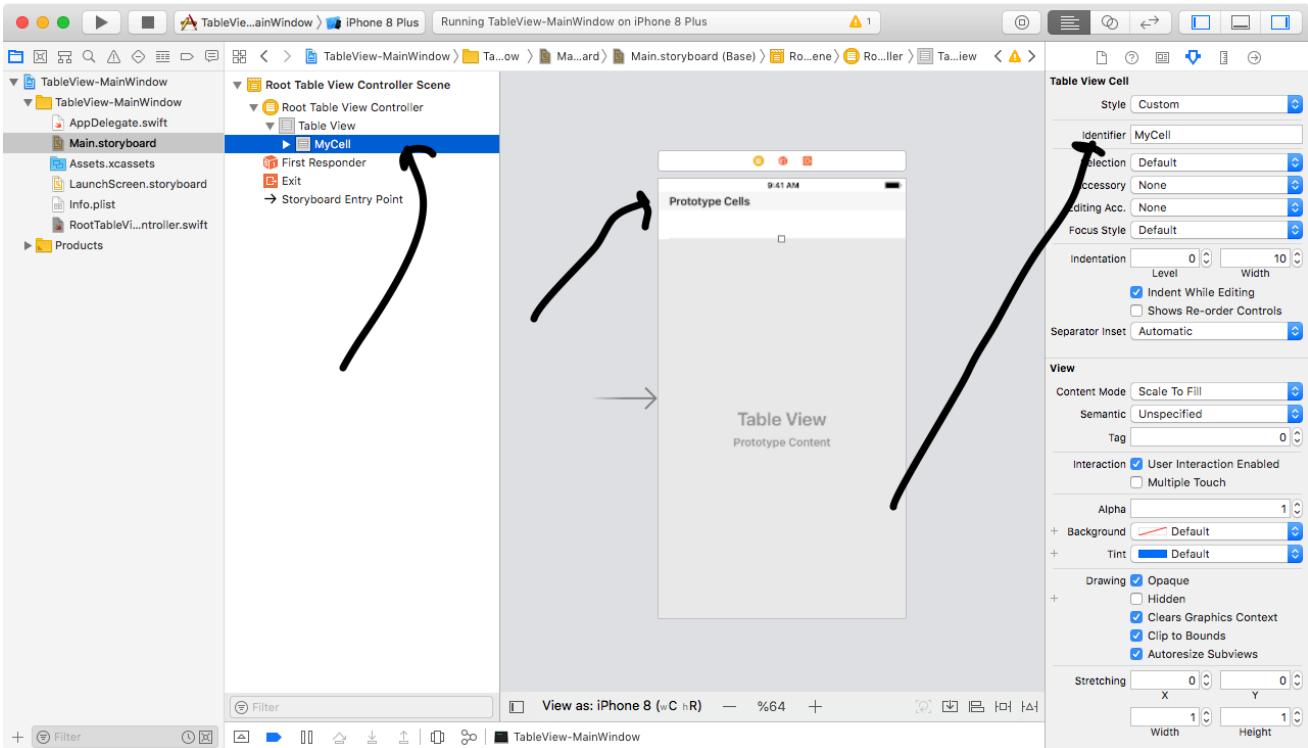


6) Artık proje çalıştırılır. Karşımıza boş bir ana penceresi UITableView olan ekran gelecektir:



7) Artık bizim UITableView nesnesi için bir UITableViewCell kalıbı oluşturmamız gereklidir. Bu kalıp görsel biçimde oluşturulabilmektedir. Ancak biz bu aşamada yalnızca görsel biçimde bir prototip olarak oluşturulmuş olan satırı ( hücreye ) bir isim vermemiz gereklidir. Bunun için Main.storyboard'ta ilgili prototip UITableViewCell nesnesi seçilir. Sonra "Attribute

"Inspector"da buna "Identifier" alanında isim verilir:



Bu biçimde biz storyboard yoluyla hücre prototipi oluşturduğumuzda bu hücrenin istenilen style'da yaratılmasını sağlayabiliriz.

8) Oluşturduğumuz UITableView nesnesinin içi boştur. Bizim bunun içini doldurmamız için aslında UITableViewDelegate ve UITableViewDataSource protokollerini desteklememiz gerekmektedir. Çünkü UITableViewController sınıfı zaten bu yeteneğe sahiptir. Hatta XCode bize bu sınıfı oluştururken ilgili metotları yazıp yorum satırı içerisine almıştır. Artık bizim bu yorum satırlarını açıp oraya küçük kod eklememiz gereklidir. Burada önemli olan nokta bu hücrenin zaten Controller sınıfı tarafından "register" ettirilmiş olduğunu. Dolayısıyla bizim yalnızca hücreleri dequeueReusableCellReusableCell(\_:cellForRowAt:) metodıyla alıp konfigüre etmemiz gereklidir.

```

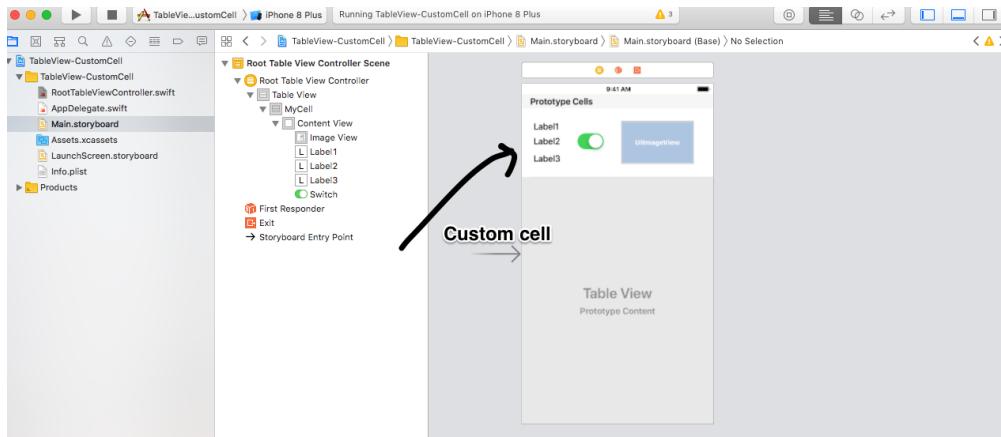
24
25 override func numberOfSections(in tableView: UITableView) -> Int {
26 // #warning Incomplete implementation, return the number of sections
27 return 1
28 }
29
30 override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
31 // #warning Incomplete implementation, return the number of rows
32 return 100
33 }
34
35
36 override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
37 }
38
39 let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell", for: indexPath)
40
41 if cell.tag == 0 {
42 cell.tag = 1
43 cell.imageView!.image = UIImage(named: "WhiteKing")
44 }
45 cell.textLabel!.text = "\(indexPath.row)"
46
47 return cell
48
49

```

## UITableViewCell Nesnelerinin Storyboard'da Custom Oluşturulması

Her ne kadar UITableViewCell nesneleri için birkaç style hazır bulundurulmuşsa da bu style'lar aslında yetersizdir. Programcı hücreleri istediği gibi customize edebilmektedir. Bunun storyboard'da yapılabilmesi için tek gereken şey görsel olarak prototip hücreyi oluşturmak ve onu bir UITableViewCell sınıfından türetilmiş sınıfta ilişkindirmektedir. Bunun için UITableView oluşturulduktan sonra şu sırasıyla işlemler yapılır:

- 1) Storyboard'ta prototype cell alanına Object Library'den istenildiği gibi kontroller girilir.



Ancak buradaki hücrenin yüksekliği otomatik olarak table view'da görülmeyecektir. Bizim yüksekliği ayrıca ayarlamamız gereklidir. Anımsanacağı gibi satırların yüksekliği UITableView sınıfında bütünsel olarak ya da UITableViewCell sınıfında hücre temelinde değiştirilebilmektedir.

- 2) Prototipi oluşturulmuş olan UITableViewCell görüntüsünün UITableViewCell sınıfından türetilmiş bir sınıfta ilişkilendirilmesi gerekmektedir. Bu ilişkilendirme için önce UITableViewCell sınıfında türetme yapılır. Sonra yine storyboard'ta Identity Inspector'da sınıf ismi girilir.

- 3) Oluşturulan storyboard nesnesindeki kontroller için ürettiğimiz sınıfta outlet'ler oluşturmamız gereklidir. Bu işlem yine

"Asistant Editor" ile yapılabilmektedir.

4) Hücrelerin yüksekliğini yaralayabilmek için iki yol vardır. Birincisi UITableView sınıfının rowHeight property'sine yeni değeri atamaktır. Bilindiği gibi controller sınıfları zaten kendi içerisinde view sınıflarının referanslarını tutarlar. Yani örneğimizde RootTableViewController sınıfının tableView isimli property'si controller'ın tuttuğu UITableView nesnesini vermektedir. Örneğin:

```
class RootTableViewController: UITableViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 self.tableView!.rowHeight = 130
 }
 //...
}
```

Satırları ayarlamanın ikinci yolu UITableViewDelegate protokolündeki tableView(\_:heightForRowAt:) metodunu override etmektir. Böylece biz her satırı farklı bir yükseklikle ayarlayabiliriz. Çünkü kontrol bir satırı göstereceği zaman bu metodu çağırılmaktadır. Bu metottan elde ettiği değeri satır yükseliği olarak ayarlamaktadır. Örneğin:

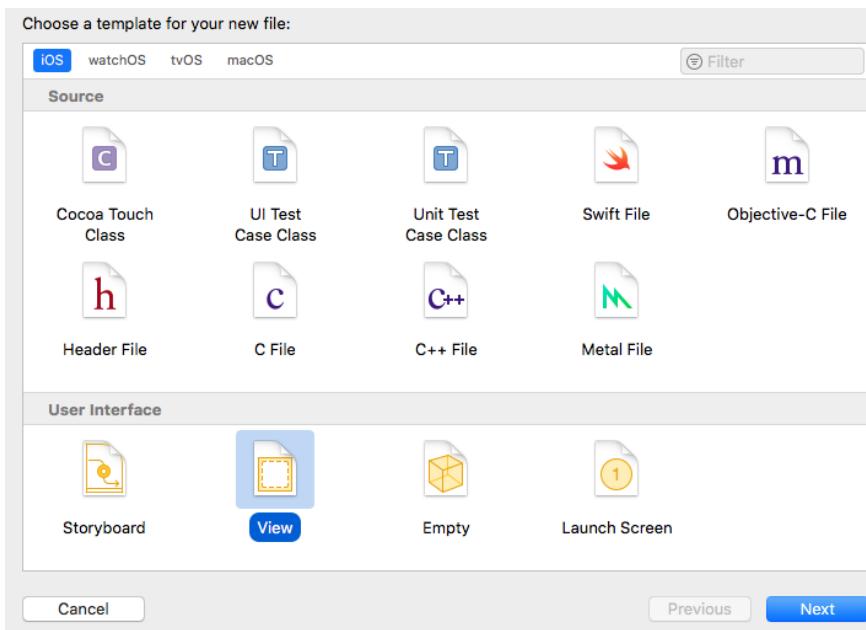
```
override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat
{
 return 130
}
```

### NIB Editör Kullanılarak UITableViewCell Tasarımı

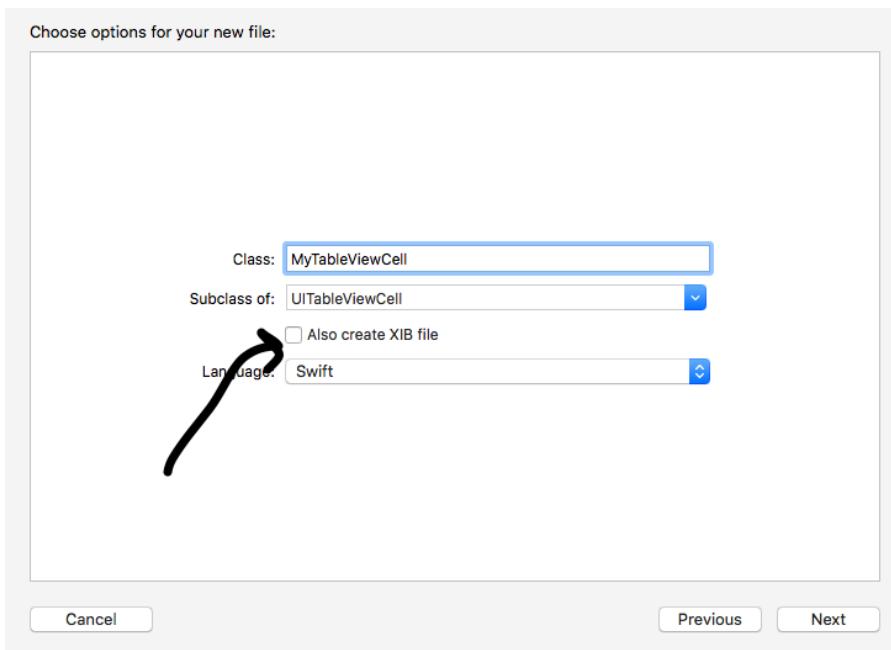
Storyboard kullanımından önce aslında NIB denilen IB arayüzü görsel tasarımda kullanılıyordu. Apple storyboard isimli ana ekran için kullanılan aracı geliştirince NIB editörü pek çok durumda gereksiz kalmaya başladı. Ancak hala NIB gereken durumlar da vardır. Biz belli hücreyi, NIB kullanarak da görsel biçimde de tasarlayabiliriz. Ancak uygulamada artık hücre tasarımlı için daha çok storyboard tercih edilmektedir. Ancak eğer TableView ana ekran için tasarlanarsa yani dışarıda yaratılmışsa burada storyboard yerine NIB kullanılması gereklidir.

NIB kullanılarak hücre tasarımlının tipik adımları şöyledir:

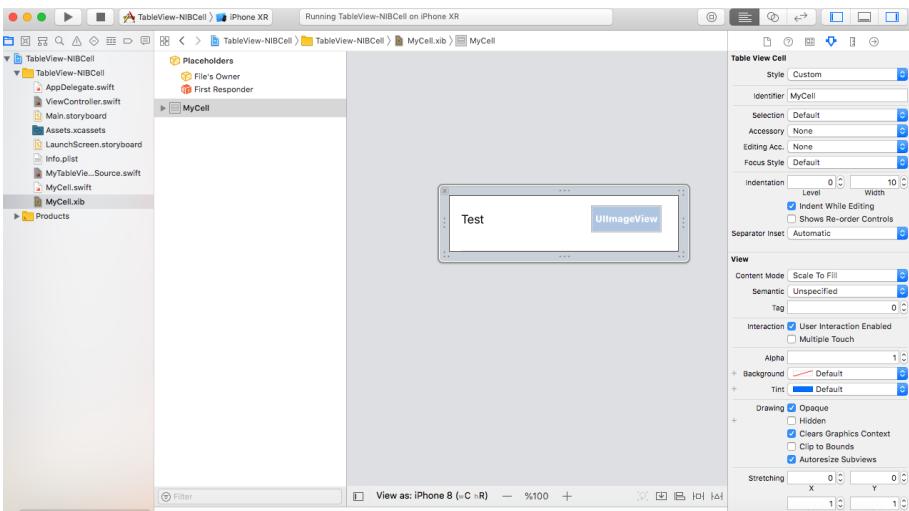
- 1) Proje yaratılır ve projeye bir UITableView nesnesi elle ya da storyboard yoluya eklenir.
- 2) Bundan sonra File/New diyalog penceresinden User Interface / View seçilir. Bu bize NIB editörünü açacaktır. Burada oluşturulan XML dosyası uzantısı .xib olacak biçimde save edilir.



3) Şimdi yine storyboard'ta yaptığımız gibi UITableViewCell sınıfından sınıf türetip bu sınıfı NIB ile ilişkilendiririz. Bu ilişkilendirme işlemi yine Identity Inspector yoluyla yapılmaktadır. Aslında önce NIB dosyası oluşturmak yerine önce sınıf türettiğimizde zaten IB bize "bunun için bir xib dosyasının yaratılıp yaratılmayacağını sormaktadır". Bu yaklaşım aslında daha kolaydır.



4) Artık Asistan editörüne geçip NIB içerisindeki hücreye elemanları yerleştirip UITableViewCell sınıfından türettiğimiz sınıf'a outlet'leri yerleştirebiliriz.



5) Şimdi register işlemini yapmamız gereklidir. Bu işlem yine ana view controller sınıfında yapılabilir. register işlemi sırasında NIB dosyası belirtilmelidir. Yani buradaki register metodu register(\_:forCellReuseIdentifier) imzalı metottur. Örneğin:

```
class ViewController: UIViewController {

 @IBOutlet weak var tableView: UITableView!
 let dds = MyTableViewCellDelegateDataSource()

 override func viewDidLoad() {
 super.viewDidLoad()

 self.tableView.frame = self.view.bounds
 self.tableView.delegate = dds
 self.tableView.dataSource = dds
 self.tableView.rowHeight = 150
 self.tableView.register(UINib(nibName: "MyCell", bundle: nil), forCellReuseIdentifier: "MyCell")
 }
}
```

6) Artık Delege ve data source sınıfımızda klasik metotları yazmalıyız. Örneğin:

```
class MyTableViewCellDelegateDataSource: NSObject, UITableViewDelegate, UITableViewDataSource {
 func numberOfSections(in tableView: UITableView) -> Int {
 return 1
 }
 func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
 return 100
 }

 func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
 let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell", for: indexPath) as! MyCell

 if cell.tag == 0 {
 cell.tag = 1
 cell.myImageView.image = UIImage(named: "WhiteKing")
 }
 }
}
```

```

 cell.myLabel.text = String(indexPath.row)
 }
}

```

## UITableView Kontrolünün Section Düzenlemesi

Biz şimdije kadar UITableView nesnemizde tek bir section kullandık. Halbuki kontrol birden fazla section'a sahip olabilmektedir. Bunun için yapılacak şey şunlardır:

- 1) UITableViewDelegate protokolündeki numberOfSections(in:) metodundan section sayısı ile geri dönülmelidir.
- 2) UITableViewDataSource protokolündeki tableView(\_:numberOfRowsInSection:) fonksiyonunda da ilgili section için o section'daki hücre sayısı ile geri dönülmelidir. Kontrol bu metodu section sayısı kadar çağrıp tüm section'ların içeriisndeki hücre sayılarını bize sormaktadır.
- 3) Artık UITableViewDataSource protokolündeki tableView(\_:cellForRowAt:) metodunda biz ilgili section için ilgili hücre ile geri dönmeliyiz. Bu metotta cellForRowAt etiketine ilişkin indexPath parametresinin iki elemanı vardır: row ve section. İşte biz bu section değerine bakarak ilgili hücreyi oluşturmalıyız.

Tabii eğer biz ana pencereye UITableViewController aracılığı ile bir UITableView nesnesi yerleştirmişsek yukarıdaki metodlar bizim UITableViewController sınıfından türettiğimiz sınıfta override edilmelidir. XCode zaten bu türetme sırasına önemli override metodları yorum satırı içerisine alarak oluşturmuş olmaktadır. Doğrudan bunun üzerinde edit işlemi yapabiliriz. Örneğin:

```

import UIKit

class RootTableViewController: UITableViewController {
 var sectionData: [[String]] = [
 ["Ali", "Veli", "Selami", "Ayşe", "Fatma", "Kerem"],
 ["Kiraz", "Kayısı", "Karpuz", "Armut", "Elma", "Muz"],
 ["Adana", "İzmir", "Malatya", "Eskişehir", "Ankara", "Bolu"]
]

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }

 // MARK: - Table view data source

 override func numberOfSections(in tableView: UITableView) -> Int
 {
 return sectionData.count
 }

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

 override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
 let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell", for: indexPath)

```

```

 if cell.tag == 0 {
 cell.accessoryType = .disclosureIndicator
 }

 cell.textLabel!.text = sectionData[indexPath.section][indexPath.row]

 return cell
 }

/*
// Override to support conditional editing of the table view.
override func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
 // Return false if you do not want the specified item to be editable.
 return true
}
*/
/*
// Override to support editing the table view.
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
 if editingStyle == .delete {
 // Delete the row from the data source
 tableView.deleteRows(at: [indexPath], with: .fade)
 } else if editingStyle == .insert {
 // Create a new instance of the appropriate class, insert it into the array, and add a
new row to the table view
 }
}
*/
/*
// Override to support rearranging the table view.
override func tableView(_ tableView: UITableView, moveRowAt fromIndexPath: IndexPath, to: IndexPath) {

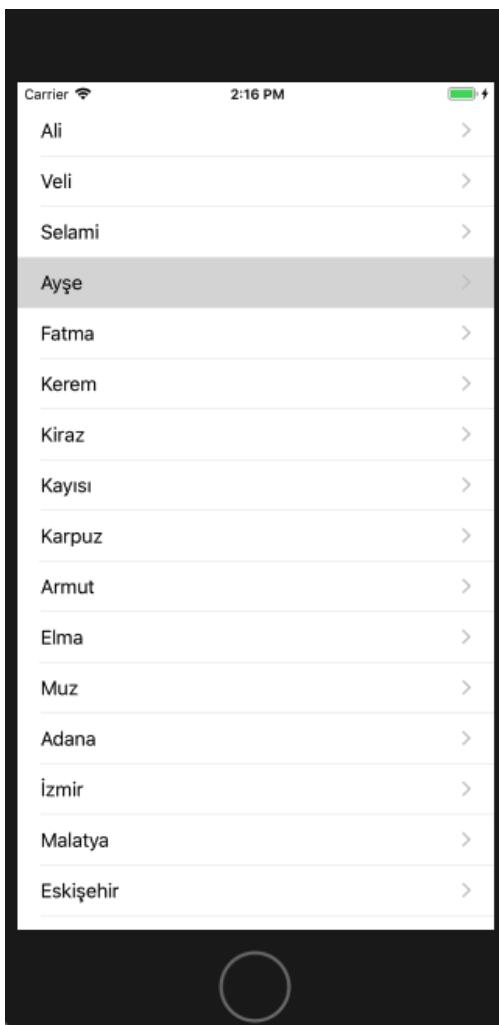
}
*/
/*
// Override to support conditional rearranging of the table view.
override func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath) -> Bool {
 // Return false if you do not want the item to be re-orderable.
 return true
}
*/
/*
// MARK: - Navigation

// In a storyboard-based application, you will often want to do a little preparation before
navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
 // Get the new view controller using segue.destination.
 // Pass the selected object to the new view controller.
}
*/

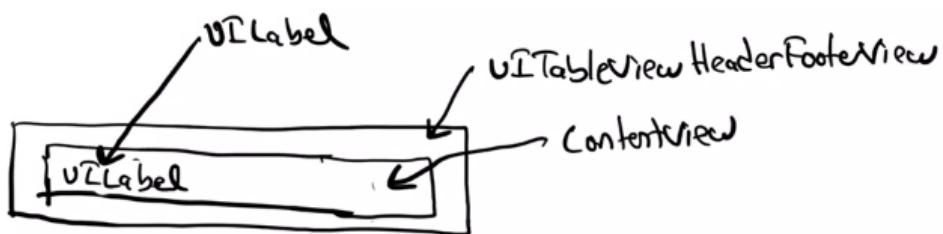
```

}

Yukarıdaki örnekte tüm section'ların bilgileri birer yazı biçiminde String türünden dizisi dizisine yerleştirilmiştir. (Swift'te çok boyutlu dizi olmadığını anımsayınız.) Bu örneğin ekran görüntüsü şöyle olacaktır:



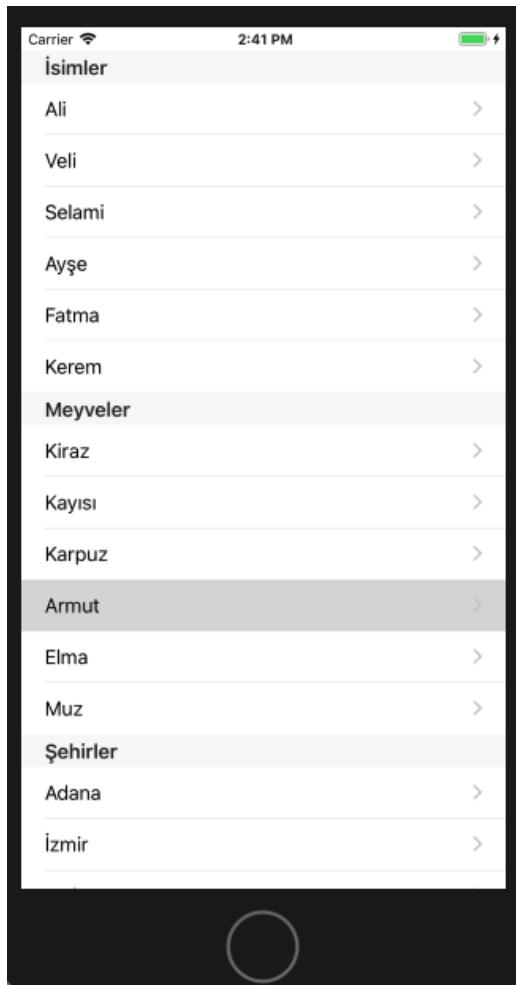
Buradaki sorun section'ların nerede başlayıp nerede bittiğinin belli olmamasıdır. İşte aslında section'ların "header" ve "footer" alanları vardır. Bu header ve footer alanları UITableViewHeaderFooterView isimli bir sınıf türündendir. Bu sınıf da aslında default style'lı UITableViewCell sınıfına benzemektedir.



Pekiyi biz bu görünütüyü nasıl oluştururuz? İşte eğer custom bir tasarım yapmayacaksak yani yalnızca buradaki UILabel nesnesine yazı set edeceksek bu işlem oldukça kolaydır. Tek yapılacak şey UITableViewDataSource protokolündeki tableView(\_:titleForHeaderInSection:) ve tableView(:titleForFooterInSection) metodlarından bir section header ya da footer yazısı ile geri dönmektir. Bu yazı yukarıdaki şekilde gördüğünüz UILabel kontrolünün içerisinde yerleştirilecektir. Örneğin:

```
override func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?
{
 return sectionHeaderTitle[section]
}
```

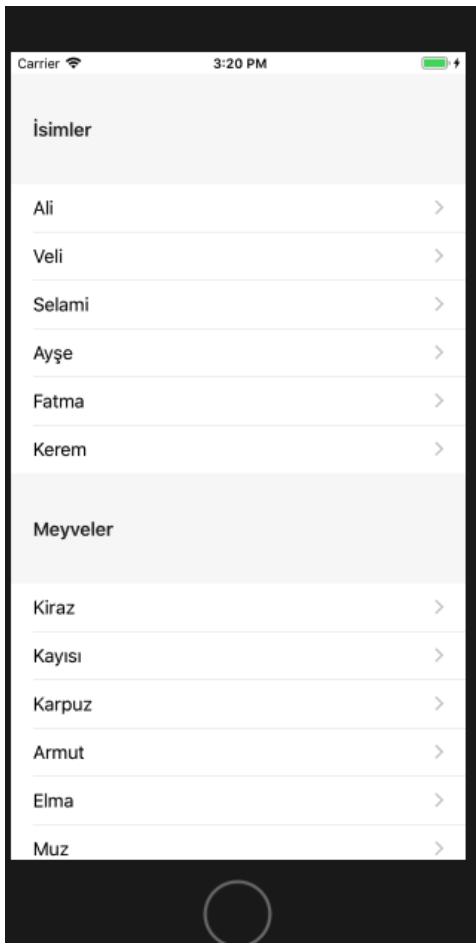
Burada section başlıklarını sectionHeaderTitle isimli bir dizide toplanmıştır ve bu dizideki ilgili elemana geri dönülmüştür. Görüntü şuna benzer olacaktır:



Bu görüntüde header kısımları hücrelerle aynı boyuttadır. Genellikle onun biraz daha yüksek yapılması tercih edilir. İşte header yüksekliğini değiştirmek için UITableViewDelegate protokolündeki tableView(\_:heightForHeaderInSection:) metodunun footer yüksekliğini değiştirmek isetableView(\_:heightForFooterInSection) metodundan yükseklik degeir ile geri dönülmesi gerekir. Tabii eğer biz UITableViewController sınıfı kullanmışsa k bu controller sınıfında bu metotları override etmeliyiz. Örneğin:

```
override func tableView(_ tableView: UITableView, heightForHeaderInSection section: Int) ->
CGFloat
{
 return 100
}
```

Şimdiki görüntü şöyle olacaktır:



Pekiyi bu header yalnızca bir yazı mı içermektedir? Default durumda evet. Eğer biz buraya başka kontroller yireceksek customize işlemi yapmalıyız. Bunun da birkaç yolu vardır.

UITableViewDelegate protokolündeki `tableView(_:willDisplayHeaderView:forSection:)` isimli metot her header görüntüleneceği zaman `tableView(_:willDisplayFooterView:forSection:)` metodu da her footer görüntüleneceği zaman çağrılmaktadır. İşte biz bu metotta metoda view parametresi olarak geçirilen view'a ekleme yapabiliyoruz. Aslında buradaki view UITableViewHeaderFooterView tüyünden bir nesnedir. Örneğin başlık kısmının sağına bir resim eklemek isteyelim. Bunun için UIImageView nesnelerini yaratıp `tableView(_:willDisplayHeaderView:forSection:)` metodunda contentView içerisine ekleyebiliriz. Örneğin:

```
override func tableView(_ tableView: UITableView, willDisplayHeaderView : UIView, forSection section: Int) {
 let hfv = willDisplayHeaderView as! UITableViewHeaderFooterView
 hfv.addSubview(sectionData[section].imageView)
}
```

Aslında UITableView uygulamalarında her section'ın bilgilerini bir yapıda oluşturup section'ları bir yapı dizisi biçiminde saklamak pratik bir yöntemdir. Tabii UITableView içerisinde çok fazla satır ve section varsa bu yöntem yerine dinamik bir oluşturma tercih edilir. Yukarıdaki örneğin eşdeğeri şöyle yazılabilir:

```
import UIKit

struct SectionData {
 var title: String
```

```

var rowTexts: [String]
var imageView: UIImageView

init(_ title: String, _ rowTexts: [String], _ imageName: String)
{
 self.title = title
 self.rowTexts = rowTexts

 self.imageView = UIImageView(image: UIImage(named:imageName))
 self.imageView.frame = CGRect(x:300, y: 27, width: 50, height: 50)
}
}

class RootTableViewController: UITableViewController {
 var sectionData: [SectionData] = [SectionData("İsimler", ["Ali", "Veli", "Selami", "Ayşe", "Fatma", "Kerem"], "WhiteKing"), SectionData("Meyveler", ["Kiraz", "Kayısı", "Karpuz", "Armut", "Elma", "Muz"], "WhiteQueen"), SectionData("Şehirler", ["Adana", "İzmir", "Malatya", "Eskişehir", "Ankara", "Bolu"], "WhiteRook")]

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }

 override func numberOfSections(in tableView: UITableView) -> Int
 {
 return sectionData.count
 }

 override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
 {
 return sectionData[section].rowTexts.count
 }

 override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
 let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell", for: indexPath)

 if cell.tag == 0 {
 cell.accessoryType = .disclosureIndicator
 }

 cell.textLabel!.text = sectionData[indexPath.section].rowTexts[indexPath.row]

 return cell
 }

 override func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?
 {
 return sectionData[section].title
 }

 override func tableView(_ tableView: UITableView, heightForHeaderInSection section: Int) -> CGFloat
 {

```

```

 return 100
 }

 override func tableView(_ tableView: UITableView, willDisplayHeaderView : UIView, forSection section: Int)
 {
 let hfv = willDisplayHeaderView as! UITableViewHeaderFooterView
 hfv.addSubview(sectionData[section].imageView)
 }

/*
// Override to support conditional editing of the table view.
override func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
 // Return false if you do not want the specified item to be editable.
 return true
}
*/
/*
// Override to support editing the table view.
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
 if editingStyle == .delete {
 // Delete the row from the data source
 tableView.deleteRows(at: [indexPath], with: .fade)
 } else if editingStyle == .insert {
 // Create a new instance of the appropriate class, insert it into the array, and add a
new row to the table view
 }
}
*/
/*
// Override to support rearranging the table view.
override func tableView(_ tableView: UITableView, moveRowAt fromIndexPath: IndexPath, to: IndexPath) {

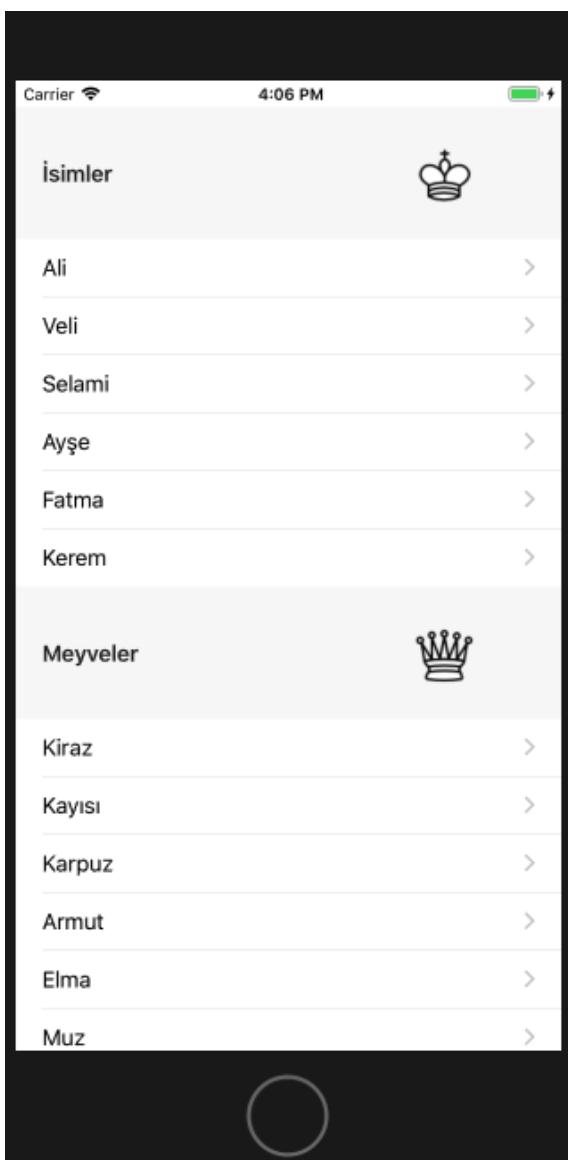
}
*/
/*
// Override to support conditional rearranging of the table view.
override func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath) -> Bool {
 // Return false if you do not want the item to be re-orderable.
 return true
}
*/
/*
// MARK: - Navigation

// In a storyboard-based application, you will often want to do a little preparation before
navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
 // Get the new view controller using segue.destination.
 // Pass the selected object to the new view controller.
}
*/

```

}

Programın görüntüsü şöyle olacaktır:



Pekiyi header footer alanının seminini nasıl boyarız? Bunun için UITableViewHeaderFooterView nesnesinin zeminini boyamayınız. Çünkü bu zemin zaten contentView penceresiyle kaplıdır. Bizim contentView penceresinin zeminini boyamamız gereklidir. Örneğin:

```
override func tableView(_ tableView: UITableView, willDisplayHeaderView : UIView, forSection section: Int)
{
 let hfv = willDisplayHeaderView as! UITableViewHeaderFooterView
 hfv.contentView.backgroundColor = .red
}
```

Ayrıca UITableViewDelegate protokolündeki tableView(\_:viewForHeaderInSection:) ve

`tableView(_:viewForFooterInSection:)` isimli metodlar bir UIView nesnesiyle geri dönmektedir. Programcı bu metodları yazıp bir UIView nesnesi ile geri dönerse artık header ya da footer kontrolünün kendisinin yarattığı UITableViewHeaderFooterView nesnesi ile değil programcının geri döndürdüğü nesneyle oluşturulacaktır. Bu metottan UITableViewHeaderFooterView nesnesi ile geri dönülmesi gerekmektedir. Sıradan bir UIView nesnesi ile geri dönülebilir. Örneğin:

```
override func tableView(_ tableView: UITableView, viewForHeaderInSection section: Int) -> UIView? {
 let view = UIView()
 view.backgroundColor = .yellow

 return view
}
```

## Swipe İşlemi İle TableView Hücrelerinin Silinmesi

Eskiden bu işlemi yapmak için biraz kod yazmak gerekiyordu. Ancak belli bir zaman önce bu işlem daha pratik hale getirilmiştir. Tek yapılacak işlem controller sınıfında ya da Delege sınıfında aşağıdaki metodu override etmektedir:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
 if editingStyle == .delete {
 toDoItems.remove(at: indexPath.row)
 tableView.reloadData()
 }
}
```

## Modal Diyalog Pencerelerinin Kullanımı

Modal diyalog pencereleri açıldığı zaman kapatmadıktan sonra arka plan etkileşiminin mümkün olmadığı pencerelerdir. iOS'ta iki temel diyalog penceresi "alert" ve "action sheet" pencereleridir. Tipik bir alert ya da action sheet pencereleri penceresi sırasıyla şu adımlardan geçilerek oluşturulur:

1) UIAlertController sınıfı türünden bir nesne yaratılır. Bu sınıfın init metodu init(title:message:preferredStyle:) biçimindedir. Buradaki title alert penceresindeki ana başlığı, message mesaj yazısının kendisini ve preferredStyle da mesaj penceresinin türünü belirtmektedir. Bu preferredStyle .alert ya da .actionSheet biçiminde olabilir. Yani modal diyalog pencereleri kendi aralarında "alert" pencereleri ve "actionSheet" pencereleri biçiminde ikiye ayrılmaktadır.

```
let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle: .alert)
```

2) Şimdi sıra action nesneleri yaratıp controller'a eklemeye gelmiştir. Action nesneleri birer düğmeye görsel olarak temsil edilmektedir. Ayrıca her action nesnesi bir metoda bağlanabilemektedir. action nesnesini controller'a eklemek için UIAlertController sınıfının addAction isimli metodu kullanılır. Bu metot addAction(\_) parametreik yapısına sahiptir. Bizden bir UIAlertAction nesnesi ister. UIAlertAction nesnesi de sınıfın init(title:style:handler:) metodu ile yaratılır. Metodun birinci parametresi düğmenin başlık yazısını, ikinci parametresi stilini belirtir. Stil şunlardan biri olabilmektedir:

```
.default
.cancel
.destructive
```

Son parametre ilgili düğmeye basıldığında çağrılacak metodu belirtir. Bu metodun parametresi UIAlertAction türünden olmalıdır. Bu parametre hangi düğmeye basılmışsa o düğmeye ilişkin UIAlertAction nesnesini belirtir. Tabii bu parametre nil de geçilebilir. Bu durumda düğmeye basıldığında herhangi bir metot çağrılmayacaktır. Örneğin diyalog penceremize üç düğme ekleyelim:

```
ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: nil))
ac.addAction(UIAlertAction(title: "No", style: .default, handler: nil))
ac.addAction(UIAlertAction(title: "Cancel", style: .default, handler: nil))
```

3) Şimdi artık sıra diyalog penceresinin görüntülenmesine gelmiştir. Bunun için UIViewController sınıfının present metodu çağrılır. Bu metod present(\_:animated:completion:) biçiminde üç parametreye sahiptir. Metodun birinci parametresi gösterilecek diyalog penceresine ilişkin UIAlertController nesnesini alır. İkinci parametre açımın animasyonlu olup olmadığını belirtmektedir. Bu parametre Bool türden True ya da False değerini alır. Üçüncü parametre default nil değerini almıştır. Diyalog penceresi kapandığında çalıştırılacak kodu belirtir. Bu parametreyi girmeyebiliriz. Örneğin:

```
self.present(ac, animated: true)
```

Bu durumda bir düğmeye basıldığında bir alert penceresi çıkartan örnek şöyle olabilir:

```
@IBAction func buttonOkClickHandler(_ sender: Any)
{
 let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle: .alert)

 ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: nil))
 ac.addAction(UIAlertAction(title: "No", style: .default, handler: nil))
 ac.addAction(UIAlertAction(title: "Cancel", style: .default, handler: nil))

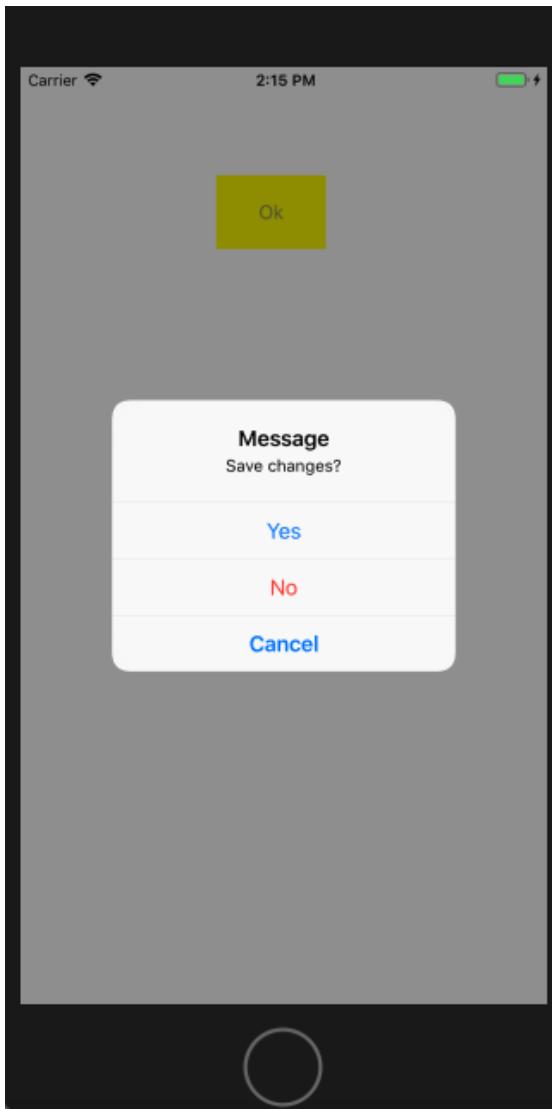
 self.present(ac, animated: true)
}
```

UIAlertAction nesnelerindeki style parametresinin anlamı şöyledir: .destructive bir bilgi kaybının olabileceğini temsil etmektedir. .cancel ise bir işlemin iptal edilebilirliği ile ilgiliidir. .default diğer durumları temsil etmektedir. Genel olarak IOS .cancel style'ına sahip düğmeyi en aşağıda göstermektedir. .destructive düğmesi ise kırmızı görüntülenmektedir. Yine düğmelerden biri bold ve mavi görüntülenir. Biz bu düğmeyi değiştirebiliriz. Örneğin:

```
@IBAction func buttonOkClickHandler(_ sender: Any)
{
 let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle: .alert)

 ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: nil))
 ac.addAction(UIAlertAction(title: "No", style: .destructive, handler: nil))
 ac.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: nil))

 self.present(ac, animated: true)
}
```



Bir düğmeye tıklandığında biz belirli bir işlemin yapılmasını sağlayabiliriz. Bunun için UIAlertAction nesnesi yaratılırken handler vermek gereklidir. Örneğin:

```
import UIKit

class ViewController: UIViewController {

 override func viewDidLoad() {
 super.viewDidLoad()
 // Do any additional setup after loading the view, typically from a nib.
 }

 @IBAction func buttonOkClickHandler(_ sender: Any)
 {
 let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle:
.alert)

 ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "No", style: .destructive, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: alertActionHandler))

 self.present(ac, animated: true)
 }
}
```

```

}

func alertActionHandler(action: UIAlertAction)
{
 print(action.title!)
}
}

```

UIActionController sınıfının preferredAction isimli property'si action nesnelerinden birini alır. Onu bold olarak gösterir. Bu controller sınıfının actions isimli property elemanları bir dizi biçiminde tüm action nesnelerini tutmaktadır. Örneğin:

```

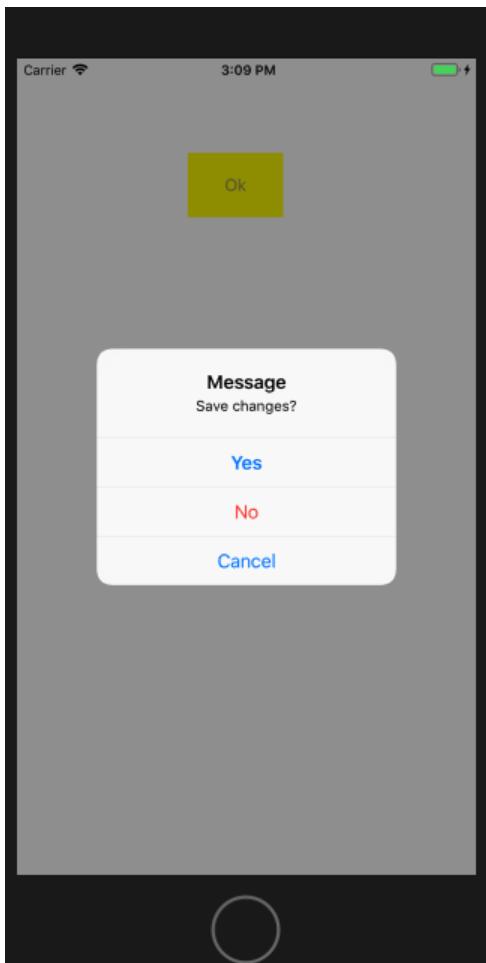
@IBAction func buttonOkClickHandler(_ sender: Any)
{
 let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle: .alert)

 ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "No", style: .destructive, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: alertActionHandler))

 ac.preferredAction = ac.actions[0];

 self.present(ac, animated: true)
}

```



Aslında Alert penceresine textField de eklenebilir. Bunun için UIAlertController sınıfının addTextField(\_:) isimli metodu kullanılır. Bu metod seçeneksel biçimde parametresi UITextField olan ve geri dönüş değeri bulunmayan bir fonksiyon ya

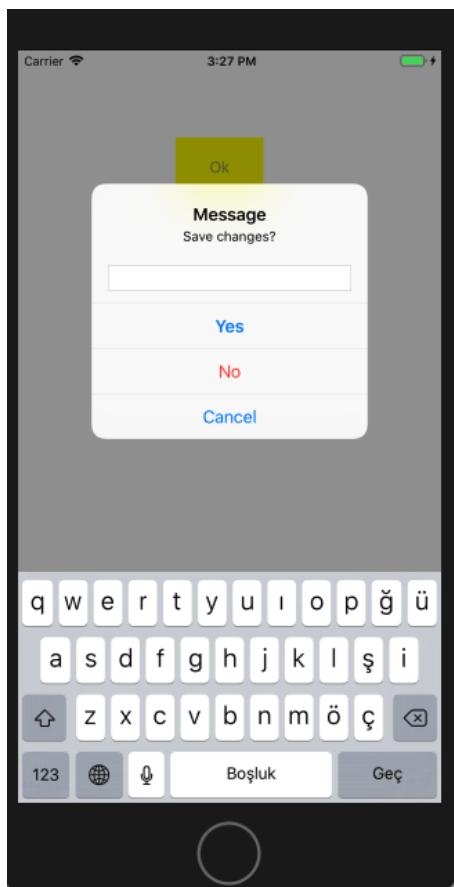
da metodu parametre olarak almaktadır. Bu parametrenin default durumu nil biçimdedir. Bu parametre tipik olarak ilgili text field seçildiğinde görüntülenecek klavye düzeninin belirlenmesi ve diğer bazı gerekli olabilecek işlemlerin yapılabilmesi için düşünülmüştür. Örneğin:

```
@IBAction func buttonOkClickHandler(_ sender: Any)
{
 let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle: .alert)

 ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "No", style: .destructive, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: alertActionHandler))
 ac.addTextField()

 ac.preferredAction = ac.actions[0];

 self.present(ac, animated: true)
}
```



text field yaratılırken klavye seçenekini şöyle belirleyebiliriz:

```
@IBAction func buttonOkClickHandler(_ sender: Any)
{
 let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle:
.actionSheet)

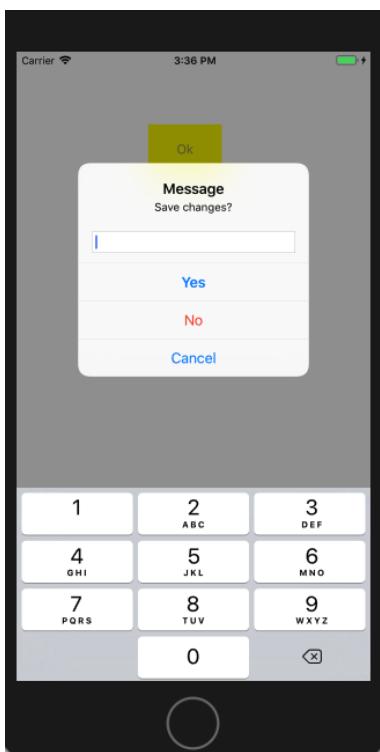
 ac.addAction(UIAlertAction(title: "Yes", style: .default, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "No", style: .destructive, handler: alertActionHandler))
 ac.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: alertActionHandler))
 ac.addTextField(configurationHandler: {tf in tf.keyboardType = .numberPad})
```

```

ac.preferredAction = ac.actions[0];
self.present(ac, animated: true)
}

```

Burada artık text field seçildiğinde nümerik bir klavye görüntüsü çıkacaktır.



Eğer fonksiyonun son parametresi fonksiyon türündense doğrudan küme parantezleri ile closure girilebildiğini anımsayınız. Yani yukarıdaki işlemle aşağıdaki Swift'te eşdeğerdir:

```
ac.addTextField {tf in tf.keyboardType = .numberPad}
```

Action Sheet penceresi de alert penceresi ile benzer amaçla kullanılmaktadır. Action sheet oluşturmak için UIViewController sınıfının init metodunda preferredStyle .actionSheet olarak girilmelidir. Örneğin:

```
let ac = UIAlertController(title: "Message", message: "Save changes?", preferredStyle: .actionSheet)
```

Örneğin:

```

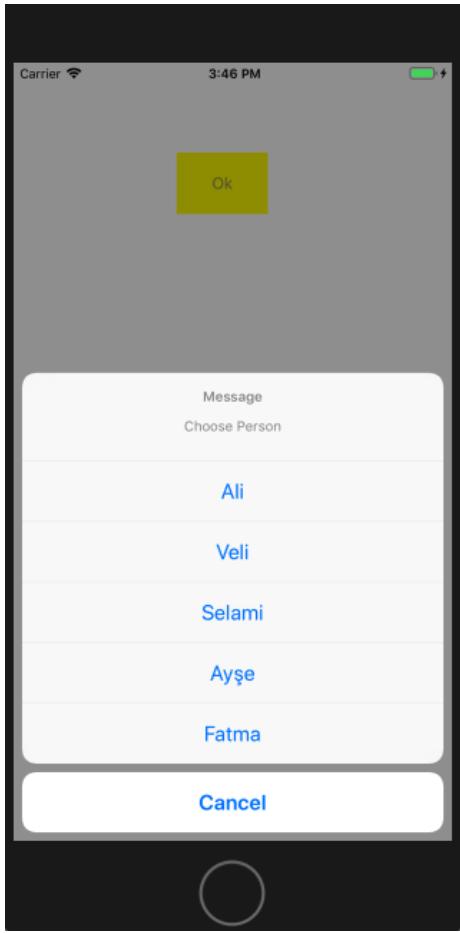
@IBAction func buttonOkClickHandler(_ sender: Any)
{
 let ac = UIAlertController(title: "Message", message: "Choose Person", preferredStyle: .actionSheet)

 for str in ["Ali", "Veli", "Selami", "Ayşe", "Fatma"] {
 ac.addAction(UIAlertAction(title: str, style: .default, handler: nil))
 }

 ac.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: alertActionHandler))
}

```

```
 self.present(ac, animated: true)
}
```



Action Sheet penceresine çok fazla düğme eklendiğinde bunlar scroll edilebilmektedir. Yani adeta kısıtlı bir UITableView etkisi oluşturulabilmektedir.

IOS'ta birkaç çeşit daha modal diyalog penceresi vardır. Bunlar daha ileride ele alınacaktır.

### **UISearchBar Kullanımı**

UISearchBar pek çok kullanıcının aşina olduğu bir kontroldür. Genellikle tek başına değil bir UITableView gibi kontrolün yukarısında bulundurulur. Kontrol manuel olarak ya da storyboard yoluyla oluşturulabilir.

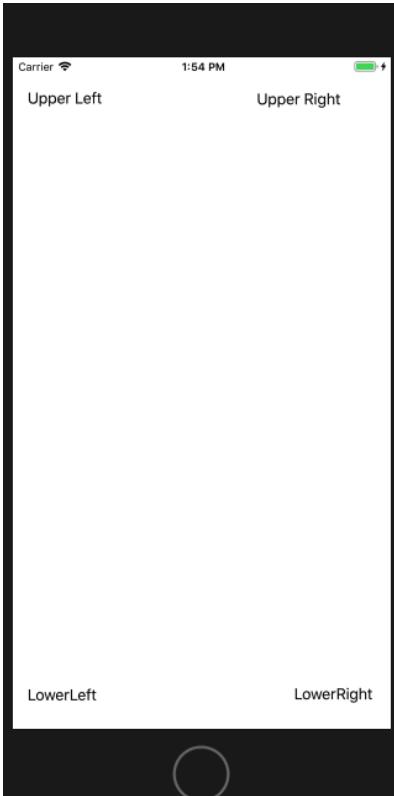
UISearchBar sınıfının text property'si search bar'daki yazıyı belirtir. plceHolder property'si yine bu text alanına hiçbir yazı girilmediğinde gösterilecek ip ucu yazısını belirtmektedir.

### **Kontrollerin Otomatik Yerleştirilmesi ve Kısıtların Oluşturulması**

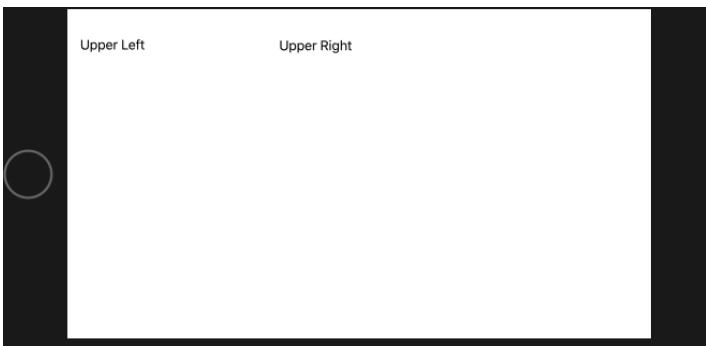
Bilindiği IOS aygıtları çeşitli boyutlarda üretilmiştir. Eskiden otomatik yerleştirme özelliği yokken programcılar IPhobe ve IPad için programları ayrı ayrı yazıyorlardı. Bu özellikten sonra artık çok çeşitli ekran boyutları ve çözünürlükleri için tek bir programın yapılması yeterli olmaktadır. Apple'ın IOS cihanları çevrilip döndürülebilmektedir. Bu işlem sırasında cihazın yönelimi (orientation) otomatik biçimde değişir. İşte kontrollerin otomatik yerleştirilmesi yalnızca farklı cihan boyaları için değil aynı zamanda yönelim değişikleri için de görüntünün otomatik biçimde düzenlenmesi amacıyla kullanılabilmektedir.

Otomatik yerleştirmenin (autolayout) neden gerektiği basit bir örnekle anlaşılabilir. Boş bir uygulamada her köşeye bir

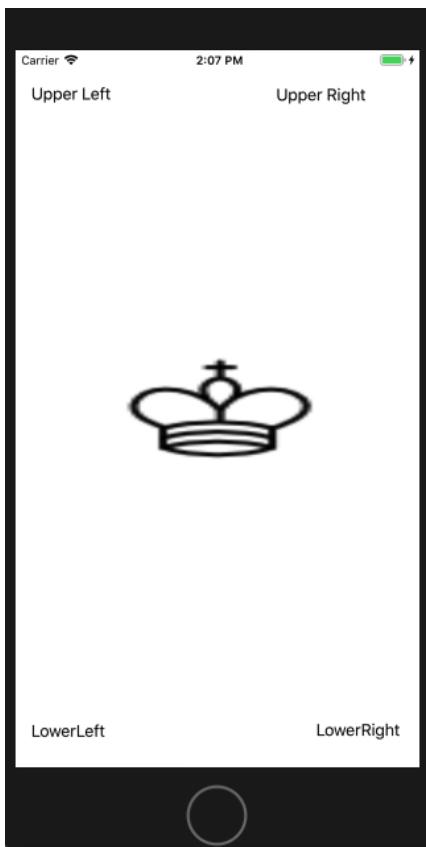
Label yerleştirmiş olalım. Cihazın boyutunu ya da yönelimini değiştirdiğimizde artık Label'lar köşelerde olmayacağı.  
Örneğin:



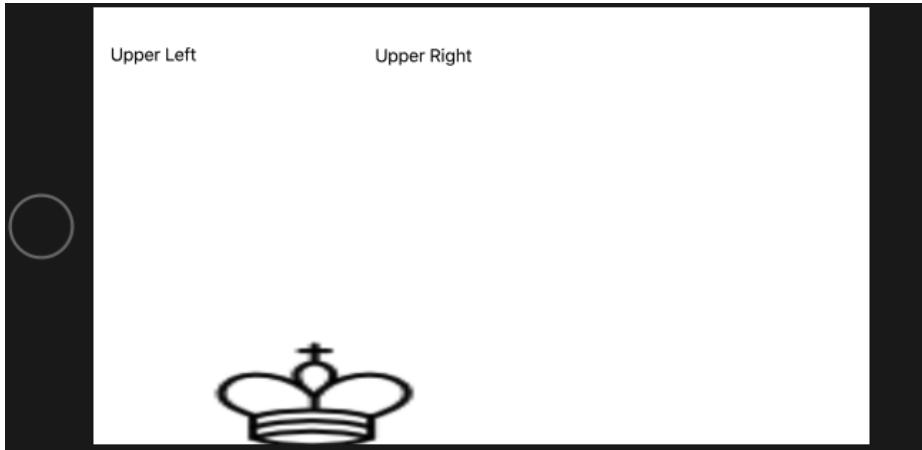
Şimdi cihazı landscape yönelimine getirelim:



İşte biz farklı boyuttaki bir aygıttı da farklı bir yönelimde de Label'ların köşelerde çıkışmasını isteyebiliriz. Bu işlem manuel yolla yapılabilir. Ancak çok zahmetlidir. İşte "autolayout" özelliği bu işlemin pratik ve zahmetsiz yapılmasını sağlamaktadır. Şimdi ekranın tam ortasında bir resim daha koyalım:



Şimdi aygıtı çevirelim:

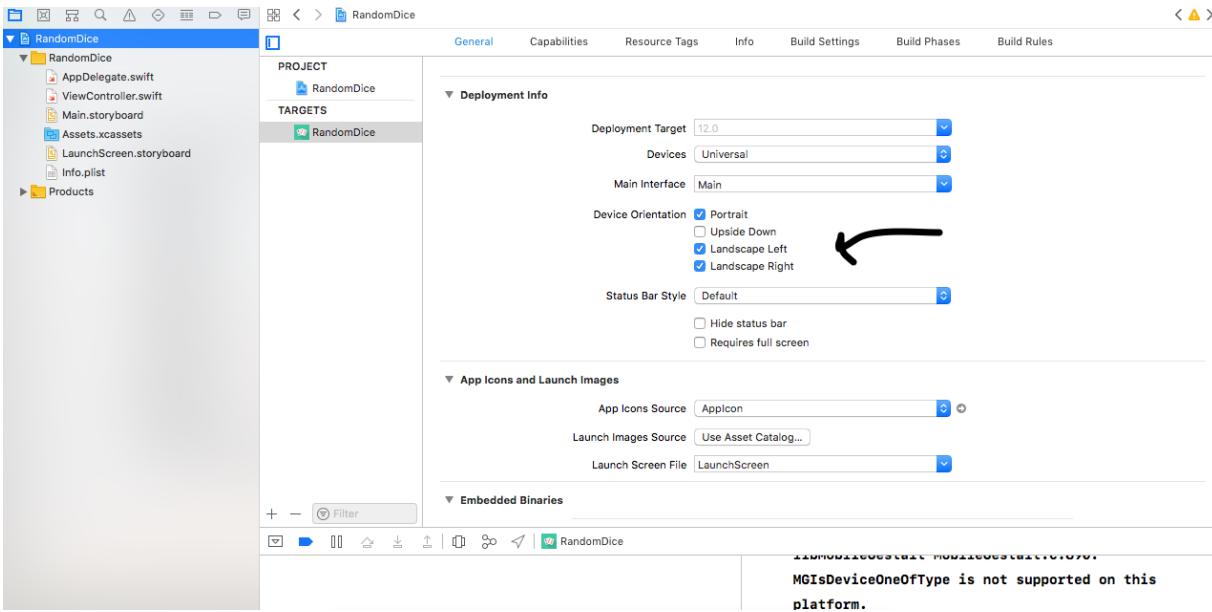


Gördüğü gibi resim ortadayken artık ortada değil. Pekiyi yönelim değişince yerler neden değişmekte? Aslında kontrollerin yerleri değişimmemektedir. Yani kontroller sol-üst köşeden yine aynı x ve y değerlerindedir. Ancak ekranın genişlik-yükseklik değerleri değişmiştir.

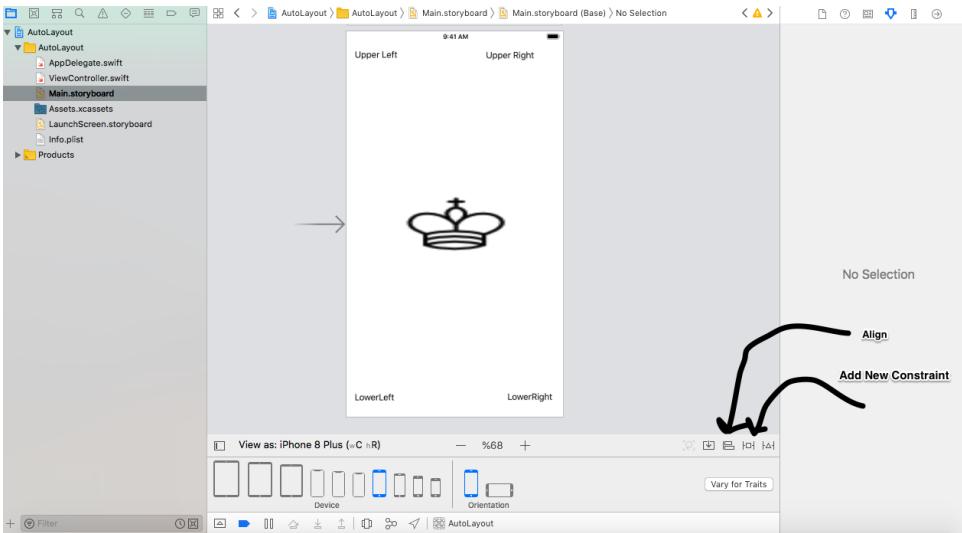
Biz çözünürlük ve yönelim değiştiğinde görüntünün otomatik olarak yeniden uygun biçimde oluşturulmasını programlama yoluyla ya da IB kullanarak sağlayabiliriz. Bu sürece "kontrollerin otomatik yerleştirilmesi (auto layout)" denilmektedir.

Aslında uygulamada cihazın hangi yönelimlere tepki vereceği baştan proje ayarlarında belirtilmektedir. Genel olarak iPhone uygulamalarının çoğu başaşağı (upside down) yönelimi dışındaki yönelimleri desteklemektedir. iPad uygulamaları ise baş aşağı yönelimini de genel olarak destekler. iPhone cihazlarının baş aşağı yönelimi desteklememesinin temel

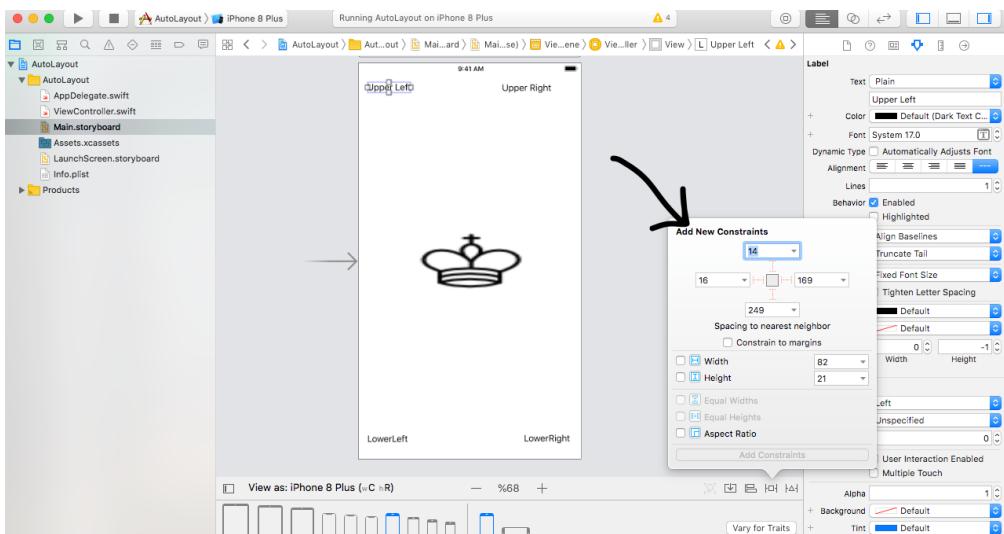
nedeni tuşlu programlarda (örneğin telefon uygulamalarında) tuşun yönelime göre ters yerde kalmasını engellemektir. Projede aygitın hangi yönelimleri destekleyeceği seçenekler kısmında belirtilmektedir. Default durum baş aşağı yönelimin dışındaki yönelimlerin desteklenmesidir.



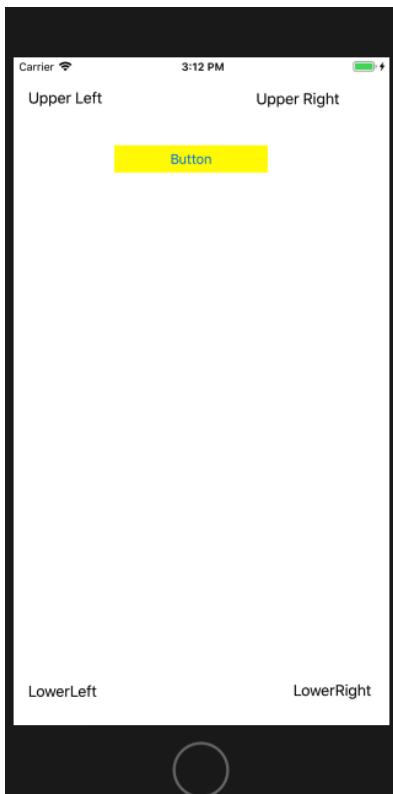
Otomatik yerleştirme kısıtlar (constraints) yoluyla yapılmaktadır. Kısıt yerlestirmek için aşağıdaki "Align" ve "Add New Constraint" tuşları kullanılmaktadır:



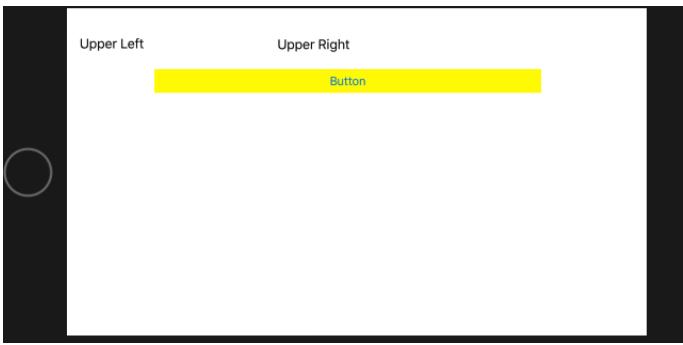
Add New Constraint işlemi yapmak için önce kontrol seçilir sonra ilgiyi düğmeye tıklanır.



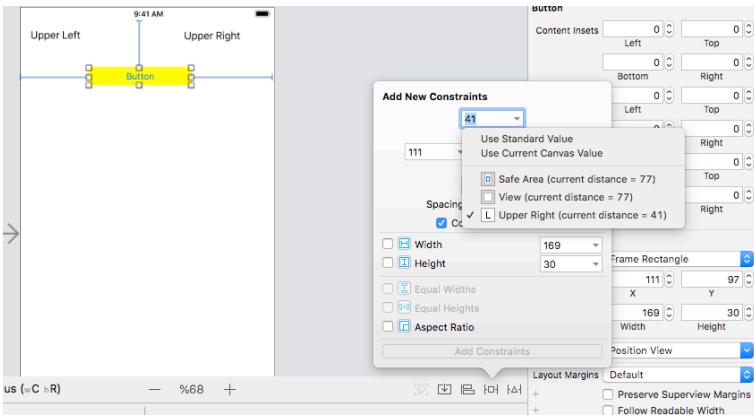
Burada dört yene demirleme değerleri belirtilebilmektedir. Buradaki değerler ilgili köşeye kontrolün ne kadar uzaklıkta olması gerektiğini belirtir. Örneğin biz kontrolümüzü yukarıya, sola ve sağa demirlersek Çözünürlük ve yönelim ne olursa olsun yukarıya, sağa ve sola hep aynı uzaklıkta kontrol görüntülenir. Tabii bunun mümkün olabilmesi için kontrolün boyutunun da otomatik değiştirilmesi gerekebilir. Örneğin bir düğmeyi yukarıya, sola ve sağa demirlemiş olalım:



Şimdi yönelimi değiştirelim:

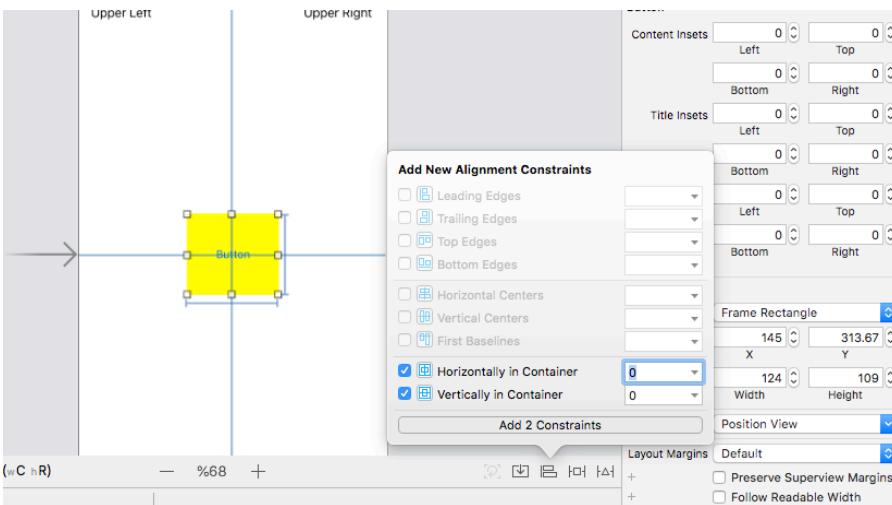


Bir kontrol demirlenirken bir başka kontrol referans alınır. Yani demirleme başka bir kontrole göre yapılmaktadır. Demirlenen hangi kontorle göre yapılacağı combobox'tan seçilebilmektedir:

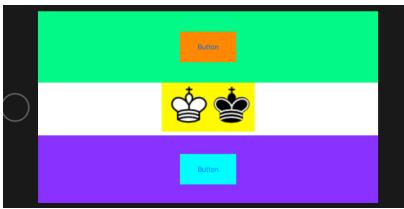
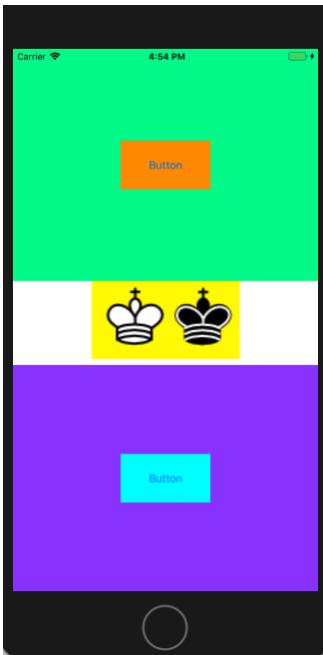


Benzer biçimde biz de yukarıdaki dört Label'ı uygun biçiminde köşelere demirleyebiliriz.

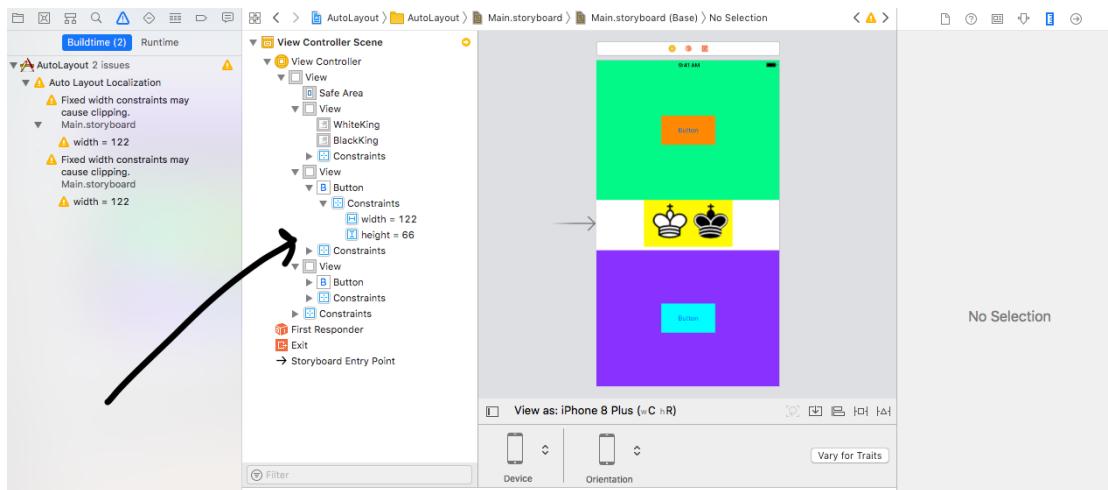
Diğer bir kısıt türü de "hızalama (alignment)" kısıtlarıdır. Örneğin biz düğmenin boyutu değişmeden onun her çözümürlükte ve yönelimde ekranın tam ortasında olmasını isteyelim. Bu durumda "hızalama kısıtlarından" faydalananmaız gereklidir. Hızalama yatay ve düşey eksene göre ortalanacak biçimde yapılmamıştır. Tabii kontrol değişik boyutlarda ortalanabilir. Programcının bu durumda bir de boyut (size) kısıt vermesi uygun olur.



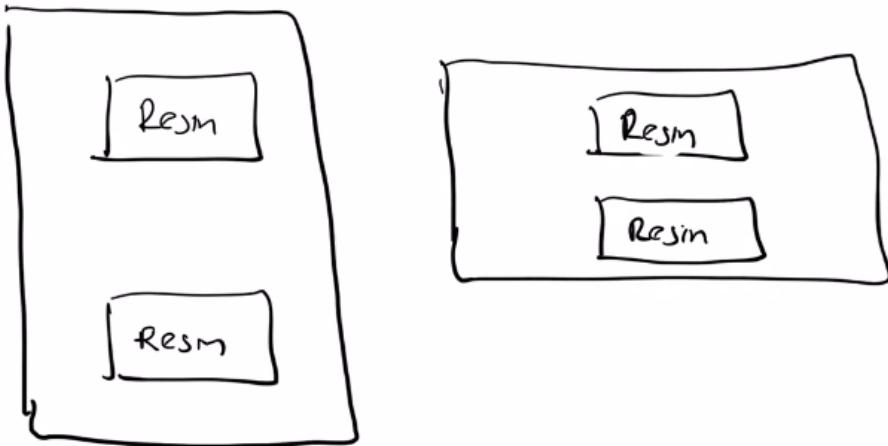
Hızalama sırasında kontrollerin tek tek hızlanması çok zahmetli bir durumdur. Bunun yerine birden fazla kontrol bir kontrolün (örneğimn tipik olarak bir view'nun) içerisinde yerleştirilerek o view hızlanmalıdır. Genel olarak karmaşık hızlamalar her zaman iç içe kontrol yerleştirerek yapılmalıdır. Örneğin ekranın ortasında yan yana iki UIImageView nesnesi onun yukarısında ve aşağısında düşey ve yatay olarak ortalanmış iki düğme yerlestireceğimizi düşünelim:



Burada en uygun tasarım üç view kullanmaktadır. Ortadaki resimler bir view'nun içerisinde yerleştirilir. Bu view yatay ve düşey içeren view'da ortalananır. Sonra yukarıdaki alan ayrı bir view içerisinde yerleştirilir. Düğme de bu view içerisinde ortalananır. Aynı şey aşağısı için de yapılır.



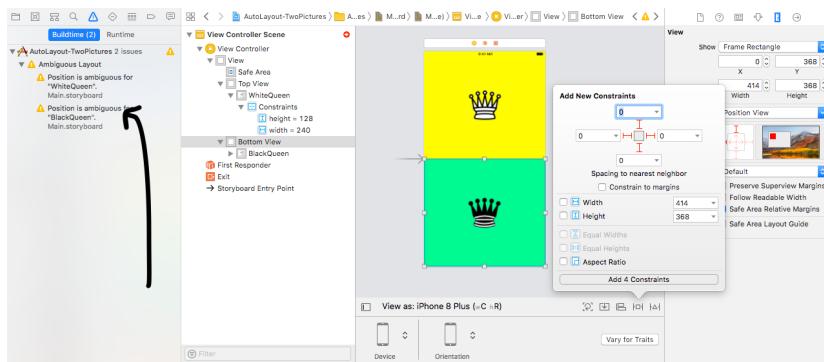
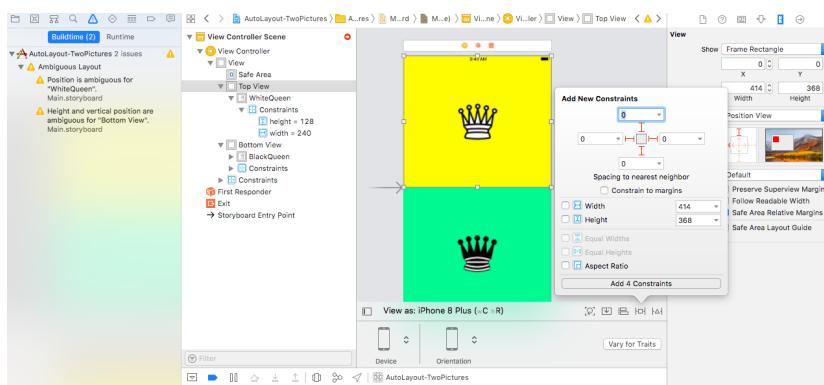
Şimdi de orta kısmı olmayan iki bölümlü view görüntüsünü hizalamaya çalışalım. Örneğin görüntü aşağıdaki gibi olsun:



Burada ekranının iki yarımlı bölgeye ayrıldığını varsayıp iki yarımlı bölgenin ortasına iki resim yerleştirilmek isteniyor.

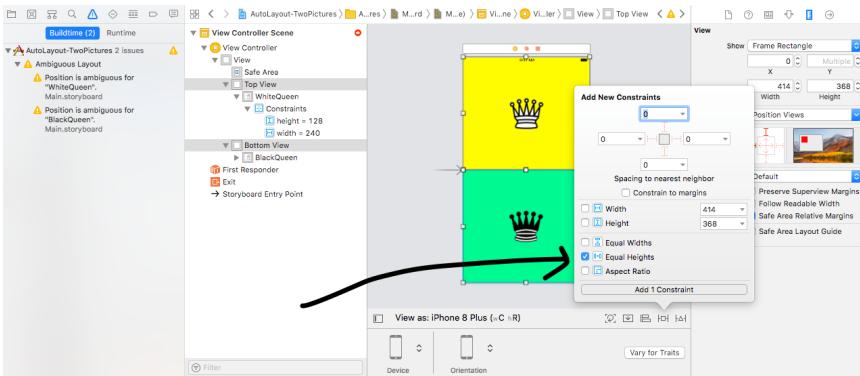
Bu işlem şöyle yapılabilir:

- 1) Her zaman bir ortalama teması varsa bir view container gerekmektedir. Bir view'ya tek bir kontrol ortalanabilir. Bu durumda örnekte iki resmin kendi alanlarına ortalanması istendiğine göre bizim de iki view oluşturmamız gereklidir. Bunlardan birine Top View diğerine Bottom View diyebiliriz.
- 2) Tasarımı kolaylaştırmak için Top View Portrait olarak ekranın yarısını kaplayacak boyuta getirilebilir. Bunun aşağısına da Bottom View yerleştirilebilir. Sonra resimler bu view'unun içerisinde ortalanmalıdır.
- 3) Top View ve Bottom View için dört köşeye demirleme yapılmalıdır. Ancak bu demirlemin yapılması iki anlamlılık oluşturur. Çünkü burada Top View ve Bottom View arasındaki ilişki belirtilmemiştir.

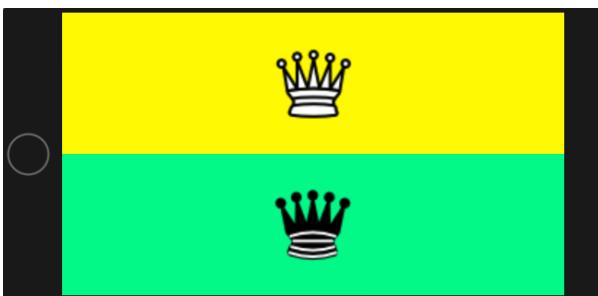
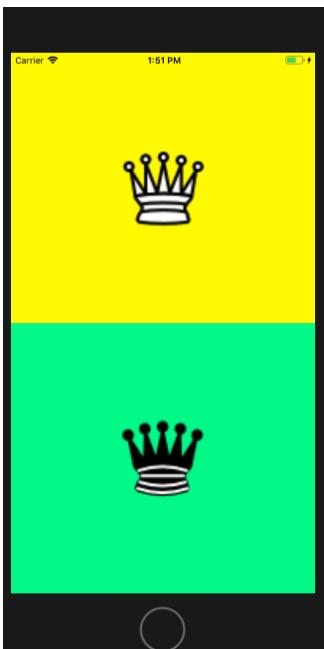


Buradaki ikianlamlılığının nedeni bu koşullar sağlandığı halde Top View ve Bottom View view'larının yüksekliklerinin tam

olarak belirlenmemesidir. İşte bu durumda "Eşit yükseklik" koşulunun da eklenmesi gereklidir. Top View ve Bottom View birlikte seçilir ve eşit yükseklik ayarlaması yapılır:



Sonuç aşağıdaki gibi olmalıdır:

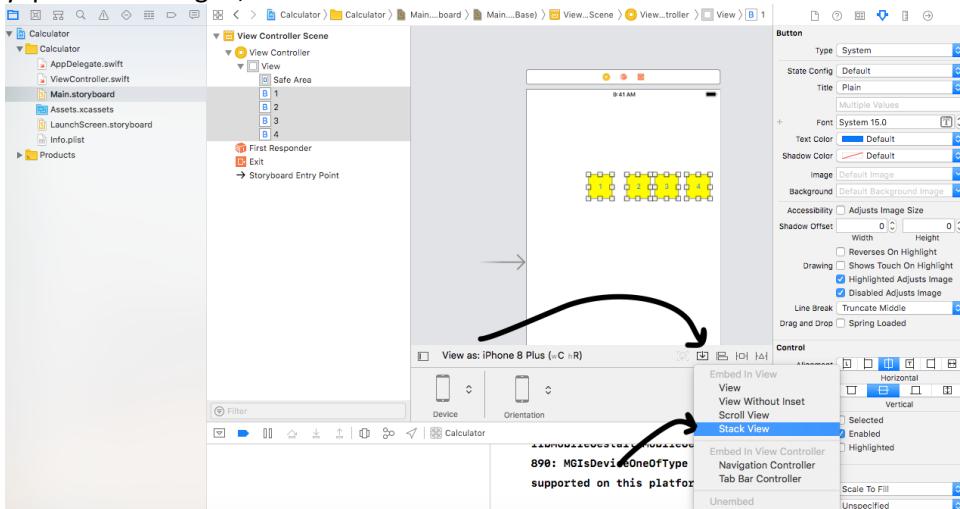


Peki burada neden "eşit genişlik" oşulunu da eklemek zorunda kalıyoruz? İşte biz Top View ve Bottom View için "sola, sağa yukarı ve aşağı 0" koşulunu verdiğimizde bu koşulları sağlayan farklı yüksekliklerde Top View ve Bottom View olabilir. Halbuki biz bunlara "eşit yükseklik" koşulunu eklediğimizde artık her şey belirlenmiş olur.

### Yatay ve Düşey StackView Kullanımı

Stackview tarz kontroller pek çok framework'te vardır. Bu kontrollerden amaç alt pencereleri hizaya sokmaktır. Yani stack view kontrolleri yatay ya da düşey olarak kendi alt pencerelerini kendi içerisinde hizalar. Böylece programcı detaylarla uğraşmak yerine bu stackview pencerelerini ana pencereye göre hizalar.

StackView pencereleri yatay ve düşey olmak üzere ikiye ayrılır. Biz Toolbox'tan yatay ve düşüey pencereleri ana pencereye yerleştirip diğerlerini bunların içine yerleştirebiliriz. Ancak onun yerine kontrolleri ana view üzerinde oluşturup sağ alttaki düğme grubunun solundaki düğmeye tıklanarak stackview seçilirse aynı işlem pratik bir biçimde yapılabilir. Örneğin;:



## IOS'ta Verilerin Kalıcılığın Sağlanması

Bir kullanıcı olarak IOS sistemlerinde uygulamaların kullandığı dosyaların birbirlerindne izole edildiğini fark etmişsinizdir. Bu tasarım kasten yapılmıştır. Bir uygulamanın başka bir uygulamanın dosyalarına erişememesi güvenilirliği artırmaktadır. Bir uygulamayı kaldırduğumızda o uygulamanın bütün dosyaları da otomatik olarak silinmektedir.

IOS'taki güncel dosya sistemi Apple Dosya Sistemi (Apple File System)'dır. Bu dosya sistemi aynı zamanda son Mac OS X sürümlerinde kullanılan dosya sistemidir. Aslında birkaç sene önce Mac OS X ve IOS HFS+ (Hierarchical File System Plus) isimli bir dosya sistemi kullanıyordu. HFS+ çok uzun süre kullanıldı. Aslında APFS (Apple File System) HFS+ dosya sisteminin biraz daha modifiye edilmiş bir biçimidir. HFS+ ve APFS asıl olarak i-node temelli UNIX/Linux sistemlerinde kullanılan dosya sistemlerine benzer bir dosya sistemidir. Aslında IOS sistemi telefonlara ya da tabletlere kurulduğunda orada normal Mac OS X sistemlerinde olduğu gibi dizinler vardır. Yalnızca IOS uygulamaların bu dizinlere erişip işlemler yapmasını tavsiye etmemektedir. Pekiyo IOS programcısı olarak biz bu dosya sistemini nasıl kullanabiliriz?

İşte Apple App Store'dan bir program IOS cihazına indirildiğinde onu asıl dosya sisteminin altında belli bir klasöre kurmaktadır. O klasörün birkaç alt klasörüne de resmi biçimde erişim hakkı vermektedir. Dolayısıyla IOS Programcılar bir dosya yaratacaklarsa ya da bir veritabanı oluşturacaklarsa bilgileri onlara sağlanan bu klasörlerde saklamalıdır. Çünkü program kaldırıldığındá Apple bu program için yaratığı klasörleri tamamen silmektedir. Başka bir deyişle ideal durumda programın kaldırılmasıyla o programın yaratmış olduğu bütün dosyalar silinmiş olmaktadır. Tabii aslında bir uygulama belirli koşullarda kendi diskinin başka klasörlerine de erişebilir. Eskiden sık uygulanan "Jailbreak" işlemleri bunu sağlamaktadır. Ya da bir uygulamanın kendi klasörü dışında erişebileceği klasörler de vardır. Fakat ne olursa olsun IOS sistemi bize ayrılan klasörler dışında dosyaların başka klasörlerde saklanması kesinlikle tavsiye etmemektedir.

Uygulama geliştirirken simülördeki programların ana makinede yükleniği yer iOS cihazındakine çok benzerdir. Örneğin bu makinede yaratılan bir projede simülördeki uygulama dizini şurasıdır:

```
file:///Users/KaanAslan/Library/Developer/CoreSimulator/Devices/A565C2AD-F0ED-4CAE-A306-2094A4D4FFAF/data/Containers/Data/Application/B0E26897-3656-42B0-ACD5-11FC55969FE6/
```

Bu uygulama klasörünün (yani uygulamanın çekildiği klasörün) altında Documents, Library, SystemData ve tmp isimli klasörler vardır. Bu dizinlere programcı erişebilmektedir. Documents dizini en resmi erişilebilir dizindir. Ancak bu dizin iTunes tarafından da dışarının kullanımına açılmıştır. iTunes kablo bağlantısı ile cihan bağlandığında bize upload dizini olarak bu dizini gösterir. Kullanıcı bu dizine iTunes kullanarak doküman yerleştirebilir. Uygulama da bu dizine bakarak dokümanları kullanabilmektedir. tmp dizini geçici dosyalar için düşünülmüş bir dizindir. Ancak buradaki dosyaların silinmesi de tamamen programcinin sorumluluğundadır. Library dizini de programcinin kullanabilecegi bir dizindir. Apple bu dizinlerde alt dizinlerin yaratılmasına da izin vermiştir. Uygulama silindiğinde tüm bu dizinlerin silineceğine dikkat ediniz.

Pekiyi biz uygulama verilerimizin kalıcılığını nasıl sağlayabiliriz? Bu konuda şunlar söylenebilir:

- Tamamen dosya işlemleriyle biz izin verilen söz konusu dizinlerde dosyalar açarak bilgilerimizi orada saklayabiliriz. Bunun için Cocoa'da dosya işlemler yapan bazı sınıflar bulunmaktadır.
- Biz Sqlite gibi gömülü bir veritabanı dosyasını bu klasörlerde oluşturabiliriz. Böylece kendi veritabanımızı cihan içerisinde tutabiliyoruz. Ancak şüphesiz SQLite gibi gömülü veritabanları çok yüksek veriler üzerinde etkin işlem yapamaktadır.
- Biz Cocoa'nın bize sunduğu "seri hale getirme (object serialization)" mekanizmasını kullanarak dolaylı biçimde bilgilerimizi dosyalarda saklayabiliriz.
- En çok kullanılan yöntemlerden biri veritabanını uzakta organize etme onlara cihazdan ya da web servislerden erişme yöntemidir. Genellikle uygulamalar cihazların disk hacimleri küçük olabildiği için veritabanlarını dışarıdan kullanmak eğilimindedir.

## IOS Sistemlerinde Dosya İşlemleri

IOS sistemlerinde dosya işlemleri için en temel sınıf FileManager isimli sınıfır. Bir FileManager nesnesi authorization belirtilerek yaratılabilir. Ya da zaten yaratılmış olan default FileManager nesnesi bir singleton biçiminde default isimli property ile elde edilebilir. Örneğin:

```
let fm: FileManager = FileManager.default
```

Şimdi biz bu FileManager nesnesi ile sınıfın metodlarını çağırarak dosya işlemlerini yapabiliriz. IOS sistemlerinde yol ifadeleri düz bir string'ten ziyade bir URL biçiminde ifade edilmiştir. URL kavramı URL isimli bir yapıyla temsil edmektedir. FileManager sınıfının dokümantasyonuna bakıldığına IOS sistemleri için söz konusu olmayan elemanların da olduğu görülebilir. Çünkü Manager sınıfı aslında Mac OS X sistemlerinde de aynı amaçla kullanılmaktadır. Bu sınıfın bazı elemanları MAC OS X sistemleri için anlamlıdır.

FileManager sınıfının bizim için en önemli metodlarından biri url isimli metottur. Bu metot bizim için ayrılmış olan yukarıda belirttiğimiz dizinlerin yol ifadelerini URL yapıs nesnesi biçiminde vermektedir. Biz bir dosya

yaratacaksak ya da bir dosyanın içlerini okuyacaksak önce söz konusu dizinlerin yol ifadelerini alıp bu yol ifadelerine ilgili dosya ismini eklememiz gerekir. url metodunun parametrik yapısı biraz detaylıdır:

```
func url(for directory: FileManager.SearchPathDirectory,
 in domain: FileManager.SearchPathDomainMask, appropriateFor url: URL?, create shouldCreate:
Bool) throws -> URL
```

Biz bu parametrelerin hepsini anlamlı biçimde girmek zorunda değiliz. Birinci parametre hangi dizinin yol ifadesini almak istediğimizi belirtmektedir. Örnek bir kullanım şöyle olabilir:

```
let fm: FileManager = FileManager.default
do {
 let url = try fm.url(for: .documentDirectory, in: .userDomainMask, appropriateFor: nil,
create: true)
 print(url.absoluteString)
}
catch {
 print("error!")
}
```

Burada biz uygulamamızın doküman dizinin (yani iTunes'un gördüğü dizinin) cihaz içerisindeki yol ifadesini almış olduk. Artık biz bu yol ifadesine bir dosya ismi ekleyip dosya işlemlerini başlatabiliriz.

Bu noktada daha detaylı olarak bir IOS cihazında hangi dizinlerin ne amaçla bulundurulduğunu ve bizim dosyalarımızı hangi dizinde oluşturmamız gereği üzerinde duralım.

**Documents Dizini:** Bu dizin yukarıda da belirtildiği gibi iTunes tarafından görülen bir dizindir. Aslında programcılar kendi IOS dosyalarını bu dizinde oluşturması her zaman önerilmez. Çünkü dosyaların iTunes ile görüntülenmesi kullanıcıların kafasını karıştırabilecektir. Tabii aslında Documents dizini proje seçenekleri kullanılarak iTunes'a kapatılabilir. Documents dizini iCloud backup'ı alınırken de backup içerisinde saklanmaktadır.

**tmp Dizini:** Bu dizin geçici dosyalar için düşünülmüştür. Bu dizin iCloud backup sırasında yedeklenmemektedir. Programcılar kalıcı olmayan dosyaları burada oluşturabilirler. Bu dizinin yol ifadesi URL olarak FileManager sınıfının temporaryDirectory metoduyla alınabilmektedir.

**Application Support Dizini:** Bu dizin programının kendi dosyalarını yerleştirmesi için en uygun dizinlerden biridir. Başlangıçta yaratılmamış durumdadır. Amcak işlem yapılacak zaman Library dizinin altında yaratılmaktadır. iCloud backup ile bu dizinin yedeklemesi yapılmaktadır.

**Caches Dizini:** Bu dizin de cache'lenecek veriler için düşünülmüştür. Library dizinin altındadır. Kullanımı tmp dizinine benzemektedir. iCloud backup ile bu diznin yedeklemesi yapılmaz.

Swift'teki temel bazı collection sınıflar aslında Objective-C'deki ismi NSXXX biçiminde olan sınıflarla uyumludur. Yani biz Swift'in bazı sınıflarını Objective-C sınıflarına dönüştürüp ters dönüşüm de yapabiliz. Swift'in String sınıfı Objective-C'nin NSString sınıfı ile, Swift'in Array sınıfı Objective-C'nin NSArray sınıfı ile, Swift'in Dictionary sınıfı Objective-C'nin NSDictionary sınıfı ile uyumludur. Objective-C'nin bu NSXXX sınıflarının bazlarında doğrudan elemanları seri hale getirip XML biçiminde bir dosyaya aktaran write metotları vardır. O halde biz de iOS'ta bazı collection nesneleri bu write metotları yoluyla doğrudan dosyalara yazabiliz.

Örneğin:

```
class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()
 let fm: FileManager = FileManager.default
 do {
 var url = try fm.url(for: .documentDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
 url.appendPathComponent("test.txt")

 try ("this is test" as NSString).write(to: url, atomically: true, encoding: String.Encoding.utf8.rawValue)
 print(url.absoluteString)
 }
 catch {
 print("error!")
 }
 }
}
```

Örneğin:

```
class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()
 let fm: FileManager = FileManager.default
 do {
 var url = try fm.url(for: .documentDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
 url.appendPathComponent("test.txt")

 let a: [Int] = [1, 2, 3, 4, 5]

 try (a as NSArray).write(to: url)
 print(url.absoluteString)
 }
 catch {
 print("file error!")
 }
 }
}
```

Burada yaratılan dosyanın içeriğine bakınız:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
 <integer>1</integer>
 <integer>2</integer>
```

```

<integer>3</integer>
<integer>4</integer>
<integer>5</integer>
</array>
</plist>

```

Örneğin:

```

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()
 let fm: FileManager = FileManager.default
 do {
 var url = try fm.url(for: .documentDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
 url.appendPathComponent("test.txt")

 let dict: Dictionary<String, Int> = ["Ali": 123, "Veli": 234, "Selami": 234]

 try (dict as NSDictionary).write(to: url)
 print(url.absoluteString)
 }
 catch {
 print("file error!")
 }
 }
}

```

Oluşturulan dosyanın içeriği şöyledir:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>Ali</key>
 <integer>123</integer>
 <key>Selami</key>
 <integer>234</integer>
 <key>Veli</key>
 <integer>234</integer>
</dict>
</plist>

```

Tabii NSString, NSArray, NSDictionary gibi sınıfların write metotları ile dosyaya yazdırduğumuz bilgileri biz bu sınıfların init metotlarıyla aynı URL'yi belirterek geri de alabiliriz. Örneğin yukarıdaki Dictionary nesnesini şöyle geri okuyabiliriz:

```

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()
 let fm: FileManager = FileManager.default

```

```

 do {
 var url = try fm.url(for: .documentDirectory, in: .userDomainMask, appropriateFor:
nil, create: true)
 url.appendPathComponent("test.txt")

 let dict = NSDictionary(contentsOf: url)
 print(dict as! Dictionary<String, Int>)
 }
 catch {
 print("file error!")
 }
 }
}

```

Diğer sınıfların kullanımı da benzerdir.

Biz yukarıda NSStringi NSArray ve NSDictionary gibi sınıfların write metotlarıyla dosyalara bir şeyler yazdık. Bu write metotları dosya varsa onun üzerine yazmaktadır. Halbuki bazı uygulamalarda dosyanın belli bir yerine bir şeylerin yazılması gerekmektedir. İşte aşağı seviyeli dosya işlemleri için en temel sınıf FileHandle isimli sınıfır. Bu sınıfı diğer programlama dillerindeki Stream sınıflarına benzetebilirisiniz. FileHandle sınıfının kullanımı şöyledir:

1) FileHandle sınıfı türünden nesne sınıfın init(forReadingFrom:), init(forWritingTo:) ya da init(forUpdating:) metotlarıyla URL belirtilerek yaratılır. Aslında String parametresi alan init metotları da vardır. Bunun için dokümanlara başvurabilirsiniz. Örneğin:

```
let fh = try FileHandle(forUpdating: url)
```

Bu init metotları dosya yoksa onu yaratmamaktadır. Yalnızca olan dosayı okuma, yazma ya da hem okuma hem de yazma amaçlı açmaktadır.

2) Yazma işlemi için write metodu, okuma işlemi için readData metodu kullanılır. write metodunun parametrik yapısı şöyledir:

```
func write(_ data: Data)
```

Metot bizden Data türünden bir nesne ister. Data isimli yapı aslında bir byte yiğinini temsil etmektedir. Bu data nesnesinin elde edilmesinin ise çeşitli yolları vardır. Örneğin pek çok Swift sınıfının içerisinde bize Data? veren (yani ilgili nesnenin verilerini seçeneksel Data yapısı biçiminde bize veren) metodlar vardır. Örneğin bu sayede biz bir string'in içerisindeki byte'ları belli bir encoding'i belirterek Data biçiminde elde edebiliriz. Sonra bunu write metodu ile yazabiliriz:

```
let fh = try FileHandle(forUpdating: url)

fh.write("this is a test".data(using: String.Encoding(rawValue: String.Encoding.utf8.rawValue))!)
```

Dosya göstericisi kavramı tabii burada da vardır. Dosya nesnesi init metoduyla ilk kez açıldığında dosya göstericisi sıfırıncı offset'tedir. Sonra yine diğer sistemlerde olduğu gibi okuma yazma yapıldığında dosya göstericisi otomatik ilerletilmektedir. seek(toFileOffset:) metodu dosya göstericisini belli bir offset'e konumlandırmak için, seekToEndOfFile() metodu ise dosya göstericisini EOF durumuna çekmek için kullanılmaktadır. Örneğin:

```

let fh = try FileHandle(forUpdating: url)
fh.seekToEndOfFile()
fh.write("this text will be written end of the text".data(using: .utf8)!)

```

Okuma işlemi soncunda bir Data nesnesi elde edilir. Örneğin:

```

let fh = try FileHandle(forReadingFrom: url)
let data = fh.readData(ofLength: 10)
let s = String(data: data, encoding: .utf8)!
print(s)

```

Burada biz readData metodu ile dosyadan 10 byte okuduk. Metot bize okunan kısmı Data? biçiminde verdi. Biz de data nesnesinden String sınıfının başlangıç metodunu kullanarak String elde ettik.

3) Yukarıda da belirtildiği gibi FileHandle sınıfı zaten var olan dosya üzerinde işlem yapmaktadır. Pekiyi dosyaların kendisi ilk kez nasıl yaratılır? İşte bu işlem yine FileManager sınıfıyla yapılmalıdır. Örneğin:

4) İşlem bittiğinde FileHandle sınıfının bitiş metodu dosyayı kapatır. Ancak biz bu işlemi daha önce yapmak isteyebiliriz. Bunun için FileHandle sınıfının örnek closeFile metodu kullanılmalıdır.

FileHandle sınıfı olan bir dosyayı açarak işlem yapmaktadır. Pekiyi dosyayı ilk kez nasıl yaratabiliriz? İşte dosya yaratmak için FileManager sınıfının createFile metodu kullanılmaktadır. Bu metot işlemin başarısına göre Bool bir değere geri dönmektedir. Örneğin:

```

let fm: FileManager = FileManager.default
do {
 var url = try fm.url(for: .applicationSupportDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
 url.appendPathComponent("mytext.txt")
 if fm.createFile(atPath: url.path, contents: nil, attributes: nil) {
 print("file successfullu created")
 } else {
 print("file cannot create")
 }
 print(url.path)
 //...
}
catch (let x)
{
 print(x.localizedDescription)
}

```

## Uygulama Ayarlarının Saklanması

Verilerin kalıcılığı için yukarıda normal dosyaları kullandık. IOS sistemlerinde uygulama ayarlarının saklanması için bir .plist modeli de kullanılmaktadır. IOS'ta uzantısı .plist olan dosyalar ayarları tutmak amacıyla bulundurulmaktadır. Biz de istersek bazı verileri sanki ayar bilgisiyimiş gibi bu .plist uzantılı dosyalarda tutabiliyoruz. Bu dosyaların içeriği aslında XML biçimdedir. Ancak UserDefaults isimli sınıf anahtar değer çifti biçiminde bu dosya üzerinde çalışmamıza izin vermektedir. UserDefaults sınıfının kullanımı şöyledir:

1) Programcı UserDefaults sınıfı türünden bir nesneyi kendisi yaratabilir. Ya da Sınıfın standard isimli özniteligi ile yaratılmış olan nesneyi doğrudan kullanabilir. Örneğin:

```
let defaults = UserDefaults.standard
```

2) Dosyaya bir bilgi yerleştirmek için sınıfın set metotları kullanılmaktadır. Overload edilmiş pek çok set metodu vardır. Biz bir diziyi de, bir Integer değeri de Float değeri de bu set metodlarıyla yazabiliriz. Örneğin bir String dizisini tümde yazmak isteyelim. Bunun için aşağıdaki set metodu kullanılmalıdır:

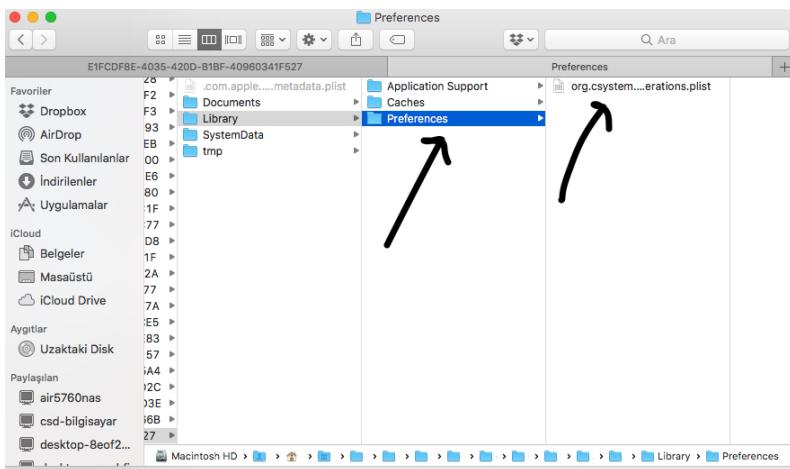
```
func set(_ value: Any?, forKey defaultName: String)
```

Örneğin:

```
let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]
```

```
let defaults = UserDefaults.standard
defaults.set(names, forKey: "myarray")
```

Peki bu .plist dosyası nerede oluşturulacaktır ve ismi ne olacaktır? İşte sistem bu dosyayı bizim için ayrılan dizinlerden (sandbox) Preferences dizini içerisinde uygulama ismi ile oluşturur. Örneğin:



3) plist dosyasına bir anahtar belirterek yazmış olduğumuz bilgi dosyadan integer, float, bool, array gibi get metodlarıyla alınır. Örneğin:

```
let defaults = UserDefaults.standard

if let names = defaults.array(forKey: "myarray") as? [String] {
 print(names)
}
else {
 print("key not found!")
}
```

Biz kendi verilerimizi uygulamamızın ayar dosyasında (.plist) dosyasında saklayabiliriz demişti. Peki her türlü bilgiyi burada saklamak uygun olur mu? Bu .plist dosyaları aslında XML dosyalarıdır. Yani bu dosyalar bir veritabanı değildir. Dolayısıyla bunlara yazma yapmak ve bunlardan okuma yapmak tümde yazma ve okuma yapmayı gerektirmektedir. Büyük verilerin bu dosyalar içerisinde tutulması hiç uygun değildir. O halde ancak küçük çaplı uygulama verileri bu ayar dosyalarında tutulmalıdır.

## PropertyList'lerin Genel Kullanımı

IOS'ta uzantısı .plist olan dosyalara "property list" dosyaları denilmektedir. Yukarıda da anlatıldığı gibi uygulama temelinde ayarların saklanması için bir tane property list dosyası oluşturulabilmektedir. Biz kendi verilerimizi de bu property list dosyasında saklayabiliriz. Ancak istersek kendimiz de başka property list dosyaları oluşturabiliriz. UserDefaults.standard biçiminde bize verilen property list nesnesi "singleton" bir nesnedir. Yani toplamda bu nesneden ve bu nesnenin temsil ettiği dosyadan bir tane vardır. Biz ne zaman UserDefaults.default desek bize hep aynı nesne verilmektedir. Böylece programcı UserDefaults.standard biçiminde property list kullandığı zaman aslında yalnızca bir tane dosya yaratılıp onu kullanıyor durumdadır.

Pekiyi biz default property list'lerde (yani uygulama ayarlarının saklanması için kullanılan .plist dosyasında) her türlü veriyi saklayabilir miyiz? Yanıt hayır. Ancak bu listelerde standart bazı türler saklanabilmektedir. Örneğin:

Pekiyi biz kendi sınıflarımızı plist dosyalarında nasıl saklarız? İşte bunun için seri hale getirme işlemi uygulanmalıdır. Bu işlem adım adım şöyle gerçekleştirilmektedir:

1) Yazma işlemleri PropertyListEncoder isimli bir sınıf yoluyla yapılır. Bu sınıfın encode isimli metodu bizden sınıf nesnesini alır (bu nesne bir container da olabilir) ondan bir Data nesnesi oluşturur. Biz de bu Data nesnesini sınıfın write metoduyla .plist dosyası biçiminde dosyaya yazarız. Ancak yazma işlemine konu olan sınıfımızın Encodable isimli protokolü destekliyor olması gereklidir. Örneğin:

```
let fm = FileManager.default
do {
 let ple = PropertyListEncoder()
 let data = try ple.encode(self.todoItems)

 var url = try fm.url(for: .cachesDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
 url.appendPathComponent("todoList.plist")
 try data.write(to: url)
}
catch (let err) {
 print(err)
}
```

2) Okuma işlemi için benzer bir yöntem kullanılır. Önce bizim dosyadaki tüm bilgileri bir Data nesnesi biçiminde okumamız gereklidir. Sonra PropertyListDecoder isimli sınıfından bir nesne yaratırız. Bu sınıfın decode isimli metodunu bizden Data nesnesini alıp ve onu decode ederek orijinal türden verecektir. Tabii decode etme işleminin düzgün yapılabilmesi için ilgili sınıfımızın Decodable protokolünü destekliyor olması gereklidir. Örneğin:

```
class ToDoItem : Encodable, Decodable {
 var contentText = ""
 var dateText = ""
```

```

var checked = false

init(_ contentText: String, _ dateText: String, _ checked: Bool)
{
 self.contentText = contentText
 self.dateText = dateText
 self.checked = checked
}
}

```

Aslında Encodable ve Decodable protokollerinin aynı anda desteklenmesi yerine Codable protokolünün desteklenmesi aynı anlamdadır.

Peki decode işlemi nerede yapılmalıdır? Tabii uygulama çalıştırışlığında hemen viewDidLoad metodunda bu işlem yapılabilir. Ancak uygulama ilk kez yaratıldığında bu dosya henüz olmayacağı için bir kontrolün de yapılması gereklidir. Örneğin:

```

override func viewDidLoad()
{
 super.viewDidLoad()

 let fm = FileManager.default
 do {
 var url = try fm.url(for: .cachesDirectory, in: .userDomainMask, appropriateFor: nil,
create: true)
 url.appendPathComponent("ToDoList.plist")

 if fm.fileExists(atPath: url.path) {
 let data = fm.contents(atPath: url.path)

 let pld = PropertyListDecoder()
 ToDoItems = try pld.decode([ToDoItem].self, from: data!)
 }
 } catch (let err) {
 print(err)
 }
}

```

## IOS'ta SQLite Kullanımı

IOS'ta SQLite Core Data özellikle kullanılabileceği gibi tamamen C API'leriyle de kullanılabilmektedir. Biz Core Data kullanımını ayrı bir başlık halinde ele alacağız. Burada SQLite'ı C API'leriyle kullanacağımız.

SQLite belli bir süreden sonra IOS içerisinde entegre edilmiştir. XCode COcoa kütüphanesi de bu C API'leri kullanabilmemiz için zaten kütüphe bulundurmaktadır. Bizim SQLite C API'lerini kullanabilmemiz için tek yapacağımız şey SQLite modülünü import etmektir. Örneğin:

```
import SQLite3
```

SQLite'ın C API'leri için aşağıdaki orijinal dokümanlara başvurabilirsiniz:

<https://www.sqlite.org/docs.html>

SQLite için ilk yapılacak şey sqlite3\_open fonksiyonu ile veritabanı dosyasını yaratmak ya da açmaktır. Bu fonksiyon eğer bir veritabanı dosyası yoksa sıfırdan onu yaratır varsa onu açar. sqlite\_open fonksiyonunun C'deki prototipi şöyledir:

```

int sqlite3_open(
 const char *filename, /* Database filename (UTF-8) */
 sqlite3 **ppDb /* OUT: SQLite db handle */
);

```

Fonskiyonun birinci parametresi veritabanı dosyasının yol ifadesini, ikinci parametresi veritabanı handle göstericisinin adresini almaktadır. Bu ikinci parametre aslında C'deki göstericiyi gösteren gösterici biçimindedir. Ancak C'deki göstericiler Swift'te OpaquePointer isimli bir yapıyla temsil edilmektedir. Dolayısıyla biz ikinci parametreye OpaquePointer? türünden bir nesnenin adresini geçirmeliyiz. sqlite3 fonksiyonları genel olarak Int32 türü ile geri dönmektedir. İşlem başarılıysa bunların geri dönüş değerleri SQLITE\_OK biçimindedir.

Örneğin:

```

var dbHandle: OpaquePointer? = nil
var result: Int32
let path = "test.sqlite"

result = sqlite3_open(path, &dbHandle);
if result != SQLITE_OK {
 print("Cannot open or create SQLite file!")
 return
}

```

Tabii bizim path olarak vereceğimiz yer uygulama dizinlerinden bir yerin içerisinde (sandbox) olmalıdır. Örneğin:

```

override func viewDidLoad()
{
 super.viewDidLoad()

 let fm = FileManager.default
 var url: URL

 do {
 url = try fm.url(for: .cachesDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
 url.appendPathComponent("test.sqlite")
 }
 catch (let err) {
 print(err)
 return
 }

 var dbHandle: OpaquePointer? = nil
 var result: Int32
 result = sqlite3_open(url.path, &dbHandle);
 if result != SQLITE_OK {
 print("Cannot open or create SQLite file!")
 return
 }
}

```

Veritabanı dosyası oluşturulduktan ya da açıldıktan sonra yukarıda örnekte görüldüğü gibi ondan bir handle değeri elde ettik. Arik bu handle değerini kullanarak C fonksiyonları ile işlemlerimizi yapacağız. SQL komutları sqlite3\_exec fonksiyonuyla işletilmektedir. sqlite3\_exec fonksiyonun C'deki prototipi şöyledir:

```

int sqlite3_exec(
 sqlite3*, /* An open database */
 const char *sql, /* SQL to be evaluated */
 int (*callback)(void*,int,char**,char**), /* Callback function */
 void *, /* 1st argument to callback */
 char **errmsg /* Error msg written here */
);

```

Fonksiyonun birinci parametresi sqlite3\_open fonksiyonundan elde edilen handle değeridir. İkinci parametre SQL cümlesini belirtmektedir. Diğer parametreler nil geçilebilir. Ancak son parametre fonksiyon başarısız olduğunda bu başarısızlığın nedeninin yerleştirileceği string nesnesini belirtmektedir. Örneğin:

```

override func viewDidLoad()
{
 super.viewDidLoad()

 let fm = FileManager.default
 var url: URL

 do {
 url = try fm.url(for: .cachesDirectory, in: .userDomainMask, appropriateFor: nil, create:
true)
 url.appendPathComponent("test.sqlite")
 print(url.path)
 }
 catch (let err) {
 print(err)
 return
 }

 var dbHandle: OpaquePointer? = nil
 var result: Int32

 if !fm.fileExists(atPath: url.path) {
 result = sqlite3_open(url.path, &dbHandle);
 if result != SQLITE_OK {
 print("Cannot open or create SQLite file!")
 return
 }

 result = sqlite3_exec(dbHandle, "CREATE TABLE student(student_id INTEGER PRIMARY KEY
AUTOINCREMENT, student_name VARCHAR(64), student_no INTEGER)", nil, nil, nil)

 if result != SQLITE_OK {
 print("cannot create table!")
 return
 }
 }

 let students = ["Özlem Tanış": 234, "Mehmet Er": 76, "Ahmet Çavus": 674,
 "Ayşe Şen": 734]

 for (key, value) in students {
 result = sqlite3_exec(dbHandle, "INSERT INTO student(student_name, student_no) VALUES
('\(key)', \(value))", nil, nil, nil)

 if result != SQLITE_OK {
 print("cannot insert record!")
 return
 }
 }
}

```

```

 }
 }
 sqlite3_close(dbHandle)
}
}

```

Pekiyi SELECT işlemi nasıl yapılmaktadır? SQL'de SELECT işlemi geriye kayıt döndürmektedir. SELECT işlemi sırasıyla şu aşamalardan geçilerek gerçekleştirilir:

1) Önce SELECT cümlesi oluşturulur ve sqlite3\_prepare\_v2 fonksiyonu sokulur. Bu fonksiyonun C'deki prototipi şöyledir:

```

int sqlite3_prepare_v2(
 sqlite3 *db, /* Database handle */
 const char *zSql, /* SQL statement, UTF-8 encoded */
 int nByte, /* Maximum length of zSql in bytes. */
 sqlite3_stmt **ppStmt, /* OUT: Statement handle */
 const char **pzTail /* OUT: Pointer to unused portion of zSql */
);

```

Fonksiyonun birinci parametresi veritabanın handle değeridir. İkinci parametresi SQL SEELCT cümlesini almaktadır. Üçüncü parametre -1 geçilmelidir. Dördüncü parametre SELECT işlemi için başka bir handle değerinin adresi alır. Programcı yine OpaquePoiner? türünden bir değişken oluşturup buraya onun adresini vermelidir. Bu SELECT handle değeri kayıtların elde edilmesinde kullanılacaktır. Son parametre yine nil geçilebilir. Fonksiyon başarı durumunda SQLITE\_OK değerine geri dönmektedir.

2) Bunadan sonra artık kayıtların tek tek ele geçirilmesi için sqlite3\_step fonksiyonu bir döngü içerisinde çağrılır. Fakat işin başında imleç (cursor) ilk kaydın 1 gerisindedir. Onu ilk kayda almak için bir step uygulamak gereklidir. sqlite3\_step fonksiyonunun C'deki prototipi şöyledir:

```
int sqlite3_step(sqlite3_stmt*);
```

Fonksiyon parametre olarak sqlite\_prepare\_v2 fonksiyonundaki SELCT handle değerini alır. İşlem başarılıysa fonksiyon SQLITE\_ROW değerine geri dönmektedir. Örneğin:

```

var selectHandle: OpaquePointer? = nil

result = sqlite3_prepare_v2(dbHandle, "SELECT student_name, student_no FROM student", -1,
&selectHandle, nil)
if result == SQLITE_OK {

 while sqlite3_step(selectHandle) != SQLITE_ROW {
 //....
 }
}

```

3) İmleç konumlandırıldıktan sonra artık ilgili kaydın sütun değerleri elde edilebilir. Bunun için sqlite3\_column\_xxx isimli fonksiyonlar kullanılır. Bu fonksiyonların birinci parametreleri SELECT HANDLE değerini, ikinci parametreleri ise select edilen sütunun indeks numarasını alır. Örneğin sütunda text bir bilgi varsa biz onun değerini sqlite3\_column\_text fonksiyonu ile, int bir bilgi varsa onun değerini sqlite3\_column\_int fonksiyonu ile alırız. Fonksiyon ilgili sütunun değerine geri dönmektedir. Örneğin:

```
var selectHandle: OpaquePointer? = nil
```

```

result = sqlite3_prepare_v2(dbHandle, "SELECT student_name, student_no FROM student", -1,
&selectHandle, nil)
if result == SQLITE_OK {

 while sqlite3_step(selectHandle) == SQLITE_ROW {
 let name = String(cString: sqlite3_column_text(selectHandle, 0)!)
 let no = sqlite3_column_int(selectHandle, 1)

 print(name, no)
 }
}

```

Burada dikkat edilmesi gereken bir nokta şudur: sqlite3\_column\_text fonksiyonu C tarzı string'i bize dönüştürerek Swift tarzı string olarak doğrudan vermemektedir. Onun yerine C Tarzı strin'in adresini UnsafePointer<UInt8>? biçiminde verir. Zaten Swift'in String sınıfının init(cString:) parametreli metodu bunu Swift string'ine dönüştürmektedir. Select işleminden sonra imleç sqlite3\_finalize fonksiyonuyla kapatılabilir. Örneğin:

```
sqlite3_finalize(selectHandle)
```

En sonunda veritabanı sqlite3\_close fonksiyonuyla kapatılmalıdır. Örneğin:

```
sqlite3_close(dbHandle)
```

O halde bütünsel örnek şöyle verilebilir:

```

override func viewDidLoad()
{
 super.viewDidLoad()

 let fm = FileManager.default
 var url: URL

 do {
 url = try fm.url(for: .cachesDirectory, in: .userDomainMask, appropriateFor: nil, create:
true)
 url.appendPathComponent("test.sqlite")
 print(url.path)
 }
 catch (let err) {
 print(err)
 return
 }

 var dbHandle: OpaquePointer? = nil
 var result: Int32

 if !fm.fileExists(atPath: url.path) {
 result = sqlite3_open(url.path, &dbHandle);
 if result != SQLITE_OK {
 print("Cannot open or create SQLite file!")
 return
 }

 result = sqlite3_exec(dbHandle, "CREATE TABLE student(student_id INTEGER PRIMARY KEY
AUTOINCREMENT, student_name VARCHAR(64), student_no INTEGER)", nil, nil, nil)
 }
}

```

```

if result != SQLITE_OK {
 print("cannot create table!")
 return
}

let students = ["Özlem Tanış": 234, "Mehmet Er": 76, "Ahmet Çavuş": 674,
 "Ayşe Şen": 734]

for (key, value) in students {
 result = sqlite3_exec(dbHandle, "INSERT INTO student(student_name, student_no) VALUES
('\(key)', \(value))", nil, nil, nil)

 if result != SQLITE_OK {
 print("cannot insert record!")
 return
 }
}
sqlite3_close(dbHandle)
}

result = sqlite3_open(url.path, &dbHandle);
if result != SQLITE_OK {
 print("Cannot open or create SQLite file!")
 return
}

var selectHandle: OpaquePointer? = nil

result = sqlite3_prepare_v2(dbHandle, "SELECT student_name, student_no FROM student", -1,
&selectHandle, nil)
if result == SQLITE_OK {

 while sqlite3_step(selectHandle) == SQLITE_ROW {
 let name = String(cString: sqlite3_column_text(selectHandle, 0)!)
 let no = sqlite3_column_int(selectHandle, 1)

 print(name, no)
 }
}
sqlite3_finalize(selectHandle)

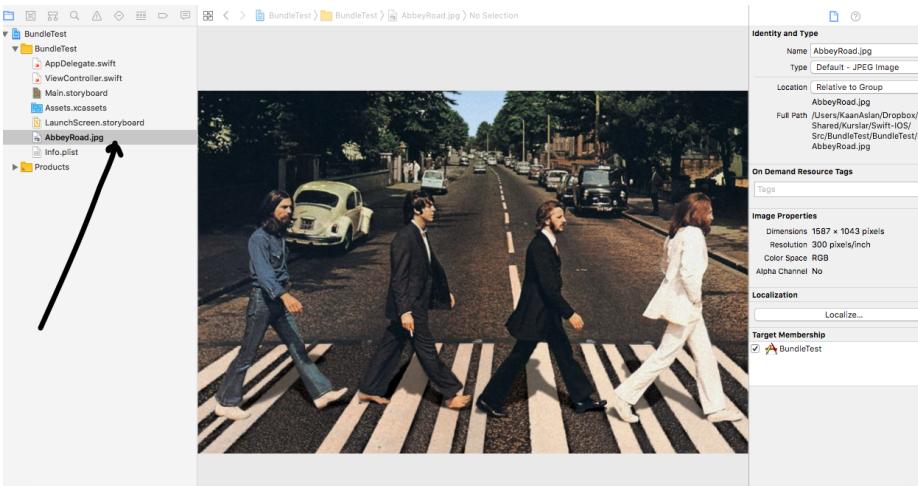
sqlite3_close(dbHandle)
}

```

## Bundle Kavramı ve Bundle Kullanımı

Biz bir IOS uygulamasını AppStore'dan indirdiğimizde uygulama kendi dosyalarıyla birlikte IOS cihazında bir dizine çekilmektedir. Yine anımsanacağı gibi uygulamanın kullanabilmesi için ismine "sandbox" denilen dizinler de oluşturulmaktadır. Uygulamanın kendisi bir "bundle"dır. Fakat programcı isterse başka bundle'lar da oluşturabilir. Bir dosyayı bir bundle'ın içerisinde yerleştirmenin avantajı bu dosyayı uygulamayla birlikte hedef aygıtta aktarmaktır. Örneğin bir SQLite veritabanı dosyası ya da resource olarak kullanılan birtakım dosyalar bundle içeriğine yerleştirilebilir. Yukarıda da belirtildiği gibi uygulamanın bir ana bundle'ı vardır. Ancak programcı isterse uygulaması için başka bundle'lar yaratabilir.

Bundle kavramı Bundle isimli sınıfı temsil edilmektedir. Bu sınıfın static main isimli property elemanı uygulamanın ana bundle'ını belirtmektedir. Örneğin biz bir uygulamamızın ana bundle'ına bir dosya ekleyelim:



Dosya Finder'dan sürükleenip proje kısmasına bırakılırsa (ya da proje üzerinde bağlam menüsüyle de bu işlem yapılabilir.) bu dosya zaten ana bundle içerisinde yerleştirilmektedir. Örneğimizde "AbbeyRoad.jpg" dosyası ana bundle içerisinde yerleştirilmiştir. Şimdi biz bu dosyanın yol ifadesini alıp normal biçimde dosyayı kullanabiliyoruz. Bu işlem şöyle yapılmaktadır:

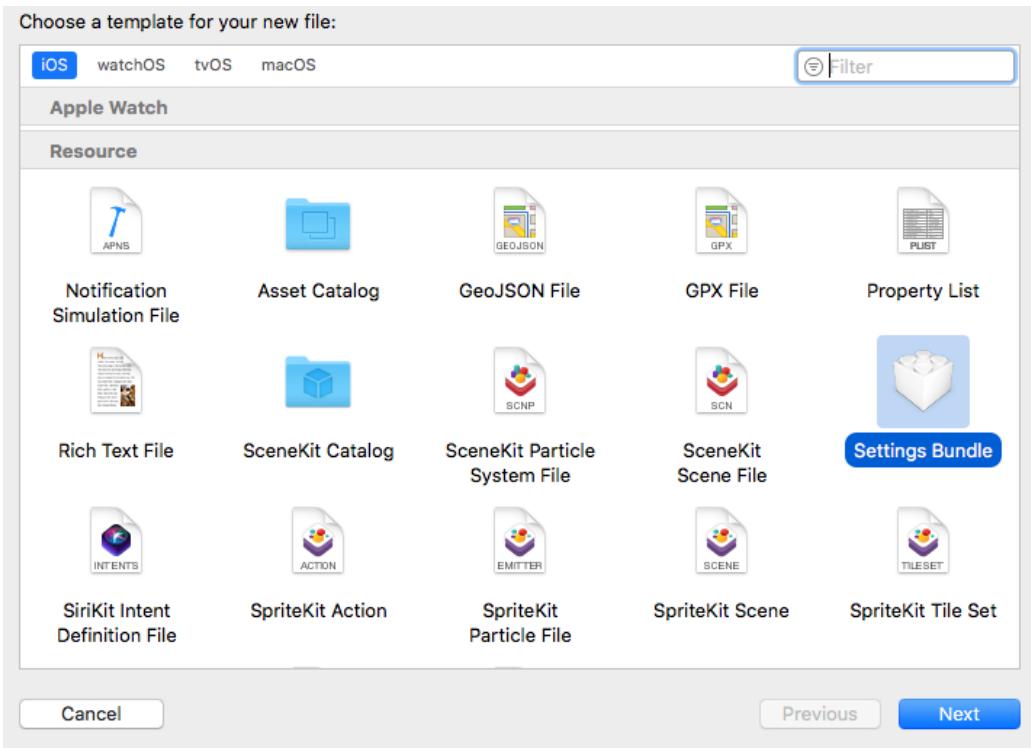
```
let url: URL? = Bundle.main.url(forResource: "AbbeyRoad", withExtension: "jpg")
```

Burada dosyanın yalnızca isminin forResource parametresinde belirtildiğine, uzantısının ayrıca withExtensions parametresiyle belirtildiğine dikkat ediniz. Şimdi biz dosyanın URL'ini elde ettiğimize göre onun yol ifadesini path property'si ile alabiliriz. Örneğin bir UIImageView içerisinde bu resmi yerlestirelim:

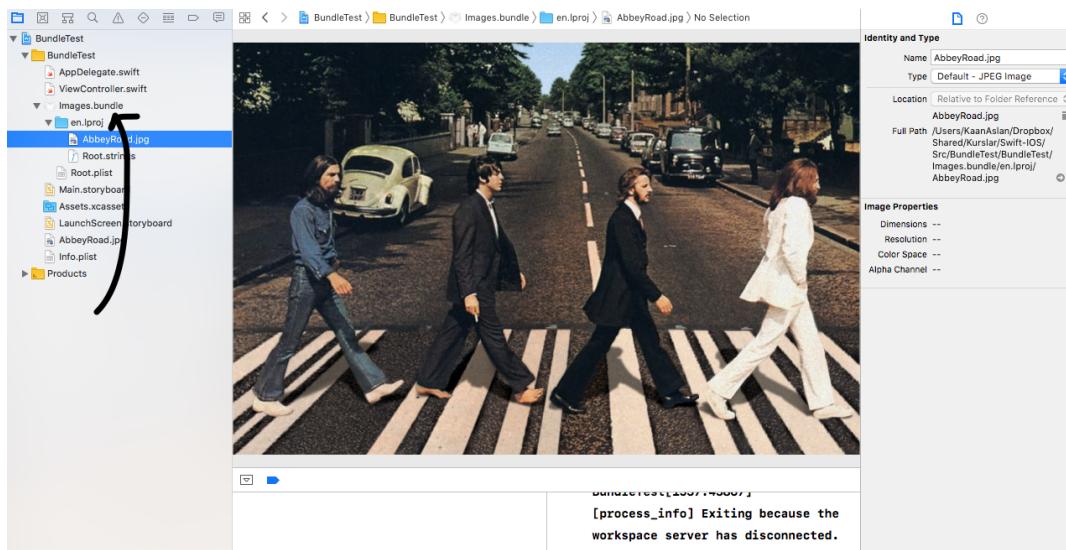
```
class ViewController: UIViewController {
 @IBOutlet weak var imageView: UIImageView!
 override func viewDidLoad() {
 super.viewDidLoad()
 let url: URL? = Bundle.main.url(forResource: "AbbeyRoad", withExtension: "jpg")
 imageView.image = UIImage(contentsOfFile: url!.path)
 }
}
```

Burada biz image nesnesindeki resmi ana bundle'dan yükledik. Tabii aslında bilindiği gibi uygulamnın Assets klasörü ile bu da pratik yapılabılır. Assests klasörü daha farklı bir organizasyondur. Bu klasör özellikle resim tarzı kaynakları depolamak için düşünülmüştür. Fakat başka dosyalar da bu Assets klasörüne yerleştirilebilmektedir.

Biz uygulama içerisinde ana bundle yerine kendi yarattığımız bundle'ı da kullanabiliyoruz. Başka bir deyişle uygulama içerisinde başka bundle'lar da yaratılabilir. Aslında bundle uygulama dizininde .bundles uzantılı bir dizin görünümündedir. XCode'da yeni bir bundle yaratmak için File/New/File seçilir. Bundan sonra diyalog penceresinden Resource/Setting Bundle seçilmelidir.



Burada Images isimli bir bundle oluşturmuş olalım:



Burada oluşturduğumuz bundle'a "resource bundle" da denilmektedir. Bu bundle bağımsız olarak değil ana bundle'in içerisinde oluşturulmuştur. Bu tür resource bundle'ların kullanımı şöyledir: Önce ana bundle nesnesi `Bundle.main` ifadesi elde edilir. Sonra bu ana bundle nesnesi ile sınıfın `path(forResource:ofType:)` metodu çağrılır. Buradan resource bundle'in yol ifadesi elde edilir. Sonra bu yol ifadesinden hareketle resource bundle için `Bundle` sınıfının `init?(path:)` metodu ile bir bundle nesnesi oluşturulur. Artık diğer işlemler benzer biçimde yapılır. Örneğin:

```
import UIKit

class ViewController: UIViewController {

 @IBOutlet weak var imageView: UIImageView!
```

```

override func viewDidLoad()
{
 super.viewDidLoad()

 let bundlePath: String? = Bundle.main.path(forResource: "Images", ofType: "bundle")
 let bundle: Bundle? = Bundle(path: bundlePath!)
 let url: URL? = bundle!.url(forResource: "AbbeyRoad", withExtension: "jpg")
 imageView.image = UIImage(contentsOfFile: url!.path)
}

}

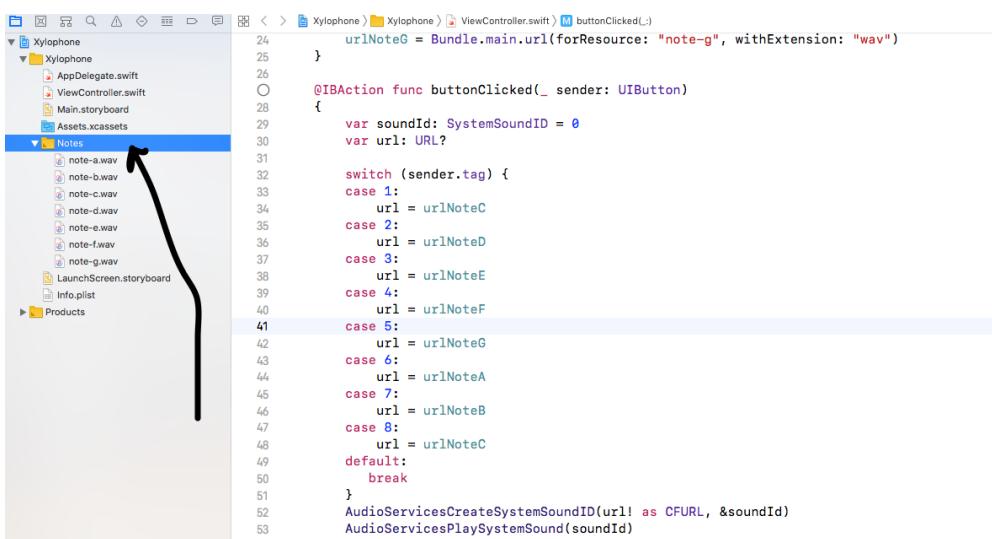
```

Aslında bundle konusu hem Mac OS X için hem de iOS için daha detaylı bir konudur. Fakat burada şimdilik konuyu sonlandıracagız.

## IOS'ta Audio İşlemleri

IOS'ta ses işlemleri farklı pek çok kütüphane kullanılabilmektedir. En yalın biçimde bir "wav" dosyasının çaldırılması "sistem ses (system sound)" sistemiyle yapılabilir. Bunun için AVFoundation modülü import edilmelidir. Bir "wav" dosyasının çaldırılması tipik olarak şöyle gerçekleştirilir:

1) Öncelikle çaldırılacak wav dosyasının URL'inin (yol ifadesinin) elde edilmesi gerekmektedir. Programcı genellikle bu dosyaları resource olarak ana bundle'ın içerisinde yerleştirmektedir. Örneğin:



Programcı genellikle bu tür resource dosyalarını proje içerisinde "New Group" oluşturarak bir dizin gibi onun içerisinde yerleştirmektedir. Önceki konuda da görüldüğü gibi bu resource dosyalarının url'leri Bundle.main.url metoduyla elde edilebilmektedir. Örneğin:

```

override func viewDidLoad()
{
 super.viewDidLoad()

 urlNoteA = Bundle.main.url(forResource: "note-a", withExtension: "wav")
 urlNoteB = Bundle.main.url(forResource: "note-b", withExtension: "wav")
 urlNoteC = Bundle.main.url(forResource: "note-c", withExtension: "wav")
 urlNoteD = Bundle.main.url(forResource: "note-d", withExtension: "wav")
 urlNoteE = Bundle.main.url(forResource: "note-e", withExtension: "wav")
 urlNoteF = Bundle.main.url(forResource: "note-f", withExtension: "wav")
 urlNoteG = Bundle.main.url(forResource: "note-g", withExtension: "wav")
}

```

}

2) Bundan sonra bir sistem sound id elde etmek gereklidir. Bu işlem AudioServicesCreateSystemSoundID fonksiyonuyla yapılmalıdır. Bu fonksiyon bizden URL yapısını değil CFURL yapısını almaktadır. Ancak URL as operatörü ile CFURL türüne dönüştürülebilmektedir.

```
var soundId: SystemSoundID = 0

AudioServicesCreateSystemSoundID(url! as CFURL, &soundId)
```

3) En sonunda çaldırma işlemi bu sound id verilerek AudioServicesPlaySystemSound fonksiyonu ile yaptırılır. Örneğin:

```
AudioServicesPlaySystemSound(soundId)
```

4) Çaldırma işleminden sonra elde edilmiş olan sound id'nin boşaltılması gereklidir. Bu boşaltma işlemi normal olarak AudioServicesDisposeSystemSoundID fonksiyonunu çağırarak yapılmalıdır. Fakat bu fonksiyonların nerede çağrılabileceği önemlidir. Biz çalma işlemini yapan AudioServicesPlaySystemSound fonksiyonundan hemen sonra bu fonksiyonları çağrılmaya çalışırsak sorun olur. Bunun nedeni çalma işleminin asenkron biçimde arka planda yapılmasıdır. Yani programın akışı AudioServicesPlaySystemSound fonksiyonundan çıktığında hala çalma işlemi devam etmektedir. Peki o zaman biz kaynakları nerede boşaltabiliriz? İşte çalma eylemi bittiğinde bizim belirlediğimiz bir fonksiyonun çağrılmasını isteyebiliriz. Bunun için AudioServicesAddSystemSoundCompletion fonksiyonu kullanılmaktadır. Programcılar genellikle bu fonksiyonda küçük bir closure girerek bu işlemi yaparlar. O halde sonlandırma işlemi tipik olarak şöyle yapılır:

```
AudioServicesCreateSystemSoundID(url! as CFURL, &soundId)
AudioServicesAddSystemSoundCompletion(soundId, nil, nil, { sndId, context in
 AudioServicesRemoveSystemSoundCompletion(sndId)
 AudioServicesDisposeSystemSoundID(sndId)
}, nil)
AudioServicesPlaySystemSound(soundId)
```

Aslında boşaltımın daha yalın bir alternatifide vardır. Burada AudioServicesAddSystemCompletion fonksiyonu yerine AudioServicesPlaySystemSoundWithCompletion fonksiyonu kullanılır. Yani kodun daha yalın alternatif şöyledir:

```
AudioServicesCreateSystemSoundID(url! as CFURL, &soundId)
AudioServicesPlaySystemSoundWithCompletion(soundId) {
 AudioServicesDisposeSystemSoundID(soundId)
 print(soundId)
}
```

Burada AudioServicesPlaySystemSoundID fonksiyonu hem çalma işlemini hem de çalma bittiğinde fonksiyon çağrırmayı tek başına yapmaktadır. O halde telefon tarzı bir uygulamasının ana kodları şöyledir:

```
import UIKit
import AVFoundation

class ViewController: UIViewController {
 var urlNoteA, urlNoteB, urlNoteC, urlNoteD, urlNoteE, urlNoteF, urlNoteG: URL?

 override func viewDidLoad()
 {
 super.viewDidLoad()

 urlNoteA = Bundle.main.url(forResource: "note-a", withExtension: "wav")
 urlNoteB = Bundle.main.url(forResource: "note-b", withExtension: "wav")
```

```

urlNoteC = Bundle.main.url(forResource: "note-c", withExtension: "wav")
urlNoteD = Bundle.main.url(forResource: "note-d", withExtension: "wav")
urlNoteE = Bundle.main.url(forResource: "note-e", withExtension: "wav")
urlNoteF = Bundle.main.url(forResource: "note-f", withExtension: "wav")
urlNoteG = Bundle.main.url(forResource: "note-g", withExtension: "wav")
}

@IBAction func buttonClicked(_ sender: UIButton)
{
 var soundId: SystemSoundID = 0
 var url: URL?

 switch (sender.tag) {
 case 1:
 url = urlNoteC
 case 2:
 url = urlNoteD
 case 3:
 url = urlNoteE
 case 4:
 url = urlNoteF
 case 5:
 url = urlNoteG
 case 6:
 url = urlNoteA
 case 7:
 url = urlNoteB
 case 8:
 url = urlNoteC
 default:
 break
 }
 AudioServicesCreateSystemSoundID(url! as CFURL, &soundId)
 AudioServicesPlaySystemSoundWithCompletion(soundId) {
 AudioServicesDisposeSystemSoundID(soundId)
 print(soundId)
 }
}
}
}

```

## MP3 ve Benzeri Dosyaların Çaldırılması

MP3, WAV ve benzeri ses dosyalarını çaldırmak için AVAudioPlayer isimli sınıf kütüphane kullanılmaktadır. Bu nedenle çaldırma işleminden önce bu sınıfın bulunduğu AVFoundation kütüphanesi import edilmelidir. Müzik çaldırma işlemi şu adımlardan geçilerek gerçekleştirilebilir:

1) Çalınacak müzik dosyasına ilişkin URL elde edilir. Örneğin dosyanın ana bundle'da resource olarak bulunduğu düşünelim:

```
let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp3")
```

2) AVAudioPlayer sınıfı türünden bir nesne oluşturulur. Ancak çalma işlemi yine asenkron yapıldığı için bu nesnenin faaliyet alanından çıktığında çöp toplayıcı tarafından yok edilmemesi gereklidir. Bunun için de en pratik yol nesne referansının sınıfın özniteliği yapılmasıdır. Nesneyi yaratırken AVAudioPlayer sınıfının init(contentsOf:) metodu kullanılabilir. Örneğin:

```
let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp3")
```

```
avPlayer = try? AVAudioPlayer(contentsOf: url!)
guard avPlayer != nil else {return}
```

3) Artık sıra müziği çaldırmaya gelmiştir. Bunun için sırasıyla önce AVAudioPlayer sınıfının prepareToPlay ve sonra da play isimli metotları çağrılır. Örneğin:

```
@IBAction func buttonPlayClickHandler(_ sender: Any)
{
 let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp3")

 avPlayer = try? AVAudioPlayer(contentsOf: url!)
 avPlayer.prepareToPlay()
 guard avPlayer != nil else {return}
 avPlayer.play();
}
```

4) AVAudioPlayer sınıfının pause metodu müziği durdurmakta kullanılır. Durdurulmuş bir müzik yeniden play metodu çağırılarak kaldığı yerden devam ettirilebilir. Örneğin biz Play ve Pause işlemini aynı zamanda yapan bir düğme ile çaldırma ve durdurma işlemlerini şöyle yapabiliriz:

```
import UIKit
import AVFoundation

class ViewController: UIViewController {

 var avPlayer: AVAudioPlayer!
 var pauseFlag: Bool = true
 @IBOutlet weak var pausePlayButton: UIButton!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp3")

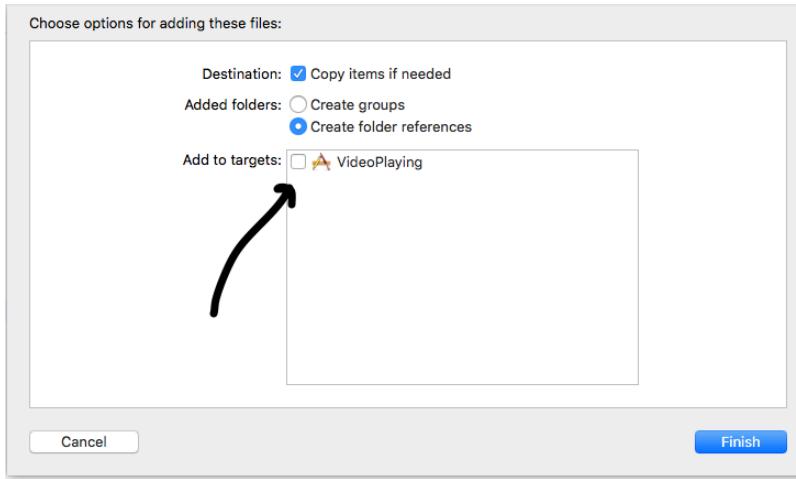
 avPlayer = try? AVAudioPlayer(contentsOf: url!)
 avPlayer.prepareToPlay()
 guard avPlayer != nil else {return}
 }

 @IBAction func puasePlayClickHandler(_ sender: Any)
 {
 if pauseFlag {
 avPlayer.play()
 pausePlayButton.setImage(UIImage(named: "Pause"), for: .normal)
 pauseFlag = false
 }
 else {
 avPlayer.pause()
 pausePlayButton.setImage(UIImage(named: "Play"), for: .normal)
 pauseFlag = true
 }
 }
}
```

## Video Görüntüleme İşlemleri

Video oynatmak aslında ses çalmak gibi basit bir işlemidir. Bunun için AVKit modülü import edilmelidir. Video oynatımı ayrı bir diyalog penceresi yoluyla ya da ana pencerenin içine gömülü biçimde gerçekleştirilebilir. Ayrı bir pencere yoluyla video oynatmak için sırasıyla şu işlemler yapılmalıdır:

1) Öncelikle görüntülenecek video'nun url'si elde edilmelidir. Video yine ana bundle içerisinde kaynak biçimde yerleştirilebilir. Yerleştirme sırasında sürükle bırak yaparken Add To Target seçenek kutusu çarpılmalıdır.



2) AVPlayerViewController sınıfı türünden bir kontrol nesnesi oluşturulur. Örneğin:

```
let avController: AVPlayerViewController = AVPlayerViewController()
```

3) Controller nesnesi oluşturulduktan sonra asıl oynatma işlemini yapacak olan AVPlayer nesnesi oluşturulmalıdır. Bu nesne oluşturulırken oynatılacak video'nun URL'si verilir. Bundan sonra AVPlayer nesnesinin referansı controller nesnesinin player isimli property'sine atanmalıdır. Örneğin:

```
let avController: AVPlayerViewController = AVPlayerViewController()
avController.player = AVPlayer(url: url!)
```

4) Artık sıra diyalog penceresinin görüntülenmesine gelmiştir. Bu işlem yine ViewController sınıfının present isimli metoduyla yapılmaktadır. Bu metodun birinci parametresine controller nesnesi verilmelidir. Örneğin:

```
import UIKit
import AVKit

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func launchButtonClicked(_ sender: Any)
 {
 let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp4")

 let avController: AVPlayerViewController = AVPlayerViewController()
 avController.player = AVPlayer(url: url!)
 self.present(avController, animated: true)
 }
}
```

}

Burada present metodunu akışı o noktada bekletmez. Akış akmaya devam eder. Yeni bir pencerede video gösterilir.



Bazen video'lar ayrı bir pencerede değil ana pencerenin içerisinde gömülü olarak da görüntülenmek istenebilir. Bu durumda yapılacaklar biraz değişmektedir. Şimdi adım adım bunu ele alalım:

1) Yine önce görüntülenecek olan video'nun url'si elde edilir. Örneğin:

```
let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp4")
```

2) Yine AVPlayerViewController sınıfı türünden bir nesne oluşturulur ve bu nesnenin player property'sine AVPlayer nesnesi atanır. Örneğin:

```
let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp4")
let avController: AVPlayerViewController = AVPlayerViewController()
avController.player = AVPlayer(url: url!)
```

3) Controller nesnesinin view property'si yoluyla controller'in view elemanına erişilir ve bu view penceresi frame property'si ile konumlandırılır. Örneğin:

```
let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp4")
let avController: AVPlayerViewController = AVPlayerViewController()
avController.player = AVPlayer(url: url!)
avController.view.frame = CGRect(x: 10, y: 400, width: 300, height: 200)
```

4) Bundan sonra yarattığımız controller ana controller'in alt controller'i yapılmalıdır. Bu işlem ana controller nesnesinin addChild metodunu ile yapılır. Daha sonra da controller'in içeriisndeki view nesnesi addSubView metodunu ile ana view içerisine eklenmelidir. Örneğin:

```
import UIKit
```

```

import AVKit

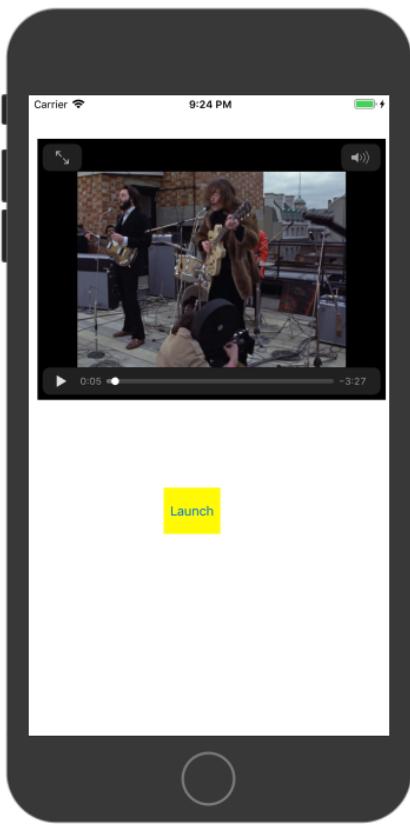
class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func launchButtonClicked(_ sender: Any)
 {
 let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp4")
 let avController: AVPlayerViewController = AVPlayerViewController()
 avController.player = AVPlayer(url: url!)
 avController.view.frame = CGRect(x: 10, y: 10, width: 400, height: 300)
 self.addChild(avController)
 self.view.addSubview(avController.view)
 }
}

```



Video oynatıcısının view penceresi örneğimizde avController.view ifadesi ile elde edilebilir. Programcı isterse bu view penceresi üzerinde değişiklikler yapabilir. Örneğin:

```

@IBAction func launchButtonClicked(_ sender: Any)
{
 let url = Bundle.main.url(forResource: "DontLetMeDown", withExtension:"mp4")
 let avController: AVPlayerViewController = AVPlayerViewController()
 avController.player = AVPlayer(url: url!)
 avController.view.frame = CGRect(x: 10, y: 50, width: 400, height: 300)
 self.addChild(avController)
}

```

```

self.view.addSubview(avController.view)
avController.view.backgroundColor = .blue

let labelName = UILabel(frame: CGRect(x: 150, y: 0, width: 200, height: 50))
labelName.textColor = .white
labelName.text = "Don't Let Me Down"
avController.view.addSubview(labelName)
}

```

Birden fazla klibin peşi sıra oynatılması istenebilir. Bunun için controller'in player property'sine AVPlayer nesnesi değil AVPlayer sınıfından türetilmiş olan AVQueuePlayer nesnesi atanmalıdır. AVQueuePlayer sınıfının init(items:) metodu bizden [AVPlayerItem] nesnesi istemektedir. İşte burada biz oynatacağımız video'ları bir dizi biçiminde vermeliyiz. AVPlayerItem nesnesi de int(url:) metodu ile oynatılacak video'nun URL'sini bizden alır. Örneğin:

```

@IBAction func launchButtonClicked(_ sender: Any)
{
 let url1 = Bundle.main.url(forResource: "DontLetMeDown", withExtension: "mp4")
 let url2 = Bundle.main.url(forResource: "TheShowMustGoOn", withExtension: "mp4")
 let url3 = Bundle.main.url(forResource: "TheHouseOfTheRisingSun", withExtension: "mp4")
 let items = [AVPlayerItem(url: url1!), AVPlayerItem(url: url2!), AVPlayerItem(url: url3!)]

 let avController: AVPlayerViewController = AVPlayerViewController()
 avController.player = AVQueuePlayer(items: items)
 avController.view.frame = CGRect(x: 10, y: 50, width: 400, height: 300)
 self.addChild(avController)
 self.view.addSubview(avController.view)
 avController.view.backgroundColor = .blue
}

```

AVPlayer sınıfının pause metodu o anda oynatılmakta olan videoyu durdurmak için kullanılır. Yeniden çalma işleminin deavm ettirilmesi yine play metoduyla yapılmaktadır. AVPlayer sınıfının rate isimli property elemanı videonun hızını ayarlamak için kullanılır. Default hız 1'dir. Biz bu 1 değerini azaltırsak videoyu yavaşlar artırırsak hızlanır. Videonun şu andaki zamansal durumu (yani ne kadarının oynatılmış olduğu) currentTime metoduyla elde edilebilir. Videonun herhangi bir zamansal noktasına seek metotlarıyla gidilebilmektedir. Bu metotlar CMTime isimli yapıyla çalışmaktadır. Örneğin:

```

@IBAction func okButtonClicked(_ sender: Any)
{
 print(avController!.player!.currentTime().seconds)
}

@IBAction func seekButtonClicked(_ sender: Any)
{
 avController!.player!.seek(to: CMTime(seconds: 50, preferredTimescale: 1))
}

```

Burada okButtonClicked metodu o anda videonun kaçinci saniyesinde olduğumuzu yazdırmaktadır. seekButtonClicked metodu ise videoyu 50'inci saniyey konumlandırmaktadır.

AVPlayer sınıfının isMuted property'si videonun sesini kapatıp açmakta kullanılır. Örneğin:

```

@IBAction func okButtonClicked(_ sender: Any)
{
 avController!.player!.isMuted = !avController!.player!.isMuted
}

```

Benzer biçimde sınıfın volume property'si de videonun sesini kısır açmak için kullanılmaktadır. Bu property 0 ile 1 arasında değer almaktadır.

AVPlayer sınıfının diğer elemanları dokümanlardan incelenebilir.

## IOS'ta Çizim İşlemleri

Bilindiği gibi UIView sınıfı temel bir pencereyi temsil etmektedir. Bu sınıfın türetilmiş görsel öğeleri temsil eden pek çok sınıf vardır. IOS'ta programcı isterse bir UIView nesnesinin içerisine kendi çizimlerini yapabilir. Aslında UIButton gibi görsel öğeleri temsil eden pencereler de bu öğeleri çizimlerle oluşturmuş durumdadır. Bir UIView penceresinin içi baştan tamamen boş durumdadır.

IOS'ta çizimler temelde iki yöntemle yapılmaktadır:

- 1) Programcı doğrudan UIView penceresinin içerisine de çizim yapabilir.
- 2) Programcı bir UIImage nesnesi yaratıp çizimleri bu nesnenin içerisine çizer. Sonra da bu çizimi bir UI nesnesinde (örneğin tipik olarak UIImageView) gösterir.

Duruma göre programcılar yukarıdaki seçeneklerden birini tercih etmektedir.

Çizim yapabilmek için bizim bir grafik bağlamına (graphics context) sahip olmamız gereklidir. Grafik bağlamı CGContext isimli bir sınıfta temsil edilmektedir. Grafik bağlamı (graphics context) elde etmenin çeşitli yolları vardır. Bazı metodlar bize grafik bağlamını kendileri elde edip vermektektir. Bazılarında ise grafik bağlamını yaratmak programıcının sorumluluğundadır.

### UIView Penceresinin İçerisine Doğrudan Çizim Yapmak

UIView penceresinin içerisine çizim yapabilmek için en çok kullanılan yöntem UIView sınıfından gelen draw isimli metodu override etmektir. draw metodu şöyle bildirilmiştir:

```
func draw(_ rect: CGRect)
```

Tabii override etme işlemi için bizin ilgili UIView sınıfından sınıf türetmemiz gereklidir. Yani örneğimiz biz düğmenin üzerine çizim yapacaksak UIButton sınıfından, boş bir pencerenin üzerine çizim yapacaksak UIView sınıfından türetme yapmamız gereklidir. draw metodunda grafik bağlamını elde edebilmek için UIGraphicsGetCurrentContext isimli fonksiyon kullanılır. Bu fonksiyon aslında zaten yaratılmış olan grafik bağlamını bize vermektedir. Örneğin:

```
import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gcontext: CGContext = UIGraphicsGetCurrentContext()!
 //..
 }
}
```

UIGraphicsGetCurrentContext metodunun geri dönüş değerinin CGContext? türündedir. İşte CGContext sınıfının çizim metodlarıyla gerçek çizimler yapılmaktadır.

Çizim işlemi biriktirmeli biçimde yapılır. Örneğin tipik olarak bir Path oluşturulur. Bu path'e eklemeler yapılır ve en

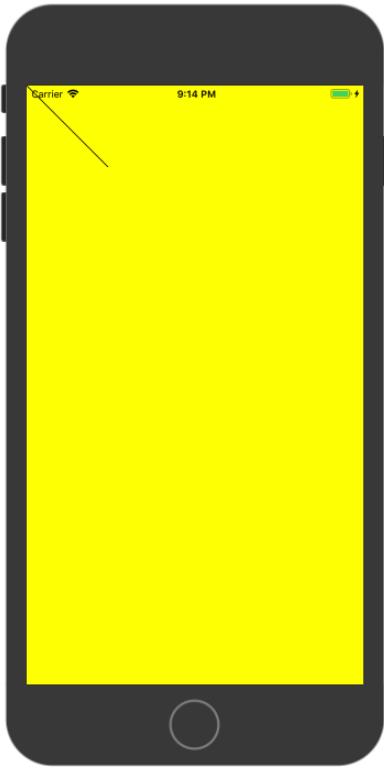
sonunda da bu path çizilir. Örneğin:

```
import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gcontext: CGContext = UIGraphicsGetCurrentContext()!

 gcontext.move(to: CGPoint(x: 0, y: 0))
 gcontext.addLine(to: CGPoint(x: 100, y: 100))
 gcontext.strokePath()
 }
}
```

Bu biçimdeki çizimlerde orijin noktası view'nun sol üst köşesidir.



Cizimin başında zaten bir path nesnesi yaratılmış durumdadır. Biz de move(to:) ve addLine(to:) metodlarıyla bir doğru çizmiş olduk. Pekiyi çizimler hangi renklerle yapılmaktadır. Çizimlerin çizgileri default olarak siyah renkle çizdirilir. Ancak default durumda iç boyalamaları yapılmamaktadır. İşte CGContext sınıfının setStrokeColor ve setFillColor metodları çizgi ve iç boyama renklerini belirlemekte kullanılmaktadır. Ancak bu metodları renkler CGColor isimli yapı türündendir. Neyse ki UIColor renklerinin CGColor biçimine dönüştürülmesi kolaydır. Örneğin:

```
import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gcontext: CGContext = UIGraphicsGetCurrentContext()!
```

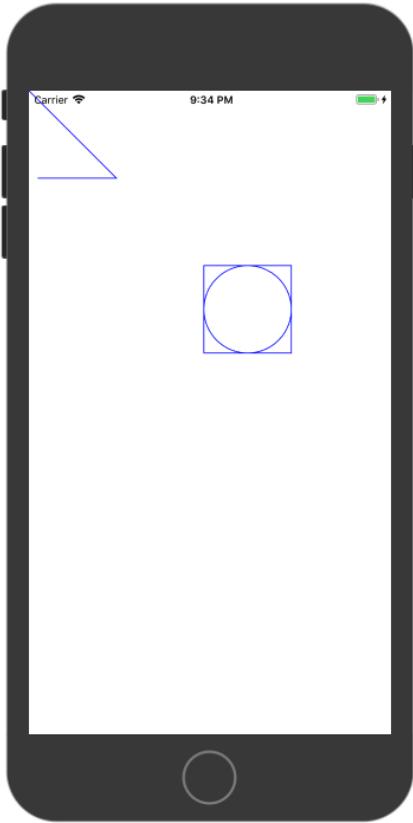
```

 gcontext.setStrokeColor(UIColor.blue.cgColor)

 gcontext.move(to: CGPoint(x: 0, y: 0))
 gcontext.addLine(to: CGPoint(x: 100, y: 100))
 gcontext.addLine(to: CGPoint(x: 10, y: 100))
 gcontext.addRect(CGRect(x: 200, y: 200, width: 100, height: 100))
 gcontext.addEllipse(in: CGRect(x: 200, y: 200, width: 100, height: 100))

 gcontext.strokePath()
 }
}

```



Grafik bağlamıyla çizimin iki temel yolu vardır. Birincisi önce bir path oluşturup bu path'e eklemeler yapıp daha sonra strokePath ya da fillPath metodlarını kullanmaktadır. Diğer yöntemde hiç path oluşturmadan doğrudan strokeXXX ve fillXXX metodlarıyla çizimler ve boyamalar yapılır. Örneğin:

```

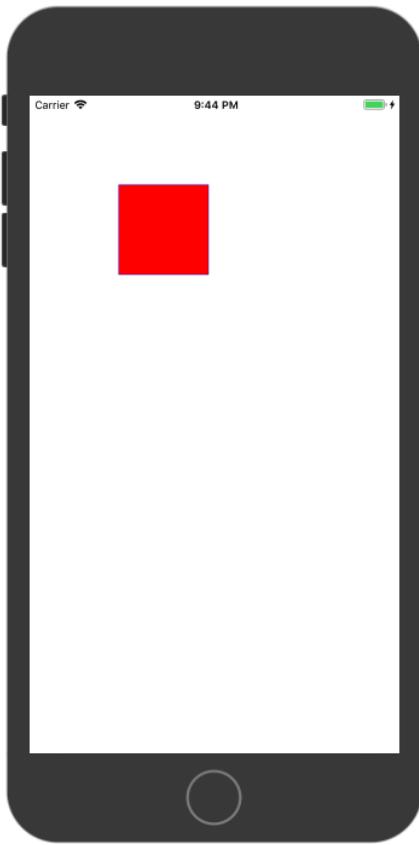
import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gcontext: CGContext = UIGraphicsGetCurrentContext()!

 gcontext.setStrokeColor(UIColor.blue.cgColor)
 gcontext.setFillColor(UIColor.red.cgColor)

 gcontext.stroke(CGRect(x: 100, y: 100, width: 100, height: 100))
 gcontext.fill(CGRect(x: 100, y: 100, width: 100, height: 100))
 }
}

```



## UIImageView Yoluyla Çizim Yapmak

Cizim yapmanın diğer bir yolu bir UIImageView nesnesi oluşturup çizimi bunun içerisinde yapmak sonra da bu UIImageView nesnesini ekranda göstermek biçimindedir. Örneğin ekranda gösterme işlemi için UIImageView sınıfını kullanabiliriz. Aslında pek çok görsel View sınıflarının image isimli bir property'leri vardır.

UIImageView yoluyla çizim yapmanın birkaç yolu olabilir. En çok kullanılan yöntem önce bir renderer nesnesi oluşturmak sonra da bu renderer nesnesi yoluyla UIImageView nesnesini elde etmektir. Bu yöntem sırasıyla şu aşamalardan geçilerek gerçekleştirilir:

1) Önce UIGraphicsImageRenderer sınıfı türünden bir renderer nesnesi yaratılır. Bu nesneya yaratılırken sınıfın init(size:) metodu kullanılmaktadır. Programcı burada oluşturacağı image'in dikdörtgensel boyutunu da vermektedir. Örneğin:

```
let renderer = UIGraphicsImageRenderer(size: CGSize(width: 375, height: 647))
```

2) UIGraphicsImageRenderere sınıfının image isimli metodu bizden bir fonksiyonu parametre olarak alır. İşte çizimler bu metotta yapılmalıdır. Programcı genellikle image metodunda fonksiyon olarak bir closure vermektedir. İşte image metodunun geri dönüş değeri UIImageView türündendir. image fonksiyonun parametrik yapısı şöyledir:

```
func image(actions: (UIGraphicsImageRendererContext) -> Void) -> UIImage
```

Göründüğü gibi fonksiyon bizden UIGraphicsImageRendererContext parametresine sahip geri dönüş değeri olmayan bir fonksiyon istemektedir. Biz bu fonksiyonu closure olarak girebiliriz UIGraphicsImageRenderereContext sınıfının cgContext elemanı CGContext sınıfı türündendir. İşte biz parametre olarak geçirilen UIGraphicsImageRendererContext nesnesinin cgContext özniteligidenden CGContext alıp çizimi bu grafik bağlam nesnesi ile yapabiliyoruz. Bu durumda çizimi

oluşturacağımız image'in içeriye yapmış oluruz. Örneğin:

```
class ViewController: UIViewController {
 @IBOutlet weak var imageView: UIImageView!
 override func viewDidLoad()
 {
 super.viewDidLoad()

 let renderer = UIGraphicsImageRenderer(size: CGSize(width: 375, height: 647))

 let image = renderer.image {
 context in
 context.cgContext.addEllipse(in: CGRect(x: 50, y: 50, width: 100, height: 100))
 context.cgContext.move(to: CGPoint(x: 100, y: 100))
 context.cgContext.addLine(to: CGPoint(x: 150, y: 100))
 context.cgContext.strokePath()
 }

 imageView.image = image
 }
}
```

Bir kez daha anımsatırsak aslında pek çok standart view nesnesinin arka planında bir iamge vardır. Bu image de bu sınıfların setImage metotlarıyla set edilmektedir. Yani biz istersek bu yöntemle bir düğmenin üzerine de çizim yapabiliriz.

### CGContext Sınıfının Önemli Metotları

Aslında en karmaşık çizimler bile birkaç temel çizimle oluşturulabilmektedir. Temel çizimler doğru, elips, yay çizimleridir. CGContext sınıfı çizim için aktif noktayı bir yol (path) kavramı içeriinde tutmaktadır. Aktif nokta yola her ekleme yapıldıktan sonra güncellenir. İstenirse sınıfın move(to:) metoduyla başka bir noktaya çekilebilir. Üç uca doğru çizimleri yapılırken bir tane move(to:) sonra addLine metotları çağrılarak çizim oluşturulabilmektedir. Örneğin:

```
import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gc: CGContext = UIGraphicsGetCurrentContext()!

 gc.move(to: CGPoint(x: 50, y: 50))
 gc.addLine(to: CGPoint(x: 50, y: 100))
 gc.addLine(to: CGPoint(x: 100, y: 100))
 gc.addLine(to: CGPoint(x: 150, y: 200))
 gc.strokePath()
 }
}
```

Tabii bağımsız doğrular için her defasında move(to:) ve addLine(to:) metotlarını yeniden çağırmak gereklidir. Örneğin:

```
import UIKit

class MyView: UIView
{
```

```

override func draw(_ rect: CGRect)
{
 let gc: CGContext = UIGraphicsGetCurrentContext()!

 for i in 0..<10 {
 gc.move(to: CGPoint(x: 50, y: 50 + i * 10))
 gc.addLine(to: CGPoint(x: 150, y: 50 + i * 10))
 gc.strokePath()
 }

}

```

`addLines(between:)` metodu bizden bir `CGPoint` dizisi alarak onların hepsini tek hamlede çizmektedir. `addEllipse(in:)` metodu ellipse çizmek için kullanılır. Ellipsin son noktası orta sağ köşedir. `addRect` ve `addRects` metotları da tek bir dikdörtgeni ve bir grup dikdörtgeni çizmek için kullanılmaktadır. Benze biçimde `addArc` metodu da yay çizmek için kullanılır.

Çizimler bir yol yani `CGPath` nesnesi yoluyla yapılmaktadır. Aslında `addXXX` metotları bu yola eleman ekler. Gerçek çizimi yapmaz. Gerçek çizimler `strokePath` ya da `fillPath` metotları çağrıldığında yapılmaktadır.

Çizim yapılırken çizgi kalınlıkları `CGContext` sınıfının `setLineWidth(_:)` metoduyla değiştirilmektedir. Bir değiştirilene kadar en son set edilen kalınlık geçerlidir. Örneğin:

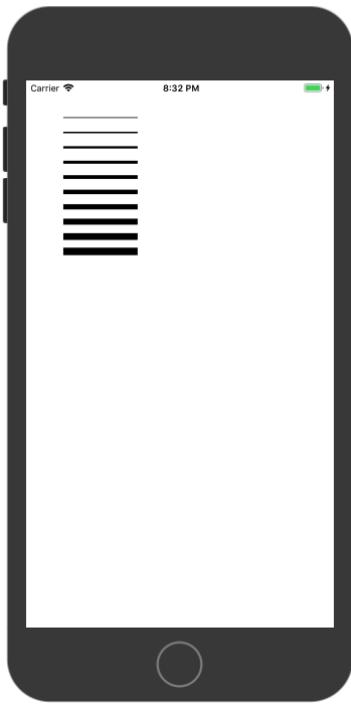
```

import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gc: CGContext = UIGraphicsGetCurrentContext()!
 var width: CGFloat

 for i in 0..<10 {
 width = CGFloat(i + 1)
 gc.setLineWidth(CGFloat(width))
 gc.move(to: CGPoint(x: 50, y: 50 + i * 20))
 gc.addLine(to: CGPoint(x: 150, y: 50 + i * 20))
 gc.strokePath()
 }
 }
}

```



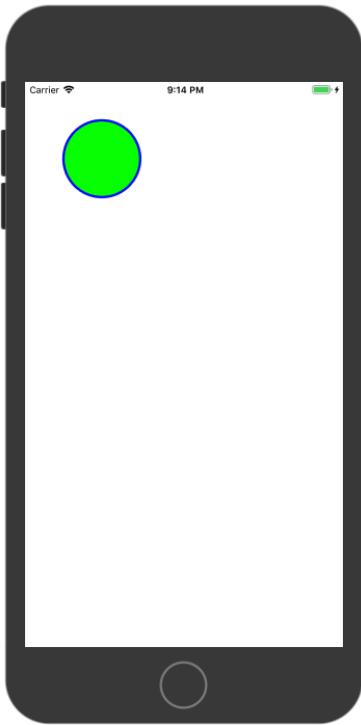
Çizgi rengini değiştirmek için setStrokeColor metodu kullanılmaktadır. Bu metot CGColor parametresi almaktadır. Şekillerin içini boyamak için şeklärerin kapalı olması gereklidir. fillPath metodu bu işlemi yapar. Ancak strokePath ve fillPath o andaki geçerli yolu reset etmektedir. Yani biz bir yol için önce strokePath sonra fillPath yapamayız. Ancak drawPath(using:) metodu ikisini reset etmeden birlikte yapmaktadır. closePath kalemin son yerdeki ucunu ilk noktaya birleştirerek şekil kapalı değilse kapatır. fillPath bu işlemi kendi içerisinde yapmaktadır. Doldurma rengi setFillColor metodune set edilmektedir. Örneğin:

```
import UIKit

class MyView: UIView
{
 override func draw(_ rect: CGRect)
 {
 let gc: CGContext = UIGraphicsGetCurrentContext()!

 gc.setLineWidth(3)
 gc.setFillColor(UIColor.green.cgColor)
 gc.setStrokeColor(UIColor.blue.cgColor)

 gc.addEllipse(in: CGRect(x: 50, y: 50, width: 100, height: 100))
 gc.drawPath(using: .fillStroke)
 }
}
```



Karmaşık çizimleri yaparken bağlam parametrelerini sürekli değiştirmek bazen yorucu olabilmektedir. Bağlam parametreleri istenirse bri stack sistemi içerisinde saklanarak geri alınabilir. Bunun için CGContext sınıfının saveGState() ve restoreGState metotları kullanılmaktadır.

## IOS'ta Thread İşlemleri

IOS'ta aslında çeşitli sınıfların metotları kendi içerisinde thread kullanmaktadır. Belli bir IOS versiyonuna kadar programcinin kendisi thread kullanamıyordu. Fakat sonraları thread kullanımı IOS sistemlerine sokuldu.

IOS'ta tüm grafik işlemler ana thread tarafından yapılmaktadır. Başka bir thread'in GUI işlemlerine müdahale etmesi programın çökmesine yol açılabilmektedir. Programcının oluşturacağı arka plan thread'ler genellikle GUI işlemi dışında birtakım yüklü işlemlerin yapılması amacıyla oluşturulmaktadır. IOS'taki GUI alt sistemi çalışma bakımından diğer işletim sistemlerindeki GUI alt sistemlerine benzemektedir. Yani IOS'ta da arka planda bir mesaj kuyruğu ve mesaj döngüsü işletilmektedir. Dolayısıyla kullanıcı bir düğmeye tıkladığında eğer orada çok uzun süren işlemler yapılrsa bu programda donma etkisine yol açar. İşte bu tür durumlarda bu yoğun işlemlerin diğer platformlarda olduğu gibi thread'lere devredilmesi gerekmektedir.

IOS'ta üç değişik biçimde thread işlemleri yapılmaktadır. Aslında bu üç biçim birbirlerini üzerine oturtulmuş yani biri diğerini kullanan sınıflarla gerçekleştirilmektedir. Burada bu yöntemleri tek tek ele alacağız.

### 1) Grand Central Dispatch (GCD) Yöntemi

Bu yöntem DispatchQueue isimli bir sınıf ile kullanılır. Genel kullanım şöyledir:

1) DispatchQueue sınıfı türünden bir nesne sınıfın init(label:) metodu ile yaratılır. Örneğin:

```
@IBAction func onButtonClicked(_ sender: Any)
{
 let dq = DispatchQueue(label: "MyThreadQueue")
 //...
}
```

2) Yaratılan bu kuyruk sistemine fonksiyonlar eklenir. Bu sınıf da eklenen fonksiyonları arka planda thread açarak çalıştırmaktadır. Fonksiyon eklemek için sınıfı birkaç metot bulunmaktadır. En yaygın kullanılımı `async` metodudur. Metodun parametrik yapısı şöyledir:

```
public func async(group: DispatchGroup? = default, qos: DispatchQoS = default, flags: DispatchWorkItemFlags = default, execute work: () -> Void)
```

Göründüğü gibi metodun ilk üç parametresi `default` değer almaktadır. O halde biz yalnızca son parametre için fonksiyon girebiliriz. Bu fonksiyonu da eğer closure biçiminde girersek etiket kullanmak zorunda kalmayız. Örneğin:

```
@IBAction func onButtonClicked(_ sender: Any)
{
 let dq = DispatchQueue(label: "MyThreadQueue")

 dq.async {
 for i in 1..<10 {
 print(i)
 sleep(1)
 }
 }

 print("ok")
 //...
}
```

Burada `async` metodu ile biz yaratılacak thread akışının çalıştırılacağı kodu vermiş olduk. Bu sınıf bu biçimde kuyruktaki diğer çalıştırılacak kod varsa onu hemen thread açıp çalıştırmaktadır. Örneğimizde "ok" yazısı bu koddan sonra ekrana basılmayacaktır. Buradaki `async` kodu arka planda thread tarafından çalıştırılacak dolayısıyla ana thread'in akışı hemen `print("ok")` çağrılarını görüp metodу terk edecektir. Böylece GUI'de herhangi bir kilitlenme olmayacağından emin olabiliriz. Peki yine kuyruğa birden fazla kod yerleştirsek ne olacaktır? Örneğin:

```
import UIKit

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 let dq = DispatchQueue(label: "MyThreadQueue")

 dq.async(execute: self.foo)
 dq.async(execute: self.bar)

 print("ok")
 //...
 }

 func foo()
 {
 for _ in 1..<10 {
 print("foo")
 }
 }
}
```

```

 sleep(1)
 }

func bar()
{
 for _ in 1..<10 {
 print("bar")
 sleep(1)
 }
}
}

```

Default durumda sınıf tek bir thread yaratarak kuyruktaki kodları sırasıyla çalıştırmaktadır. Dolayısıyla bu örnekte ekrana önce foo yazıları sonra bar ayıları basılacaktır. Tabii yine ana thread burada beklemeyecektir. Bu kodlardaki DispatchQueue nesnesinin yerel olmasının bir sakıncası yoktur. Çünkü kuyruk nesnesi son kod kuyruktan alınan kadar yaşamaya devam etmektedir. Eğer kuyruktaki kodların (fonksiyonların) tamamen farklı threadler tarafından çalıştırılması isteniyorsa DispatchQueue nesnesi yaratılırken init metodunun attributes etiketine ilişkin parametresine .concurrent değerinin geçilmesi gereklidir. Artık async metotları ile kuyruğa yerleştirilen kodlar farklı thread'lerde aynı anda asenkron biçimde çalıştırılacaktır. Örneğin:

```

import UIKit

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 let dq = DispatchQueue(label: "MyThreadQueue", attributes: .concurrent)

 dq.async(execute: self.foo)
 dq.async(execute: self.bar)

 print("ok")
 //...
 }

 func foo()
 {
 for _ in 1..<10 {
 print("foo")
 sleep(1)
 }
 }

 func bar()
 {
 for _ in 1..<10 {
 print("bar")
 sleep(1)
 }
 }
}

```

Burada artık foo ve bar fonksiyonlarının kodları birlikte asenkron olarak çalıştırılacaktır.

Kuyruğa kod eklemek için DispatchQueue sınıfının async metodlarının yanı sıra sync metodları da vardır. sync metodları senkron bir biçimde kodu çalıştırırlar. Burada senkron demek çalıştırılan kod bitmeden akışın sync metodunda bekletilmesi demektir. Halbuki async çağrısı yapıldığında akış hemen async metodundan çıkmaktadır. Örneğin:

```
import UIKit

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 let dq = DispatchQueue(label: "MyThreadQueue", attributes: .concurrent)

 dq.sync(execute: self.foo)
 print("one")
 dq.sync(execute: self.bar)
 print("two")
 //...
 }

 func foo()
 {
 for _ in 1..<10 {
 print("foo")
 sleep(1)
 }
 }

 func bar()
 {
 for _ in 1..<10 {
 print("bar")
 sleep(1)
 }
 }
}
```

Burada çağrı async ile değil sync ile yapılmıştır. Dolayısıyla normal fonksiyon çağrı gibi akış sync fonksiyonunda bekletecektir. İlk bakışta sync fonksiyonu gereksizmiş gibi gelebilir. Ancak thread'lerarası işlemlerde senkronizasyon için bu çağrı tercih edilebilmektedir. sync çağrısı ayrı bir thread'in yaratılmasına yol açabilmektedir. Ancak çağrıının yapıldığı thread ayrıca bekletilir. Bunun tek thread'lı bir akıştan bazı farklılıklarını söz konusu olabilmektedir.

Daha önceden de belirtildiği gibi IOS'ta GUI işlemleri (yani mesaj döngüsü) ana thread (main thread tarafından) yapılmaktadır. Bizim hiçbir biçimde başka bir thread'ten GUI elemanlara erişip bir GUI işlemi yapmamamız gereklidir. Aksi takdirde program kilitlenebilmektedir. Bu kilitlenmenin en önemli nedeni ana mesaj döngüsünün ana thread tarafından ele alınması ve başka bir thread'ten GUI işlemi yapıldığında bir çatışmanın (race condition) oluşmasıdır. Aslında diğer framework'lerde de bu durum böyledir. Yani diğer framework'lerde de GUI işlemlerinin başka thread'ler tarafından yapılması sorunlara yol açabilmektedir. Tabii bazen gerçekten başka bir thread kodunun GUI nesneleri üzerinde

güncelleme yapması istenebilir. Örneğin bir thread arka planda birtakım değerler elde edip bu değerleri bir UILabel nesnesi üzerine yazdırmak isteyebilir. IOS'ta bu işlem biraz ileride ele alınacağı gibi aslında ana thread'e yaptırılır. Aşağıdaki kodda düğmeye basıldığında thread içerisinde UILabel nesnesi güncellenmek istenmiştir. Bu da programın kilitlenmesine yol açacaktır:

```
import UIKit

class ViewController: UIViewController {

 @IBOutlet weak var label: UILabel!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 let dq = DispatchQueue(label: "MyThreadQueue", attributes: .concurrent)

 dq.async(execute: self.foo)
 print("Ok")
 //...
 }

 func foo()
 {
 for i in 0..<10 {
 label.text = String(i)
 sleep(1)
 }
 }
}
```

Aslında DispatchQueue nesnesine biz doğrudan kod eklemek yerine DispatchWorkItem türünden nesneler de ekleyebiliriz. Bir DispatchWorkItem nesnesi yaratılırken bizden bir kod bloğu istenmektedir. Örneğin:

```
import UIKit

class ViewController: UIViewController {

 var dwi: DispatchWorkItem?

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }

 @IBAction func buttonOkClicked(_ sender: Any)
 {
 dwi = DispatchWorkItem {
 for i in 0..<10 {
 print(i)
 sleep(1)
 }
 }
 }
}
```

```

 DispatchQueue.global().async(execute: dwi!)
 }
}

```

Kodun DispatchWorkItem sınıfıyla temsil edilmesinin bir avantajı akış üzerinde bazı işlemlerin yapılabilmesidir. Örneğin DispatchWorkItem nesnesi ile sınıfın cancel metodu çağrıldığında bu nesne üzerinde isCancelled property'sinin true ile geri döönmesine yol açar. Bu da thread akışının sonlandırılabilmesi için olanak sağlamaktadır. Örneğin:

```

import UIKit

class ViewController: UIViewController {

 var dwi: DispatchWorkItem?

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @IBAction func buttonOkClicked(_ sender: Any)
 {
 dwi = DispatchWorkItem {
 for i in 0..<10 {
 print(i)
 sleep(1)
 if self.dwi!.isCancelled {
 break
 }
 }
 }
 DispatchQueue.global().async(execute: dwi!)
 }

 @IBAction func buttonCancelClicked(_ sender: Any)
 {
 dwi!.cancel()
 }
}

```

DispatchWorkItem sınıfının notify ve perform metodları da vardır. Bu metodlar hakkında bilgi IOS dokümanlarından elde edilebilir.

DispatchQueue sınıfının main isimli bir static property elemanı vardır. Bu property bize singleton olarak yaratılmış bir DispatchQueue nesnesini vermektedir. DispatchQueue.main ile elde ettiğimiz DispatchQueue nesnesi ana thread'e ilişkin DispatchQueue nesnesidir. Yani biz bu nesneyi kullanarak senkron ya da asenkron kod çalıştıracak olsak bu kodlar ana aslında thread tarafından çalıştırılırlar. O halde biz GUI üzerinde güncelleme yapmak için bu DispatchQueue.main nesnesini kullanabiliriz. Aşağıdaki örnekte yeni bir thread yaratılmış ve bu thread'in her saniyede ürettiği değer bir UILabel nesnesinde gösterilmiştir:

```

import UIKit

class ViewController: UIViewController {

 @IBOutlet weak var label: UILabel!

 override func viewDidLoad()

```

```

{
 super.viewDidLoad()

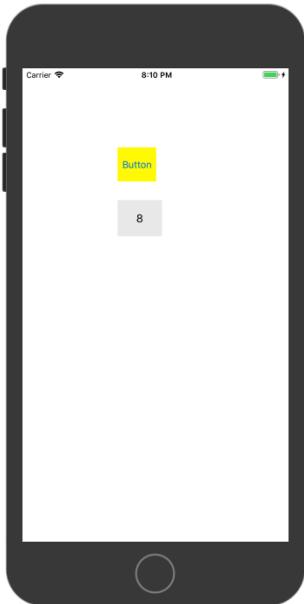
}

@IBAction func onButtonClicked(_ sender: Any)
{
 let dq = DispatchQueue(label: "MyThreadQueue", attributes: .concurrent)

 dq.async(execute: self.foo)
 print("Ok")
 //...
}

func foo()
{
 for i in 0..<10 {
 DispatchQueue.main.async {
 self.label.text = String(i)
 }
 sleep(1)
 }
}
}

```



Burada artık UILabel üzerinde güncelleme işlemi ana thread tarafından yapılmaktadır. Yarattığımız thread yapılacak işlemleri ana thread'in kuyruğuna atmaktır ve bu kodlar aslında ana threda tarafından çalıştırılmaktadır. Şimdi async metodu ile sync metodu arasındaki fark daha iyi anlaşılabilir. Örneğin:

```

func foo()
{
 for i in 0..<10 {
 DispatchQueue.main.async {
 self.label.text = String(i)
 sleep(1)
 }
 print("ok")
 }
}

```

}

Burada çalışma async ile yapılmıştır. Dolayısıyla sleep ile bekletilen aslında ana thread'tır. Bu durumda "ok" yazıları hemen console ekranına çıkacaktır. Çünkü async çağrılan thread'i bekletmemektedir. Şimdi aynı kodun sync'li versyonuna bakalım:

```
func foo()
{
 for i in 0..<10 {
 DispatchQueue.main.sync {
 self.label.text = String(i)
 sleep(1)
 }
 print("ok")
 }
}
```

Burada artık "ok" yazıları etiket güncellendikten sonra console ekranına çıkacaktır.

Şimdi GUI arayüzüne bir tane UIProgressView yerleştirip onu başka bir thread'ten azar azar artıralım:

```
import UIKit

class ViewController: UIViewController {

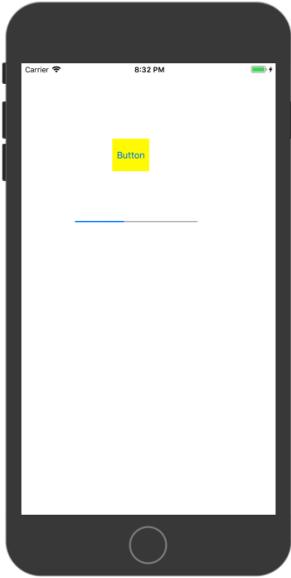
 @IBOutlet weak var progressView: UIProgressView!

 override func viewDidLoad()
 {
 super.viewDidLoad()
 progressView.progress = 0
 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 let dq = DispatchQueue(label: "MyThreadQueue", attributes: .concurrent)

 dq.async(execute: self.foo)
 //...
 }

 func foo()
 {
 for _ in 0..<10 {
 DispatchQueue.main.async {
 self.progressView.progress += 0.1
 }
 sleep(1)
 }
 }
}
```



Bu örnekte UIProgressView'daki artırımın thread açılmadan ana thread tarafından yapıldığında donma etkisi oluşacağına dikkat ediniz.

DispatchQueue sınıfının static global isimli metodu bize concurrent bir DispatchQueue nesnesi vermektedir. Yani aslında biz thread işlemleri için ayrı bir DispatchQueue nesnesi yaratmak yerine doğrudan yaratılmış olan bu nesneyi de kullanabiliriz. Yukarıdaki UIProgressView örneğini aşağıdaki gibi daha kompakt hale getirebiliriz:

```
import UIKit

class ViewController: UIViewController {

 @IBOutlet weak var progressView: UIProgressView!

 override func viewDidLoad()
 {
 super.viewDidLoad()
 progressView.progress = 0
 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 DispatchQueue.global(qos: .default).async {
 for _ in 0..<10 {
 DispatchQueue.main.async {
 self.progressView.progress += 0.1
 }
 sleep(1)
 }
 }
 //...
 }
}
```

Genellikle programcılar bu global nesnesini kullanmaktadır.

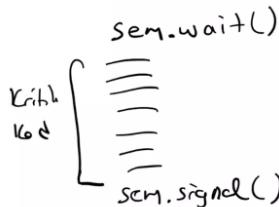
Bir DispatchQueue nesnesi oluşturulurken "quality of service" ismi altında yapılacak thread'e bir öncelik verilebilmektedir. Bu öncelik high, low gibi genel bir kategori belirtmektedir. "Quality of service" için şu değerler kullanılabilir:

```
.background
.default
.unspecified
.userInitiated
.userInteractive
.utility
```

defult normalk önceliği belirtir. Programcı eğer konu hakkında ayrıntılı bilgeye sahip değilse bu seçenekü seçmelidir. background seçeneği düşük bir thread önceliğini temsil eder. Arka planda sessizce çalışan thread'ler için bu öncelik belirlenebilir. userInteractive thread'ler kullanıcı tarafından tetiklenen ancak zamanının büyük kısmını uykuda geçiren thread'ler için uygundur.

GCD tarzı asenkron thread işlemlerinde thread'lerin senkronize edilmesi gerekebilir. Gerçekten de asenkron çalışan kodlar ortak birtakım kaynaklara erişirken sorunlar oluşabilmektedir. İşte senkronizasyon amacıyla birkaç senkronizasyon nesnesi bulundurulmuştur. Bu amaçla kullanılan en önemli senkronizasyon nesnesi DispatchSemaphore nesneleridir. DispatchSemaphore sınıfı semafor kavramını gerçekleştirmektedir. Bu nesneyle kritik kod şöyle oluşturulur:

```
let sem = DispatchSemaphore(value:1)
```



Thread akışlarından biri kritik koda girdiğinde semaphore sayacı wait fmetodu tarafından 1 eksiltılır. Artık sayıç 0 olduğu için diğer thread2ler kritik koda girmeden bekleyecektir. signal metodu sayıç 1 artırmaktadır. Eğer semaphore'in başlangıç sayıç 1 ise böyle semaphore'lara "ikili semaphore'lar (binary semaphores)" denilmektedir. Örnek bir kullanım şöyle olabilir:

```
import UIKit

class ViewController: UIViewController {
 let sem = DispatchSemaphore(value:1)

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 DispatchQueue.global(qos: .default).async {
 for _ in 0..<10 {
 self.foo("thread-1")
 }
 }

 DispatchQueue.global(qos: .default).async {
 for _ in 0..<10 {
 self.foo("thread-2")
 }
 }
 }
}
```

```

 }
 }

//...

}

func foo(_ str: String)
{
 sem.wait()
 print("-----")
 print("\(str): Step 1")
 usleep(useconds_t(Int.random(in: 100000...300000)))
 print("\(str): Step 2")
 usleep(useconds_t(Int.random(in: 100000...300000)))
 print("\(str): Step 3")
 usleep(useconds_t(Int.random(in: 100000...300000)))
 print("\(str): Step 4")
 usleep(useconds_t(Int.random(in: 100000...300000)))
 print("\(str): Step 5")
 usleep(useconds_t(Int.random(in: 100000...300000)))
 sem.signal()
}
}

```

Üretici-Tüketici problemi (producer-consumer problem) de bilindiği iki semaphore ile çözülebilmektedir. Bu problemde thread'lerden birine "üretici (producer) thread" diğerine "tüketici (consumer) thread" denilmektedir. Üretici thread bir değer elde edip bunu tüketici thread'e verir. Tüketici thread de değeri işler. Bu problemde üreticinin tüketici henüz eskisi değeri almadan yeni bir değeri paylaşılan alana yerleştirmemesi gereklidir. Benzer biçimde tüketici thread de üretici henüz yeni değeri koymadan eski değeri ikinci kez almamalıdır. Çözüm şöyle yapılabilir:

```

import UIKit

class ViewController: UIViewController {

 let semProducer = DispatchSemaphore(value:1)
 let semConsumer = DispatchSemaphore(value:0)
 var shared = 0

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }

 @IBAction func onButtonClicked(_ sender: Any)
 {
 DispatchQueue.global(qos: .default).async(execute: producerFunc)
 DispatchQueue.global(qos: .default).async(execute: consumerFunc)
 }

 func producerFunc()
 {
 var i = 0

 while true {
 usleep(useconds_t(Int.random(in: 100000...300000)))
 semProducer.wait()
 shared = i
 semConsumer.signal()
 }
 }
}

```

```

 if i == 99 {
 break
 }
 i += 1
 }
}

func consumerFunc()
{
 var val: Int

 while true {
 semConsumer.wait()
 val = shared
 semProducer.signal()
 usleep(useconds_t(Int.random(in: 100000...300000)))
 print(val, terminator: " ")
 if val == 99 {
 break
 }
 }
 print()
}
}

```

## 2) OperationQueue Yöntemi

Thread oluşturmak için OperationQueue yöntemi aslında GCD yönteminin üzerine oturtulmuştur. Yani OperationQueue arka planda GCD kullanılarak gerçekleştirılmıştır. Bu nedenle OperationQueue yöntemi daha yüksek seviyelidir. OperationQueue yöntemi tipik olarak şöyle uygulanmaktadır:

1) Kuyruğa atılacak her bir işe "iş (job)" ya da "görev (task)" denilmektedir. Görevler Operation sınıfından türetilmiş sınıflarla temsil edilmektedir. Operation abstract bir sınıfır. Programcı Operation sınıfından sınıf türetip o türemiş sınıfta main metodunu override eder. Örneğin:

```

import UIKit

class MyOperation: Operation
{
 var str: String!

 init(_ str: String)
 {
 super.init()
 self.str = str
 }

 override func main()
 {
 for i in 0..<10 {
 print("\(self.str!): \(i)")
 sleep(1)
 }
 }
}

```

2) Bir OperationQueue nesnesi oluşturulur. Bu nesne sınıfın özniteliği yapılabilir. Örneğin:

```
let opq = OperationQueue()
```

3) Artık Operation nesneleri yaratılarak OperationQueue sınıfının addOperation(\_:) örnek metodu ile görevler kuruğa eklenir. Örneğin:

```
import UIKit

class ViewController: UIViewController {

 let opq = OperationQueue()

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let op1 = MyOperation("Task-1")
 let op2 = MyOperation("Task-2")

 opq.addOperation(op1)
 opq.addOperation(op2)
 //
 }
}
```

4) Operation nesneleri OperationQueue nesnesine eklendiği anda ayrı bir thread olarak çalışma başlatılır. Söz konusu thread'in çalışmaya başladığı yer override sttiğimiz main metodunu olacaktır. Aslında OperationQueue bize her bir görevin ayrı bir thread'te çalıştırılıp çalıştırılmayacağı konusunda bize bir garanti vermemeğtedir. Bu kısım tamamen OperationQueue içerisinde soyutlanmıştır. Ancak programcı kuyruğu belli ölçülerde kontrol edebilmektedir.

5) OperationQueue içerisindeki görevlerin sayısı sınıfın operationCount isimli örnek özniteliği ile elde edilebilir. Bnezer biçimde kuyruğa eklediğimiz Operation nesnelerini de biz istersek sınıfın operations isimli örnek özniteliği ile elde edebiliriz. operations örnek özniteliği [Operation] türündendir. Çalışması biten (yani main metodу biten) görevler otomatik olarak kuyruktan atılmaktadır. Yani operationCount bize aktif biçimde çalışmakta olan görevlerin sayısını verir.

OperationQueue sınıfının cancelAllOperations isimli örnek metodu tüm Operation nesnelerinde cancel isimli metodun çağrılmasına yol açar. Programcı kendi Operation sınıfında bu metodı override edebilir. Operation sınıfının cancel metodu ise sınıfın isCancelled property'sini true yapmaktadır. İşte programcı Operation sınıfının cancel metodu tarafından set edilen isCancelled property'sine bakarak çıkış kendisi yapabilir.



```

override func cancel()
{
 //...
 super.cancel()
}

override func main()
{
 //...
 if isCancelled {
 //...
 }
}

```

Örneğin:

```

import UIKit

class MyOperation: Operation
{
 var str: String!

 init(_ str: String)
 {
 super.init()
 self.str = str
 }

 override func main()
 {
 for i in 0..<10 {
 print("\(self.str!): \(i)")
 sleep(1)
 if isCancelled {
 break
 }
 }
 }

 override func cancel()
 {
 print("cancellation occurs")
 super.cancel()
 }
}

```

OperationQueue sınıfının maxConcurrentOperationCount isimli read/write property elemanı kuyruktaki en fazla kaç görevin aynı anda farklı thread'ler atarından çalıştırılacağını belirtir. Başka bir deyişle OperationQueue en fazla burada belirtilen miktarda thread yaratmaktadır. Bu property'nin default değeri defaultMaxConcurrentOperationCount property'si ile belirtilen değerdir. Örneğin maxConcurrentOperationCount değeri 5'e çekilmiş olsun. Biz bu durumda kuyruğa 10 tane görev yerleştirsek bile bunların yalnızca 5 tanesi açılışacaktır. Bu görevlerden biten yerine kuyrukta sırada bekleyen görev çalışmaya başlayacaktır. Fakat aynı anda çalışan görevlerin sayısı en fazla 5 olacaktır.

Aslında Operation nesnelerine yukarıdaki gibi bizim isim vermek için ek bir öznitelik kullanmamıza gerek yoktur. Zaten Operation sınıfının name isimli örnek özniteliği bu amaçla kullanılabilirdir. Örneğin:

```
import UIKit

class MyOperation: Operation
{
 init(_ name: String)
 {
 super.init()
 self.name = name
 }

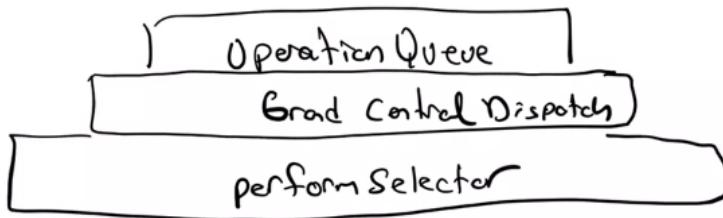
 override func main()
 {
 for i in 0..<10 {
 print("\(self.name!): \(i)")
 sleep(1)
 if isCancelled {
 break
 }
 }
 }

 override func cancel()
 {
 print("cancellation occurs")
 super.cancel()
 }
}
```

OperationQueue sınıfının waitUntilAllOperationsAreFinished() isimli metodu kuyruktaki tüm görevler bitene kadar bu metodu çağrıran thread'i blokede bekletir.

### 3) performSelector Fonksiyonu İle Thread'lerin Aşağı Seviyeli Yaratılması

Üç thread katılım yönteminden en aşağı seviyeli olan performSelector yöntemidir. Gerçekten de bu yöntem arka planda zaten GCD ve dolayısıyla OperationQueue tarafından kullanılmaktadır.



Yönetmin kullanımı oldukça kolaydır.

1) Thread olarak çağrılacak fonksiyon ya da metot @objc özelliği ile bildirilir. Örneğin:

```
@objc
func foo(_ name: String)
{
 for i in 0..<10 {
```

```

 print(i)
 sleep(1)
 }
}

```

2) NSObject sınıfından gelen performSelector(inBackground:with:) metodu fonksiyon argümanıyla çağrılarak çağrılır. Örneğin:

```

import UIKit

class ViewController: UIViewController {

 override func viewDidLoad()
 {
 super.viewDidLoad()

 }

 @objc
 func foo(_ name: String)
 {
 for i in 0..<10 {
 print(i)
 sleep(1)
 }
 }

 @IBAction func buttonRunClicked(_ sender: Any)
 {
 self.performSelector(inBackground: #selector(foo), with: "Test")
 }
}

```

Göründüğü gibi biz bu yöntemde tek bir metod çağrıları ile thread'i yaratıyoruz. performSelector metodu NSObject sınıfından gelmektedir. Yani global bir fonksiyon değildir. Bu metodun ikinci parametresi olan with thread fonksiyonuna geçirilecek argümanı belirlemekte kullanılır. Bu argüman herhangi bir türden olabilmektedir. Burada önemli bir nokta bu b嘿imde yaratılan thread akışlarının Swift'in çöp toplama mekanizmasının dışında kaldığıdır. Bunun için programcının autoreleasepool isimli fonksiyonla bir blok oluşturması gereklidir. Örneğin:

```

@objc
func foo(_ name: String)
{
 autoreleasepool {
 for i in 0..<10 {
 print(i)
 sleep(1)
 }
 }
}

```

## Timer Kullanımı

iOS'ta arka planda periyodik işlem yapanın biri thread kullanmaktadır. Önceki konuda da deðindiðimiz gibi UI elemanlar yalnızca ana thread tarafından güncellenmelidir. Dolayısıyla önceki konuda başka bir thread'ten UI elemanlarının güncellenmesi için main isimli DispatchQueue nesnesini kullanmıştık. iOS'ta arka planda periyodik işlemler yapabilmek için Timer mekanizması da kullanılmaktadır. Timer mekanizması ana thread tarafından işletildiği için bu mekanizmada doğrudan UI elemanlar güncellenebilir.

Timer mekanizması şöyle kullanılmaktadır:

1) Programcı sınıfın özniteliği olarak Timer sınıfı türündne bir değişken bildirir. Aslında Timer değişkeni yerel bir değişken de olabilir. Ancak başka yerlerden bu nesnenin kullanılabilmesi için sınıfın örnek özniteliği yapılması uygun olur. Örneğin:

```
import UIKit

class ViewController: UIViewController {
 var timer: Timer!

 override func viewDidLoad() {
 super.viewDidLoad()
 // Do any additional setup after loading the view, typically from a nib.
 }

 @IBAction func buttonOkClicked(_ sender: Any)
 {
 //...
 }
}
```

2) Timer nesnesi Timer sınıfının static scheduledTimer isimli metodlarıyla yaratılır. Örneğin:

```
import UIKit

class ViewController: UIViewController {
 var timer: Timer!

 override func viewDidLoad() {
 super.viewDidLoad()
 // Do any additional setup after loading the view, typically from a nib.
 }

 func timerProc(timer: Timer)
 {
 print(".", terminator: "")
 }

 @IBAction func buttonOkClicked(_ sender: Any)
 {
 timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true, block: timerProc)
 }
}
```

scheduledTimer metodu timer'ı yaratır yaratmaz hemen çalıştırmaktadır.

3) Timer sınıfının örnek invalidate metodu timer'ı durdurmaktadır. Artık invalidate edilmiş timer bir daha kullanılamaz. Örneğin:

```
import UIKit

class ViewController: UIViewController {
 var timer: Timer!

 override func viewDidLoad() {
 super.viewDidLoad()
```

```

 // Do any additional setup after loading the view, typically from a nib.
}

func timerProc(timer: Timer)
{
 print(".", terminator: "")
}

@IBAction func buttonOkClicked(_ sender: Any)
{
 timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true, block: timerProc)
}

@IBAction func buttonStopClicked(_ sender: Any)
{
 timer.invalidate()
}
}

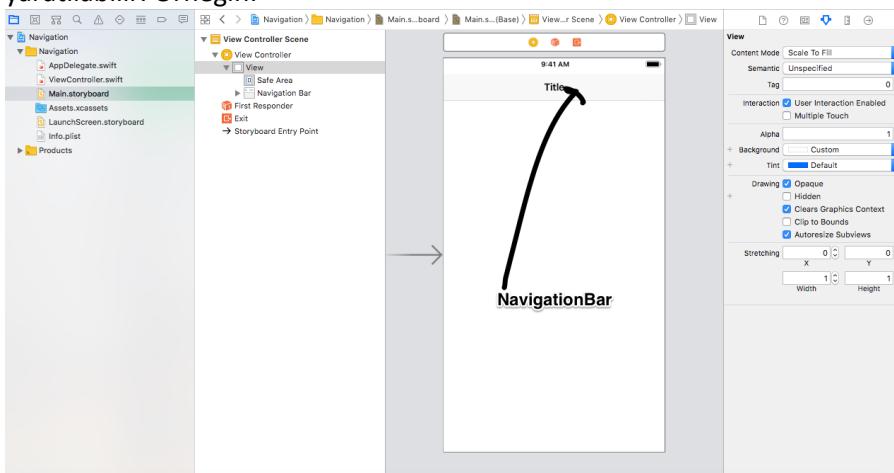
```

Timer sınıfı belirli bir zaman geldiğinde (alarm mekanizması olarak düşünebilirsiniz) belli bir kodun çalışmasını da sağlayabilmektedir. Birden fazla timer nesnesi de yaratılabilmektektir.

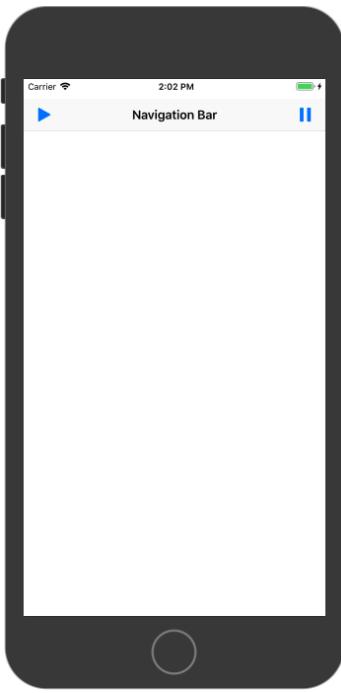
### NavigationBar ve Toolbar Kullanımı

Ekranlar arasında geçiş yapılan ISO uygulamalarında NavigationBar isimli UI elemanı sıkça kullanılmaktadır. IOS'taki Toolbar elemanı ise daha çok masaüstü sistemlerindeki StatusBar kontrollerine benzetilebilir.

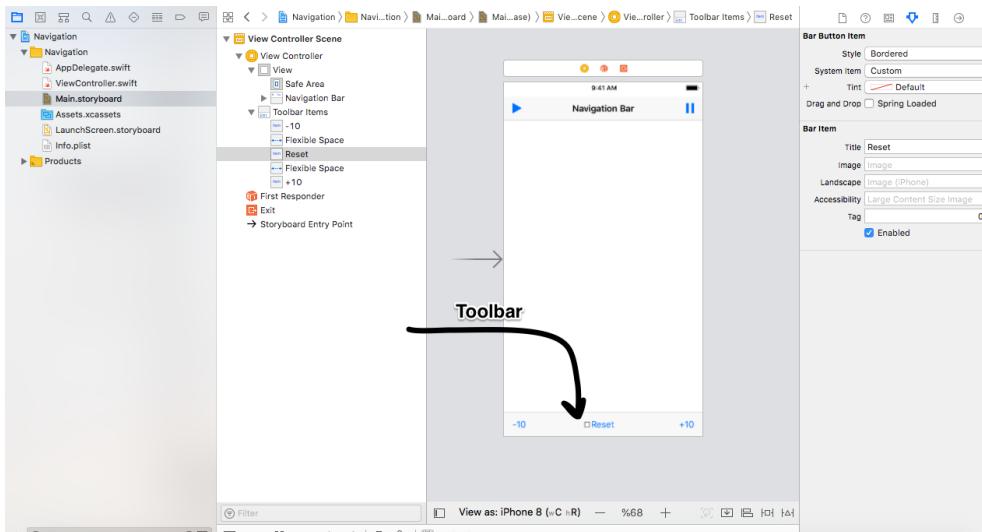
Bir NavigationBar yine manule biçimde yaratılabilir. Ya da Object Library'den sürükleneerek view üzerine bırakılarak da yaratılabilir. Örneğin:



NavigationBar pencerelerinin üzerine genellikleBarButton nesneleri yerleştirilmektedir. YerleştirilenBarButton nesnelerinin alt türleri vardır. Bu alt türler bu nesneler üzerindeki icon'ların belirlenmesini sağlamaktadır. Örneğin bir NavigationBar'ın sol ve sağına ikiBarButton nesnesi yerleştirmış olalım:



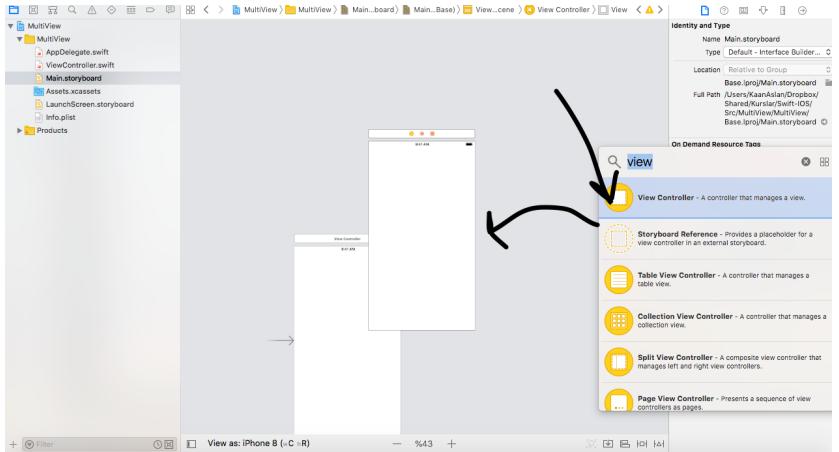
Toolbar元件はNavigationBarと同様です。しかし、通常は画面の下部に配置されます。Toolbarに含まれるBarButtonItemは、デフォルトでは横並びで表示されますが、Space要素を介して間隔を設けることができます。Space要素にはFixed SpaceとFlexible Spaceがあります。Flexible Spaceは、指定した幅よりも広くなる場合に自動的に拡張する機能を持っています。



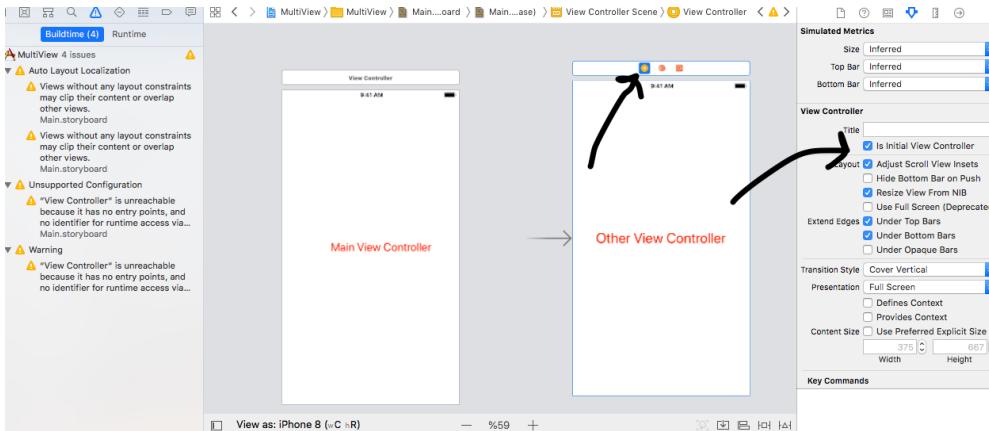
## Birden Fazla View Controller İle Çalışmak

Şimdideki uygulamalarımızda tek bir view controller ile çalıştık. Dolayısıyla uygulamalarımızın tek bir ana penceresi vardı. Halbuki pek çok uygulamada bizim ekranda gördüğümüz view belli düğmelere basıldığında değişmektedir. Peki bu nasıl sağlanmaktadır? Teorik olarak tek bir view oluşturup oradaki alt view'ları silip ekleyerek sanka ekran geçisi yapılıyor izlenimi verilebilir. Ancak bu yöntem hem zordur hem de bu yöntemde görsel tasarım yapılamaz. İşte iOS programcıları ekran geçişleri için birden fazla view controller kullanmaktadır. Birden fazla viewcontroller kullanma işlemi şöyle gerçekleştirilmektedir:

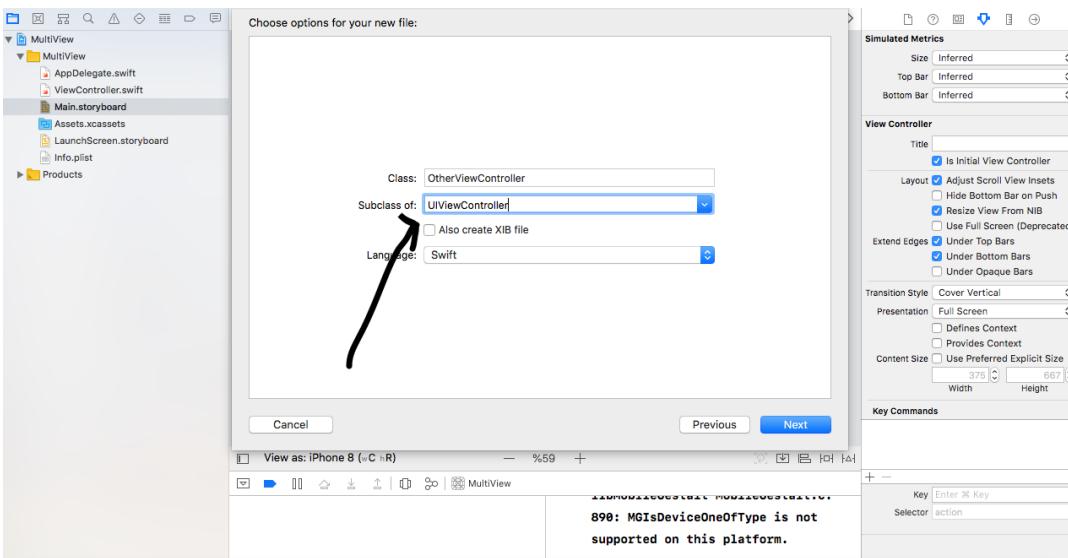
1) Library kısmından ViewController seçilerek Main.storyboard'a sürüklenecek bırakılır. Örneğin:



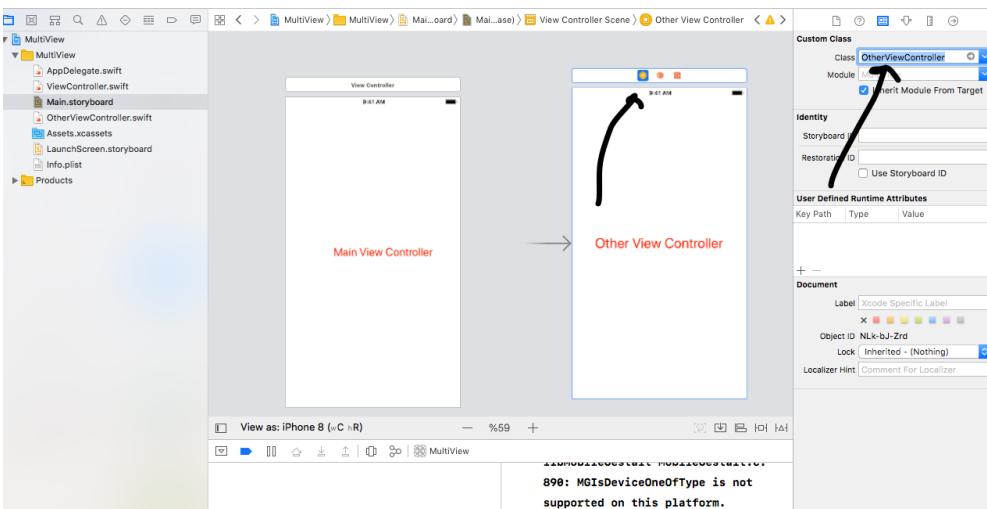
Buradaki ok (arrow) hangi view controller ile uygulamanın başlatılacağını belirtmektedir. Bu oku sürükle bırakla istedigimiz view controller'a getirebiliriz. Ya da eşdeğer bir biçimde bir view controller'i seçerek "Is Initial View Controller" seçenek kutusu çarpılanır.



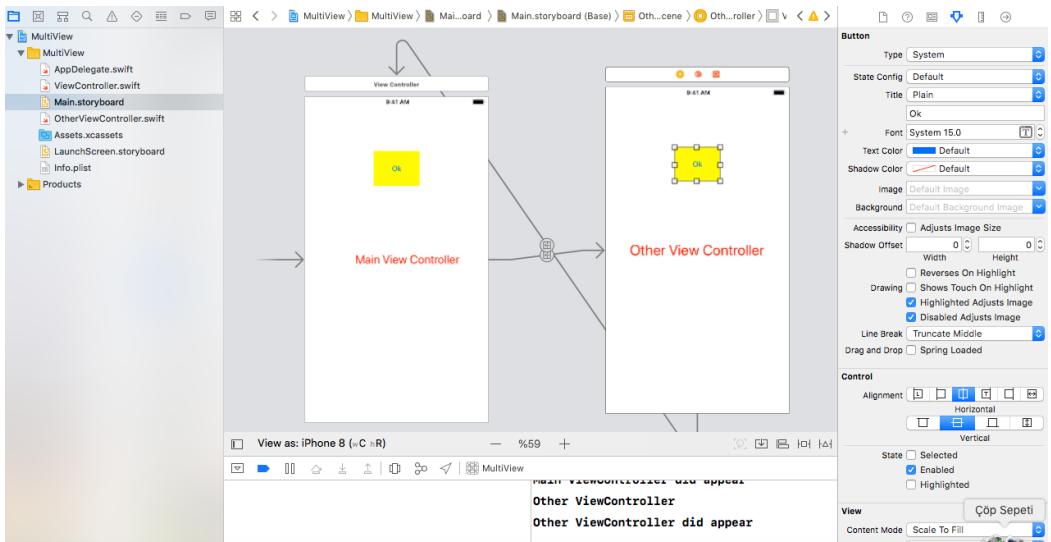
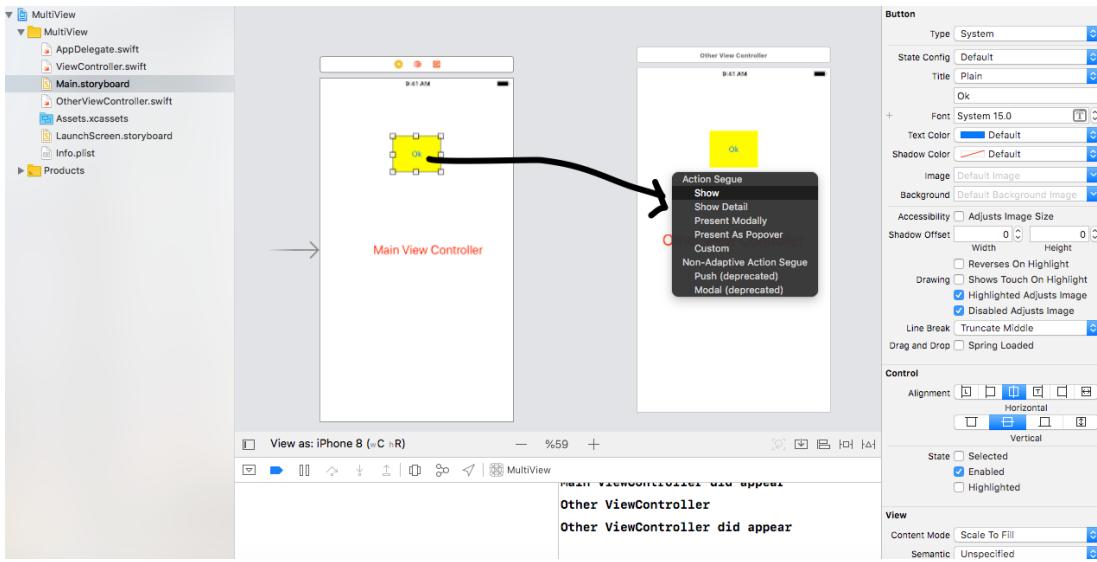
2) Sürüklenebilir bırakılan viewcontroller için programcının bir controller sınıfı oluşturup ilişkilendirme yapması gereklidir. Bunun için File/New/File seçilir. Buradan "Cocoa Touch Class" seçilerek taban sınıf UIViewController biçiminde belirlenir.



Bundan sonra Main.storyboard'a gelip sınıfı ilişkilendireceğimiz viewcontroller'i seçip "identity inspector"dan ilişkilendirmeyi yaparız.



4) Pekiyi bir view'dan diğerine geçiş nasıl yapılmaktadır? Örneğin ana view bir login ekranı olabilir ve kullanıcı login olduktan sonra artık başka bir view görüntülenmek istenebilir. Bu işlem programlama yoluyla yapılabileceği gibi görsel yolla da yapılabilmektedir. Görsel yolla yapmak için tıklandığında geçiş yapılacak bir düğme view'a yerleştirilir. Sonra o düğme üzerinden Ctrl tuşuyla fare sürüklerek geçilecek view'ya gelinir. Çıkan popup'ta "view" seçilmelidir. Örneğin:

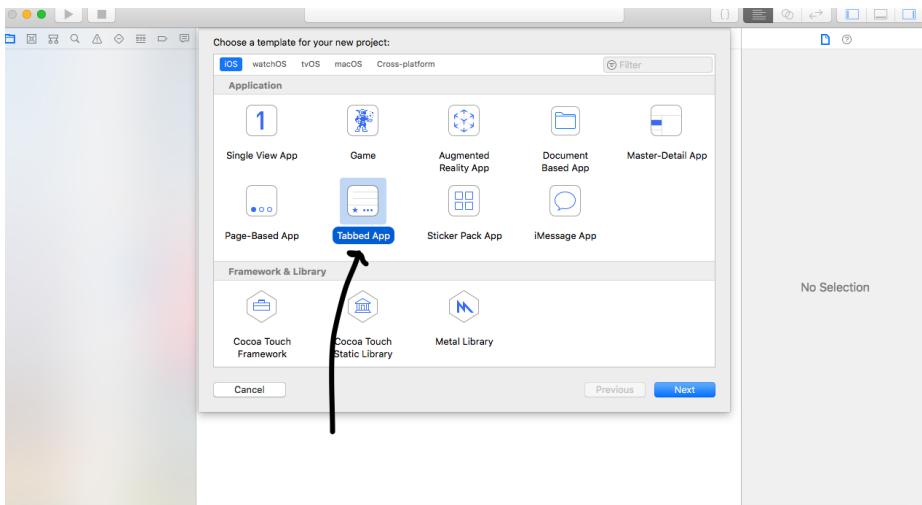


Burada hangi view'da düğmeye tıklarsak diğerine geçilecektir. Bu biçimdeki geçişlerde her defasında yeniden ViewController nesnesi yaratılıp viewDidLoad metodu yeniden çağrılmaktadır.

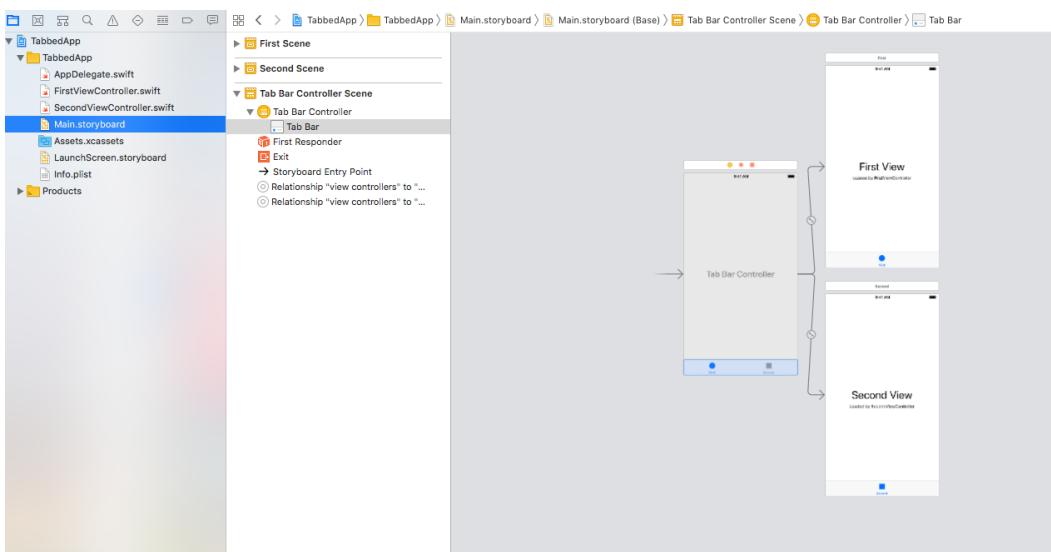
## Tab'lı Uygulamalar

Tab'lı uygulamalar bir tabbar vasıtasıyla geçiş yapılan uygulamalardır. Sırasıyla şu aşamalardan geçirilerek oluşturulmaktadır:

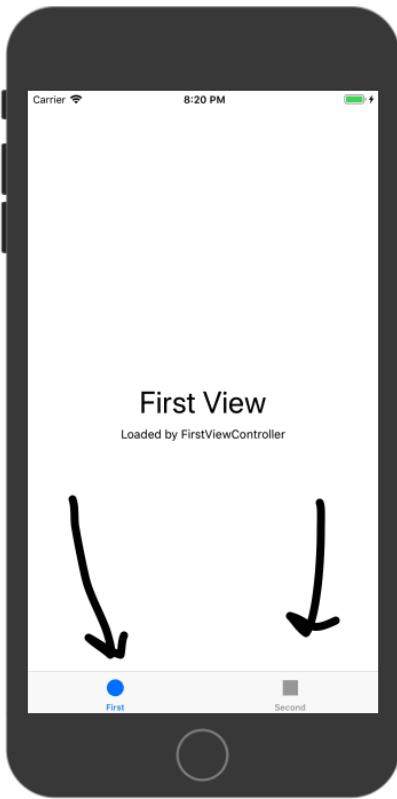
- 1) File/New/Project diyalog penceresi açılır. Oradan "Tabbed App" seçilir.



Uygulama oluşturulduğunda wizard ana viewcontroller'da bir tane tabbar ve iki tane ViewController oluşturur.

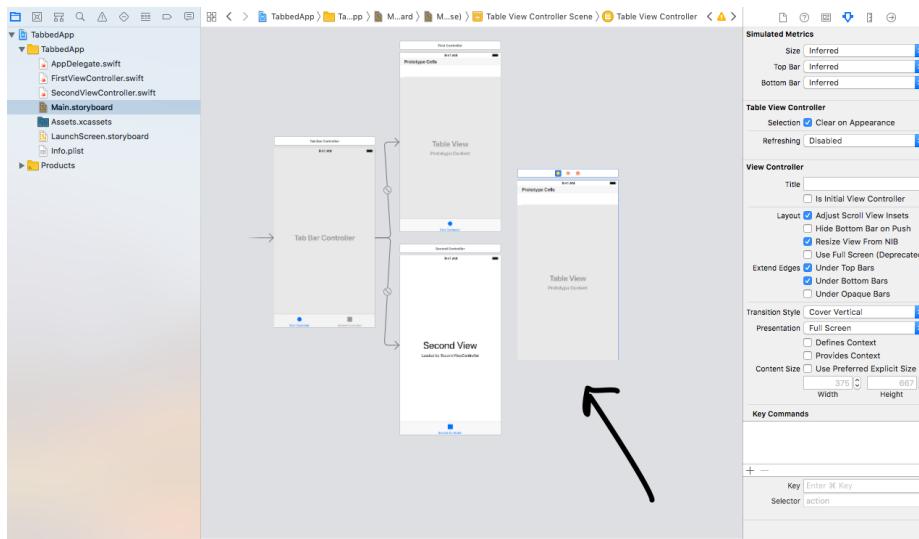


Uygulamayı çalıştırduğumızda şöyle bir görüntü elde edilecektir:

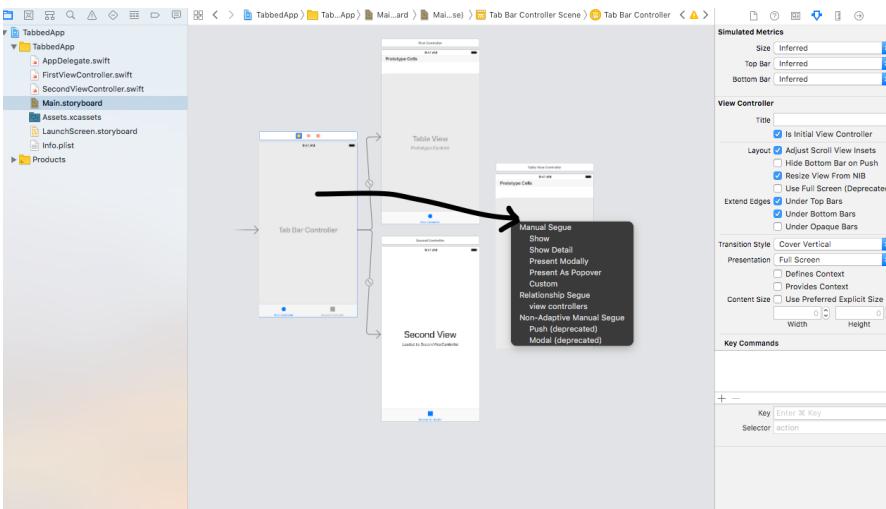


Bir Tabbed uygulamada bir view ilk kez görüntülendiğinde ilgili ViewController sınıfının viewDidLoad isimli sanal metodu yalnızca bir kez çağrılmaktadır. Eğer view her aktif hale getirildiğinde bir metodun çağrılması isteniyorsa bu durumda ViewController sınıfında viewWillAppear(\_:) metodunu override edilmelidir. Aslında viewDidAppear(\_:) isimli bir metod da vardır. Ancak bu metod henüz view görüntülenmeden önce çağrılmaktadır. Halbuki viewWillAppear(\_:) metodunu view görüntülendikten sonra çağrılmaktadır. Tabii bu metodlar view ilk kez görüntülendiğinde de çağrılmaktadır.

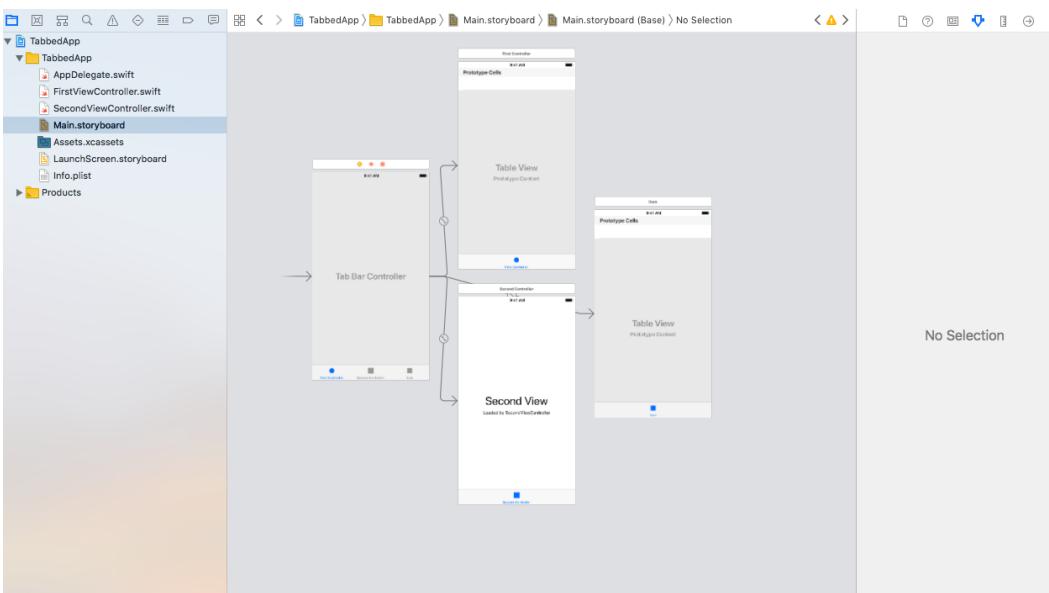
2) Tabbed uygulama yaratıldıktan sonra programcı isterse ona başka tab'lar yerleştirebilir. Bunun için Object Library'den bir viewController nesnesi storyboard'a bırakılır.



Bundan sonra storyboard'ta ana kontroller penceresinden Ctrl tuşuna basılarak fare yeni eklenen controller'a sürüklendir.



Çıkan popup pencereden "ViewController" seçilir.



Bu örnekte biz üçüncü tab için viewcontroller olarak TableViewController yerleştirdik. Buraya normak bir ViewController da yerlestirebilirdik.

3) ViewController sürüklənerek stroryboard'a bırakılıp görsel ilişkilendirme yapıldıktan sonra sınıfal ilişkilendirme de yapılmalıdır. Yani daha önceden de yaptığımız gibi yeni bir ViewController sınıfı yaratıp bunu görseldeki view controller ile ilişkilendirmemiz gerekdir. Bu işlem yine File/New/File/Cocoa Touch Class seçilerek yapılır.

Böylece artık yeni bir tab'ı eklemiş olmaktayız.

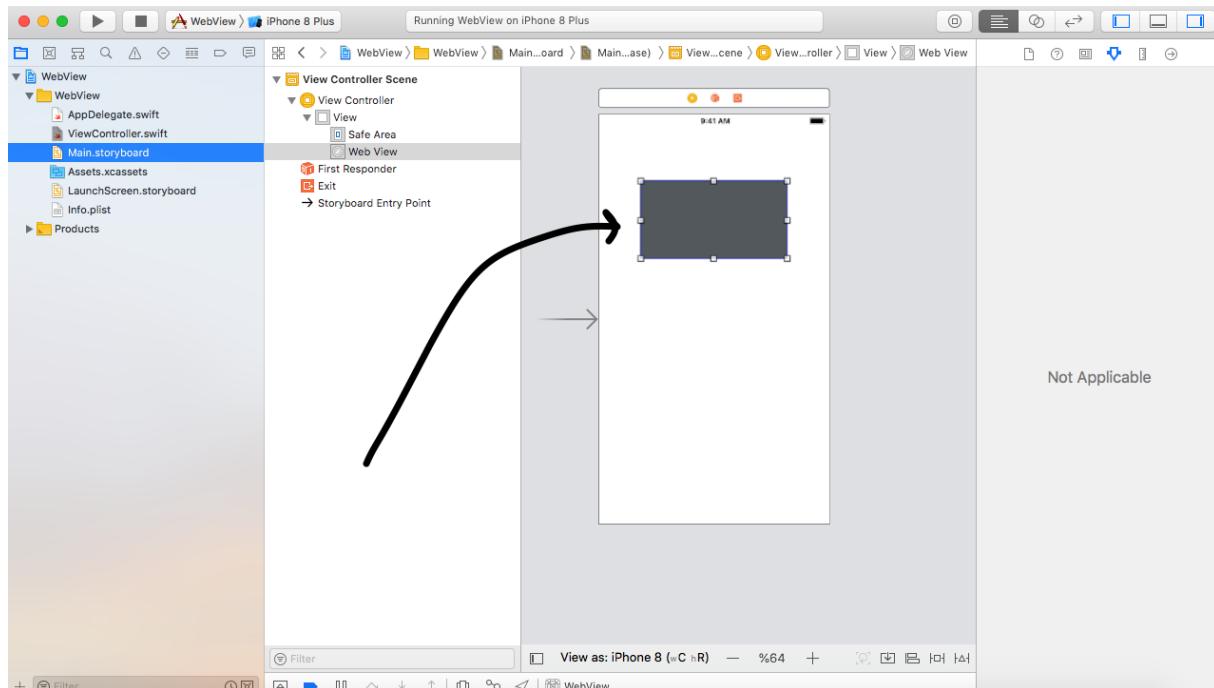
### **WKWebView Kullanımı**

Web sayfası görüntülemek için her framework'te kontroller bulundurulmaktadır. IOS Cocoa Touch'ta da bu işlem büyük ölçüde WKWebView sınıfıyla yapılmaktadır. Bu sınıf hem bir web sitesini browser gibi gösterebilirken aynı zamanda bir HTML dosyanın içini ve hatta içerisinde HTML ve CSS bilgilerinin bulunduğu yazışal (string) ifadeleri de görüntüleyebilmektedir. Programcılar bu view penceresini bazen hyper text görüntüleme için de kullanılabilmektedir. Aslında bir süre önceye kadar bu işlem için ağırlıklı biçimde UIWebView sınıfı kullanılıyordu. Ancak WKWebView sınıfı zaten UIWebView sınıfının işlevselliliklerini kapsamaktadır. Bu sınıfın UIWebView sınıfından daha üstün yetenekleri vardır.

Artık IOS'un 102lu versiyonlarıyla birlikte UIWebView "deprecated" yapılmıştır. UIWebView kendi içerisinde .pdf gibi, .doc gibi dosyaları da görüntüleyebilmektedir. Bu da onun başka bir kullanım gereklisini oluşturmaktadır.

WKWebView penceresi şu aşamalardan geçilerek kullanılmaktadır:

- 1) Object Library'den WKWebView nesnesi sürükleneerek storyboard'a bırakılır ve bunun için bir outlet oluşturulur.



```
import UIKit
import WebKit

class ViewController: UIViewController {

 @IBOutlet weak var webView: WKWebView!

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }
}
```

Maalesef outlet oluşturulduğunda XCode sınıfın içinde bulunduğu modülü import etmemektedir. WebKit modülünün programcı tarafından manuel olarak import edilmesi gereklidir.

- 2) Bir Web sitesini görüntüleyebilmek için onun URL'si oluşturularak WKWebView sınıfının load(\_:) metodu kullanılır. load metodunun parametrik yapısı şöyledir:

```
func load(_ request: URLRequest) -> WKNavigation?
```

Metot bizden URLRequest isimli bir yapı nesnesi istemektedir. URLRequest görüntülenecek URL'i tutan ayrı bir yapıdır. Bir URLRequest nesnesi de URL nesnesi verilerek yaratılmaktadır. Örneğin:

```
import UIKit
import WebKit
```

```

class ViewController: UIViewController {

 @IBOutlet weak var webView: WKWebView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let urlRequest = URLRequest(url: URL(string: "https://terapikulubu.com")!)
 webView.load(urlRequest)
 }
}

```

Bir dosyanın içerisindeki HTML kodlarını görüntülemek için sınıfın loadFileURL(\_:allowingAccessTo:) metodu kullanılmaktadır. Örneğin:

```

import UIKit
import WebKit

class ViewController: UIViewController {

 @IBOutlet weak var webView: WKWebView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let url: URL? = Bundle.main.url(forResource: "test", withExtension: "html")

 webView.loadFileURL(url!, allowingReadAccessTo: url!)
 }
}

```

Bir HTML string'i de sınıfın loadHTMLString(\_:baseURL:) metodu ile görüntüleyebiliriz. Örneğin:

```

import UIKit
import WebKit

class ViewController: UIViewController {

 @IBOutlet weak var webView: WKWebView!

 override func viewDidLoad()
 {
 super.viewDidLoad()
 webView.loadHTMLString("<html><head><title>Test</title></head><body><h1>This is a test</h1><p>Yes this is a test</p></body>", baseURL: nil)
 }
}

```

3) WKWebView sınıfının goBack() ve goForward isimli metodları ileri-geri gidişler için kullanılır. Tabii geride ve ileride bir URL yoksa bu gidişler gerçekleşmeyecektir. Bunun için programcı isterse canGoBack() ve canGoForward metodları ile geride ve ileride URL var mı diye bakabilir. Örneğin:

```

import UIKit
import WebKit

```

```

class ViewController: UIViewController {

 @IBOutlet weak var webView: WKWebView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

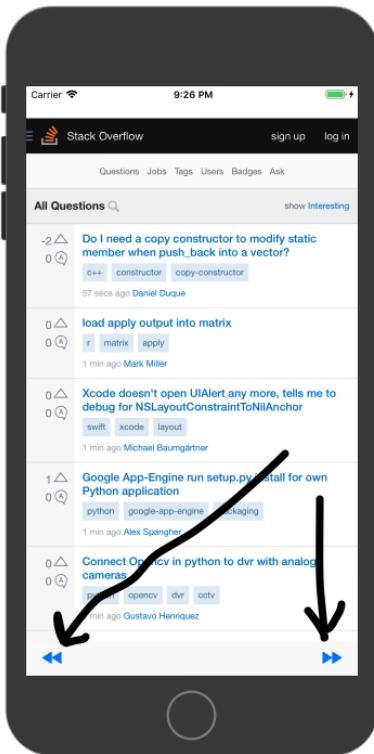
 let urlRequest = URLRequest(url: URL(string: "https://stackoverflow.com")!)
 webView.load(urlRequest)

 }

 @IBAction func goBackButtonClicked(_ sender: Any)
 {
 webView.goBack()
 }

 @IBAction func goForwardButtonClicked(_ sender: Any)
 {
 webView.goForward()
 }
}

```



Sınıfın go(to:) isimli metodu gerideki ya da ilerideki belli bir sayfaya gitmek için kullanılır. Aslında view gezilen yerleri kendi içerisinde WKBackForwardList isimli bir collection içerisinde tutmaktadır. Bu collection WKBackForwardListItem nesnelerini tutar. İşte go(to:) metodu da parametre olarak WKBackForwardListItem nesnesini bizden istemektedir. Sınıfın backForwardList isimli property elemanı bize bu WKBackForwardList nesnesini vermektedir. Örneğin:

```
webView.go(to: webView.backForwardList.forwardItem!)
```

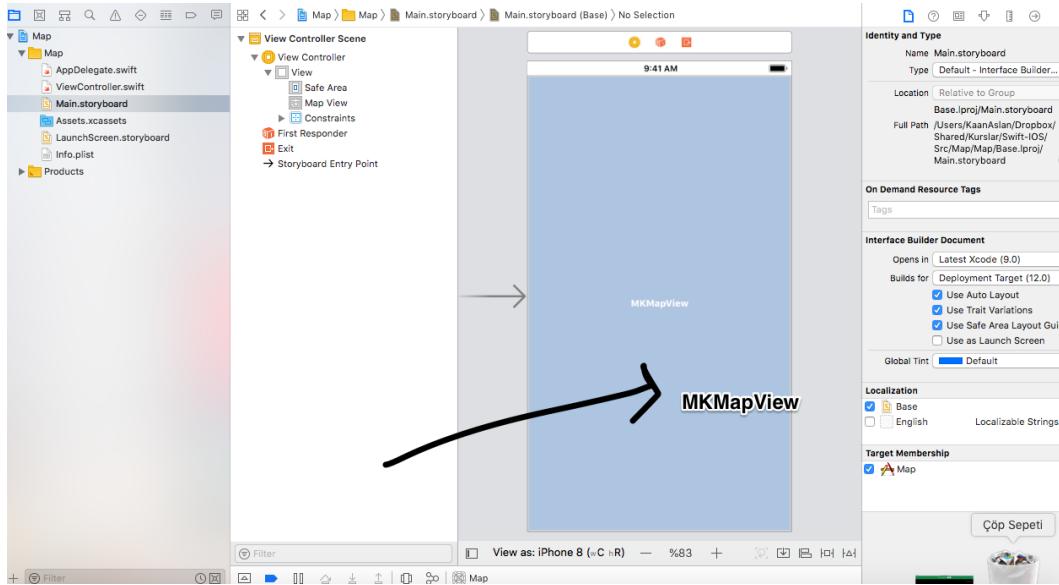
4) Bir HTML görüntülenirken programcı araya girerek bazı etkinliklerde bulunabilir. WKWebView nesnesi bazı

durumlarda programcının bazı metodlarını çağrılmaktadır. İşte bunun için WKWebView sınıfının iki delegate elemanı vardır. Bunlar uiDelegate ve navigationDelegate isimli elemanlardır. Bu elemanlar sırasıyla WKUIDelegate? ve WKNavigationDelegate? isimli protokol türündendir. Yani programcı bu elemanlara bu protokollerini destekleyen sınıfların referanslarını atayarak ilgili olaylar olduğunda kendi metodlarının çağrılmasını sağlayabilir. Tabii bu delegate sınıfları ana controller sınıfının kendisi de olabilir. Bu durumda bu elemanlara self referansının atanması gereklidir.

WKUIDelegate protokolünün üç önemli metodу wbView(\_:didStartProvisionalNavigation:) webView(\_:didCommit) ve webView(\_:didFinished) metodlarıdır. Bu metodlar sırasıyla istek server'a iletildiğinde, view ilk dataları aldığımda ve view tüm dataları aldığımda çağrılmaktadır. Örneğin programcı web sitesi yüklediğinde bir işlem yapacaksaya webView(\_:didFinished) metodunda yapabilir.)

## Harita Görüntüleme İşlemleri

iOS kendi içerisinde entegre edilmiş bir harita görüntüleme sistemine sahiptir. Zaten iOS'un kendi içerisinde de harita görüntülemek için bir "Harita (Map)" uygulaması da bulunmaktadır. Harita görüntülemek için MKMapView isimli View penceresi bulundurulmuştur. Bu pencere kodlama yoluyla ya da story board'da IB ile yaratılabilir.



Harita işlemleri için MapKit mdoülünün import edilmesi gereklidir. Programcı story board'a sürüklendiği MKMapView penceresi için bir outlet oluşturmalıdır.

```
import UIKit
import MapKit

class ViewController: UIViewController {

 @IBOutlet weak var mapView: MKMapView!

 override func viewDidLoad()
 {
 super.viewDidLoad()
 // Do any additional setup after loading the view, typically from a nib.
 }
}
```

Harita bu biçimde MKMapView ile ilk kez gösterildiğinde bölgelik ayarlarında belirtilen ülke gösterilir. Haritada belli bir yeri gösterebilmek için şu işlemlerin yapılması gereklidir:

1) Önce CLLocationCoordinate2D sınıfının init(latitude:longitude:) metodu ile bir CLLocationCoordinate2D nesnesi oluşturulur. Bu nesne oluştururken koordinatlar enlem (latitude) ve boylam (longitude) olarak verilir. Örneğin:

```
let location = CLLocationCoordinate2D(latitude: 41.065794, longitude: 29.002823)
```

Burada latitude enlem, longitude ise boylam bilgisi belirtmektedir. Buradaki birim "derece.dakika.saniye" değil "derece.derecenin milyonda biri" biçimindedir.

2) Görüntüleme için koordinat bilgisinin yanı sıra bir de zoom bilgisine gereksinim vardır. Bu zoom bilgisine span denilmektedir. Span (aralık anlamına geliyor) haritanın ne kadar bir aralığı göstereceği bilgisidir. Bu bilgi de noktalı bir sayı biçimindedir. Span oluşturmak için MKCoordinateSpan isimli sınıfın init(latitudeDelta:longitudeDelta:) isimli metodu kullanılır. Örneğin:

```
let span = MKCoordinateSpan(latitudeDelta: 0.01, longitudeDelta: 0.01)
```

3) Artık sıra görüntüleme için gereken bir region nesnesi oluşturmaya gelmiştir. Region nesnesi span ve location verilerek MKCoordinateRegion sınıfının init(center:span:) metodu ile oluşturulmaktadır. Örneğin:

```
let region = MKCoordinateRegion(center: location, span: span)
```

4) Oluşturulan region MKMapView sınıfının setRegion(\_:animated:) metodu ile görüntülenir.

Örneğin:

```
import UIKit
import MapKit

class ViewController: UIViewController {

 @IBOutlet weak var mapView: MKMapView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let location = CLLocationCoordinate2D(latitude: 41.068278, longitude: 29.000870)
 let span = MKCoordinateSpan(latitudeDelta: 0.001, longitudeDelta: 0.001)
 let region = MKCoordinateRegion(center: location, span: span)
 mapView.setRegion(region, animated: true)
 }
}
```

Tabii ilgili nokta haritada belli bir zoom değeriyle gösterildikten sonra programcı başka şeyler de yapmak isteyebilir. En çok istenen şeylerden biri "annotation" yapmaktadır. (Yani haritada belli bir noktaya bir işaretçi (igne) koymaktır.) Annotation işlemi şöyle yapılmaktadır:

1) Öncelikle bir tane MKPointAnnotation sınıfı türünden init() metodu ile nesne yaratılır. Örneğin:

```
let annotation = MKPointAnnotation()
```

2) Oluşturulan nesnenin çeşitli property'leri set edilerek annotation bilgisi nesneye yerleştirilir. Örneğin title property'si işaretçide görüntülenecek başlık yazısını belirtmektedir. subTitle property'si ise başlık yazısının altında görüntülenecek yazıyı belirtmektedir. Nihayet MKPointAnnotation sınıfının coordinate isimli property'si işaretçinin konulacağı koordinatı

belirtmektedir. Örneğin:

```
let annotation = MKPointAnnotation()
annotation.title = "Dernek"
annotation.subtitle = "C ve Sistem Programcılar Derneği"
annotation.coordinate = location
```

3) Oluşturulan bu annotation nesnesi MKMapView sınıfının addAnnotation(\_:) metodu ile view penceresine iliştiir. Örneğin:

```
import UIKit
import MapKit

class ViewController: UIViewController {

 @IBOutlet weak var mapView: MKMapView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let location = CLLocationCoordinate2D(latitude: 41.068205, longitude: 29.000837)
 let span = MKCoordinateSpan(latitudeDelta: 0.001, longitudeDelta: 0.001)
 let region = MKCoordinateRegion(center: location, span: span)
 mapView.setRegion(region, animated: true)

 let annotation = MKPointAnnotation()
 annotation.title = "Dernek"
 annotation.subtitle = "C ve Sistem Programcılar Derneği"
 annotation.coordinate = location
 mapView.addAnnotation(annotation)
 }
}
```

Bir annotation nesnesini yine istersek MKMapView sınıfının removeAnnotation(\_:) metodu ile silebiliriz. Örneğin tipik olarak belli bir annotation'ın üzerine tıklandığında onun otomatik silinmesi gerekebilir. İşte bu tür annotation olayları için MKMapViewDelegate protokolü kullanılmaktadır. Bu protokolün çeşitli metotları belli olaylar olduğunda belli metodların çağrılmasını sağlamaktadır. Örneğin bu protokolün mapView(\_:didSelect:) metodu bir annotation seçildiğinde çağrılır. Biz de bu annotation'ı aşağıdaki gibi silebiliriz:

```
func mapView(_ mapView: MKMapView, didSelect view: MKAnnotationView)
{
 mapView.removeAnnotation(view.annotation!)
}
```

Siz de MKMapViewDelegate protokolünün diğer önemli metotlarını dokümanlardan gözden geçiriniz.

Harita üzerinde bir noktaya tıklandığında o nokta hakkında bilgi almak ya da bu bilgiyi çeşitli biçimlerde kullanmak ya da o noktaya bir işaretçi koymak gerekebilir. İşte MKMapView penceresi üzerinde parmakla dokunup belli bir süre beklenliğinde bizim istediğimiz bir metodun çağrılması sağlanabilmektedir. Bu işlem sırasıyla şu adımlardan geçirilerek gerçekleştiriliyor:

1) Öncelikle parmağımızla basıp parmağımızı bir süre beklettiğimizde bir event elde edebilmek için UILongPressGestureRecognizer sınıfı türünden bir nesneyi init(target:action) metoduyla yaratmak gereklidir. Buradaki target View pencerelerinde de gördüğümüz gibi metodu çağrılacak sınıfın nesneyi belirtir ve bu nesne genelliksel

`self` biçiminde geçilmektedir. `action` ise bu sınıfta bir metot belirmektedir. Bu metodun `#selector(...)` sentaksı ile verilmesi ve metodun `@objc` ile özniteliklendirilmesi gerekmektedir.

```
let gestureRecognizer = UILongPressGestureRecognizer(target: self, action:
#selector(longPress(gestureRecognizer)))
//...

@objc
func longPress(gestureRecognizer: UIGestureRecognizer)
{
 //...
}
```

2) Oluşturulan bu `UILongPressGestureRecognizer` sınıfının `minimumPressDuration` isimli property elemanı saniye cinsinden basım için bekletecek zaman aralığını belirtmektedir. Bu property'nin default değeri 0.5'tir. Örneğin:

```
let gestureRecognizer = UILongPressGestureRecognizer(target: self, action:
#selector(longPress(gestureRecognizer)))

gestureRecognizer.minimumPressDuration = 1
```

3) Artık oluşturulan bu nesne `MKMapView` sınıfının `addGestureRecognizer(_:)` metodu ile `MKMapView` penceresine ilişirilir. Örneğin:

```
import UIKit
import MapKit

class ViewController: UIViewController {

 @IBOutlet weak var mapView: MKMapView!

 override func viewDidLoad()
{
 super.viewDidLoad()

 let location = CLLocationCoordinate2D(latitude: 41.068205, longitude: 29.000837)
 let span = MKCoordinateSpan(latitudeDelta: 0.001, longitudeDelta: 0.001)
 let region = MKCoordinateRegion(center: location, span: span)
 mapView.setRegion(region, animated: true)

 let annotation = MKPointAnnotation()
 annotation.title = "Dernek"
 annotation.subtitle = "C ve Sistem Programcılar Derneği"
 annotation.coordinate = location
 mapView.addAnnotation(annotation)

 let gestureRecognizer = UILongPressGestureRecognizer(target: self, action:
#selector(longPress(gestureRecognizer)))

 gestureRecognizer.minimumPressDuration = 1
 mapView.addGestureRecognizer(gestureRecognizer)
 }

 @objc
 func longPress(gestureRecognizer: UIGestureRecognizer)
{
 //...
 }
```

```
}
```

Pekiyi oluşturduğumuz bu action metodu içerisinde neler yapabiliriz? İşte burada metodun parametresi olan UIGestureRecognizer sınıfının içerisinde basılan yerin harita üzerindeki koordinat bilgisi (enlem boylam olarak) alınabilir. Bunun için önce UIGestureRecognizer sınıfının location(in:) metodu çağrılmalıdır. Ancak bu metot bize basılan yerin haritadaki konumunu vermez. Pencere içerisindeki pixel koordinatını vermektedir. İşte bizim de bu pixel koordinatını MKMapView sınıfının convert(\_:toCoordinateFrom:) metodu ile harita koordinatına çevirmemiz gereklidir. Artık bundan sonra biz yukarıda yaptığımız gibi bir MKPointAnnotation nesnesi yaratıp o noktaya bir işaretçi yerleştirebiliriz. Örneğin:

```
import UIKit
import MapKit

class ViewController: UIViewController {

 @IBOutlet weak var mapView: MKMapView!

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let location = CLLocationCoordinate2D(latitude: 41.068205, longitude: 29.000837)
 let span = MKCoordinateSpan(latitudeDelta: 0.001, longitudeDelta: 0.001)
 let region = MKCoordinateRegion(center: location, span: span)
 mapView.setRegion(region, animated: true)

 let annotation = MKPointAnnotation()
 annotation.title = "Dernek"
 annotation.subtitle = "C ve Sistem Programcılar Derneği"
 annotation.coordinate = location
 mapView.addAnnotation(annotation)

 let gestureRecognizer = UILongPressGestureRecognizer(target: self, action:
#selector(longPress(gestureRecognizer)))

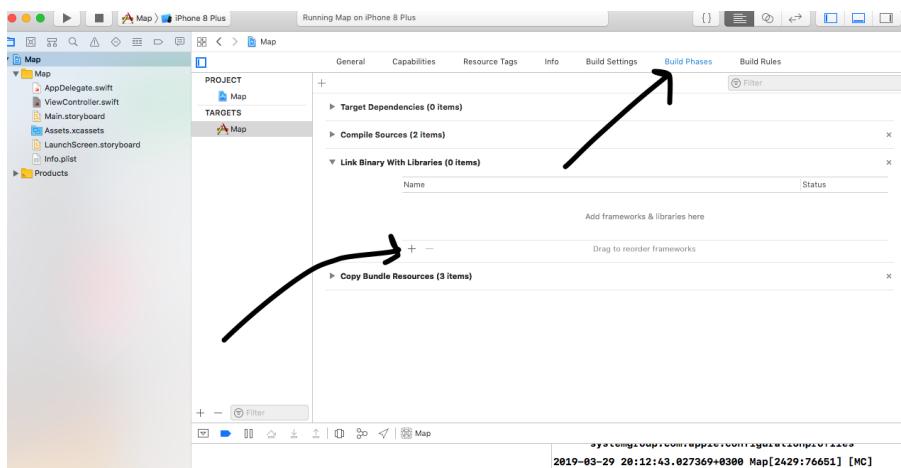
 gestureRecognizer.minimumPressDuration = 1
 mapView.addGestureRecognizer(gestureRecognizer)
 }

 @objc
 func longPress(gestureRecognizer: UIGestureRecognizer)
 {
 let point = gestureRecognizer.location(in: mapView)
 let location = mapView.convert(point, toCoordinateFrom: self.mapView)

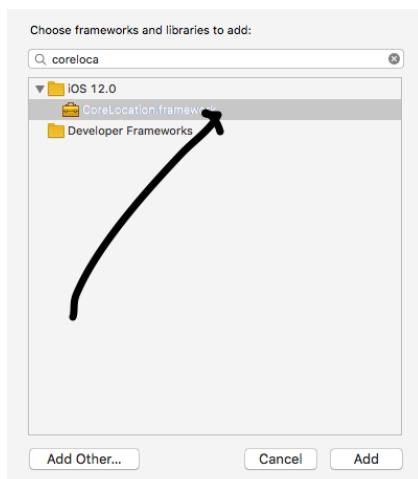
 let annotation = MKPointAnnotation()
 annotation.title = "Way Point"
 annotation.subtitle = "\(location.latitude), \(location.longitude)"
 annotation.coordinate = location
 mapView.addAnnotation(annotation)
 }
}
```

MKMapView üzerinde yer takibi yapmak gerekmektedir. Yani GPS ya da GSM aktif olduğunda harita buradan alınan bilgileri bize verebilmektedir. Böylece biz de haritayı dinamik biçimde görüntüleyebiliriz. Bunu sağlamak için

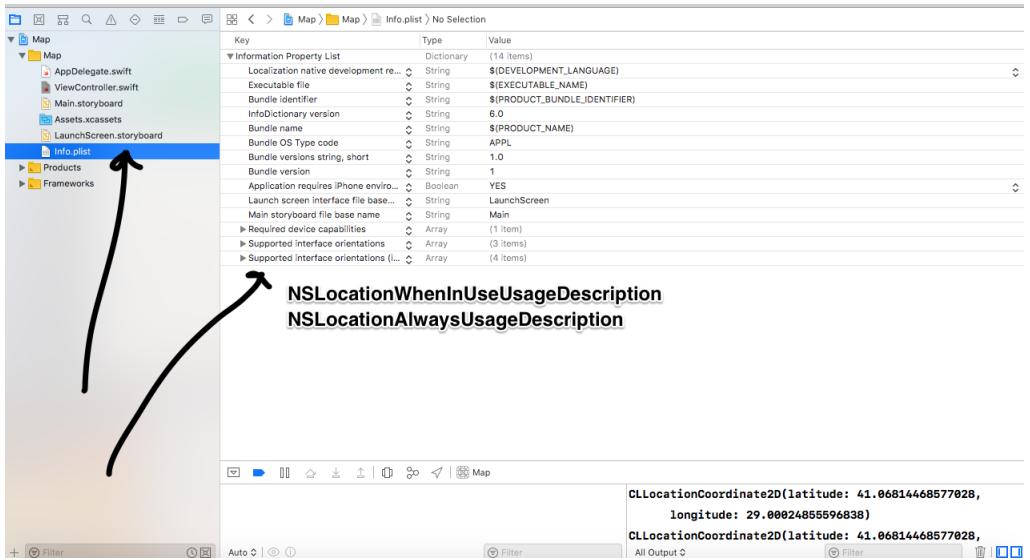
CoreLocation modülü kullanılmaktadır. Ancak bu modül default olarak link aşamasına dahil edilmemektedir. Bu nedenle programcının önce bu modülü link aşamasına dahil etmesi gereklidir. Bu işlem Proje menüsünde "Build Phases" sekmesi kullanılarak yapılmaktadır.



Bu sekmeden CoreLocation.framework eklenir.



Bu işlemden sonra ayrıca kod içerisinde yine bizim CoreLocation modülünü import etmemiz gereklidir. Harita üzerinde konum alan uygulamalarda güvenlik nedeniyle programcının bu isteği projenin plist dosyasında gerekçesiyle oluşturulması gerekmektedir.



Bu info.plist konfigürasyon dosyasına iki anahtarın girilmesi gerekmektedir. Bunlar NSLocationWhenInUseUsageDescription ve NSLocationAlwaysUsagaDescription anahtarlarıdır. Bu anahtara karşı gelen değerler String türünden gerekçe belirten yazılar olmalıdır. Bu yazılar kullanıcıya pop penceresinde izin istemek amacıyla çıkartılamaktadır.

Sonraki işlemler adım adım şöyle yapılmaktadır:

1) Bizim Controller sınıfımızın CLLocationManagerDelegate isimli protokolü desteklemesi gerekmektedir. Örneğin:

```
import UIKit
import MapKit
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate
{
 //...
}
```

2) Bizim bu işlemler için bir tane CLLocationManager türünden bir nesneyi sınıfın init() metodu ile yaratmamız gereklidir. Ancak bu nesnenin referansının yerel bir değişkende değil de sınıfın bir örnek öznirteliği olarak bulundurulması uygundur. Örneğin:

```
class ViewController: UIViewController, CLLocationManagerDelegate {

 @IBOutlet weak var mapView: MKMapView!
 let locationManager = CLLocationManager()
 //...
}
```

3) CLLocationManager sınıfının delegate property'sine CLLocationManagerDelegate protokolünü destekleyen nesne atanmalıdır. Örneğimizde bu nesne self nesnesidir. Bu işleminden sonra bir duyarlılık belirlemesi yapılmalıdır. Bu sınıfın desiredAccuracy isimli property elemanına atama yapılarak gerçekleştirilecektir. Atama yapılacak bazı değerler zaten önceden bildirilmiş biçimde bulunmaktadır. Örneğin tipik olarak bu property'ye kCLLocationAccuracyBest değeri atanabilir.

4) Property'ler set edildikten sonra artık yetkilendirme isteği için sınıfın requestWhenInUseAuthorization() metodu

çağrılmalıdır. Nihayet en sonunda sınıfın startUpdatingLocation() metodu ile izleme başlatılır.

```
locationManager.delegate = self
locationManager.desiredAccuracy = kCLLocationAccuracyBest
locationManager.requestWhenInUseAuthorization()
locationManager.startUpdatingLocation()
```

5) Şimdi izleme süreci başladığını göre bizim çağrılmak üzere bir methodu bildirmemiz gereklidir. İşte CLLocationManagerDelegate protokolünün çeşitli metodları vardır. Biz tipik olarak protokolün locationManager(\_:didUpdateLocations:) isimli metodunu bildiririz.

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])
{
 //...
}
```

Burada didUpdateLocations etiketli parametrenin [CLLocation] biçiminde bir dizi olduğuna dikkat ediniz. Normal olarak bu dizide tek bir elemanın bulunur. Ancak sistem yoğunsa kaçırlanan bazı location'lar da diziye kodlanarak tek bir metod çağrıması yapılabilmektedir.

CLLocation sınıfının coordinate isimli property elemanı CLLocationCoordinate2D sınıfı türündendir. Bu property hareket halindeyken bize o noktanın koordinatlarını verir. Böylece harita üzerinde canlı takip yapılmaktadır. Örneğin:

```
import UIKit
import MapKit
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate {

 @IBOutlet weak var coordinateLabel: UILabel!
 @IBOutlet weak var mapView: MKMapView!
 let locationManager = CLLocationManager()

 override func viewDidLoad()
 {
 super.viewDidLoad()

 let location = CLLocationCoordinate2D(latitude: 41.068205, longitude: 29.000837)
 let span = MKCoordinateSpan(latitudeDelta: 0.001, longitudeDelta: 0.001)
 let region = MKCoordinateRegion(center: location, span: span)
 mapView.setRegion(region, animated: true)

 let annotation = MKPointAnnotation()
 annotation.title = "Dernek"
 annotation.subtitle = "C ve Sistem Programcılar Derneği"
 annotation.coordinate = location
 mapView.addAnnotation(annotation)

 locationManager.delegate = self
 locationManager.desiredAccuracy = kCLLocationAccuracyBest
 locationManager.requestWhenInUseAuthorization()
 locationManager.startUpdatingLocation()
 }

 func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])
 {
 let location = locations[0]
```

```

 let span = MKCoordinateSpan(latitudeDelta: 0.001, longitudeDelta: 0.001)
 let region = MKCoordinateRegion(center: location.coordinate, span: span)
 mapView.setRegion(region, animated: true)
 coordinateLabel.text = "\(location.coordinate.latitude)\n\(location.coordinate.longitude)"
 }
}

```

## Fotoğraf Kütüphanesinin (Photo Library) Kullanılması

IOS aygıtlarının kendi içerisinde built-in bir fotoğraf kütüphanesi vardır. IOS'un fotoğraf uygulaması daaslındabu kütüphaneyi kullanmaktadır. Bu fotoğraf kütüphanesi UIImagePickerController isimli bir diyalog penceresiyle kolay bir biçimde kullanılmaktadır. Kullanım sırasıyla şu aşamalardan geçilerek yapılır.

1) Öncelikle programcı UIImagePickerController sınıfı türünden bir nesne yaratır. Bu sınıf ile ilgili deşikenin sınıfın örnek özniteliği yapılması uygun olur. Örneğin:

```

import UIKit

class ViewController: UIViewController {

 let ipc = UIImagePickerController()

 override func viewDidLoad()
 {
 super.viewDidLoad()
 // Do any additional setup after loading the view, typically from a nib.
 }
}

```

2) Programcı UIImagePickerController nesnesinin bazı property'lerini uygun biçimde set etmek zorundadır. Sınıfın sourceType isimli property elemanı UIImagePickerController.SourceType isimli enum türündendir. Bu eleman açılacak pencerenin hangi fotoğrafları gösterecek biçimde konfigüre edileceğini belirtmektedir. Bu property'ye atanacak değerler şunlardır:

photoLibrary: Tipik kullanım bu biçimdededir. Tüm fotoğraf kütüphanesi görüntülenir.

camera: Burada doğrudan ön ya da arka kamera tarafından çekilmiş olan görüntüler görüntülenir.

savedPhotosAlbum: Burada saklanan fotoğraflar görüntülenir.

UIImagePickerController sınıfının delegate isimli property'si ise UIImagePickerControllerDelegate ve UINavigationControllerDelegate protokollerini desteklemek zorundadır. Bazı olaylar sonucunda bu protokollerin metodları çağrılmaktadır. Örneğin:

```

import UIKit

class ViewController: UIViewController, UIImagePickerControllerDelegate,
UINavigationControllerDelegate {

 let ipc = UIImagePickerController()

 override func viewDidLoad()
 {
 super.viewDidLoad()
 }
}

```

```

@IBAction func choosePhotoButtonClicked(_ sender: Any)
{
 ipc.sourceType = .photoLibrary
 ipc.delegate = self
 //...
}

```

3) Artık UIViewController sınıfının present metodu ile ilgili view penceresi bir diyalog penceresi gibi görüntülenir. Örneğin:

```

@IBAction func choosePhotoButtonClicked(_ sender: Any)
{
 ipc.sourceType = .photoLibrary
 ipc.delegate = self
 self.present(ipc, animated: true, completion: nil)
}

```

4) Artık bir fotoğraf seçildiğinde uygun bir işlemi yapmak isteriz. Bunun için UIImagePickerControllerDelegate protokolünün imagePickerController(\_:didFinish:) metodu bildirilmelidir. Bu metod fotoğraf seçildiğinde çağrılır. Tabii bu metod protokolün optional metodu olduğu için hiç yazılmayabilir. Bu durumda fotoğraf seçildiğinde otomatik olarak diyalog penceresi kapatılır. Benzer biçimde aynı protokolün imagePickerControllerDidCancel(\_) metodu diyalog penceresinden cancel tuşu ile çıkışlığında çağrılmaktadır. Örneğin:

```

func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any])
{
 print("Selected")
}

func imagePickerControllerDidCancel(_ picker: UIImagePickerController)
{
 print("canceled")
}

```

Bu metodlar sınıfa eklendiğinde artık diyalog penceresi otomatik kapatılmamaktadır. Onu kapatmak artık programcının görevidir. Anımsanacağı gibi bu işlem için UIViewController sınıfının dismiss metodu kullanılmaktadır. Örneğin:

```

func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any])
{
 print("Selected")
 self.dismiss(animated: true, completion: nil)
}

```

Tabii aslında bir resim seçildiğinden seçilen resmin alınıp kullanılması gereklidir. İşte bu metod bize seçilen resmi de vermektedir. İşte metodun info isimli parametresi bir sözlük türündendir. Bu sözlüğün anahtarı UIImagePickerController.InfoKey isimli yapı türünden değeri de anahtara göre değişebilmektedir. Örneğin yapının originalImage property'si anahtar olarak verilirse değer olarak UIImage nesnesi elde edilmektedir. Eğer anahtar olarak imageURL verilirse değer olarak URL nesnesi elde edilmektedir. Bu yapının diğer başka elemanları da vardır.