

# App-Entwicklung für iOS und OS X

SS 2016

Stephan Gimbel



# NSUserDefaults

Alle iOS Apps besitzen ein "eingebautes" Dictionary um User-spezifische Informationen zu speichern. Diese können nach dem Beenden der App und erneutem Start wieder verwendet werden, bleiben also erhalten.

Gespeichert werden können z.B. Integers, Booleans, Strings, Arrays, Dictionaries, Dates und mehr.

Dies ersetzt keine Datenbank und sollte nicht missbraucht werden um umfangreiche Datensammlungen zu speichern!

```
let defaults = NSUserDefaults.standardUserDefaults()
defaults.setInteger(55, forKey: "Age")
defaults.setBool(true, forKey: "UseTouchID")
defaults.setDouble(M_PI, forKey: "Pi")

defaults.setObject("Tim Cook", forKey: "Name")
defaults.setObject(NSDate(), forKey: "LastRun")
```

# NSUserDefaults

... auch für Arrays und Dictionaries

```
let array = ["Hello", "World"]
defaults.setObject(array, forKey: "SavedArray")

let dict = ["Name": "Tim", "Country": "US"]
defaults.setObject(dict, forKey: "SavedDict")
```

Wenn beim Auslesen ein Setting nicht gefunden werden kann, dann gibt NSUserDefaults einen Default-Wert zurück.

- integerForKey() Integer wenn Key existiert, sonst 0
- boolForKey() Boolean wenn Key existiert, sonst false
- floatForKey() Float wenn Key existiert, sonst 0.0
- doubleForKey() Double wenn Key existiert, sonst 0.0
- objectForKey() AnyObject?, benötigt conditionally typecast zum erwarteten Datentyp

# NSUserDefaults

Das Auslesen geschieht wie folgt:

```
let defaults = NSUserDefaults.standardUserDefaults()

let age = defaults.integerForKey("Age")
let useTouchID = defaults.boolForKey("UseTouchID")
let pi = defaults.doubleForKey("Pi")
```

Da der Empfang von Objekten immer ein Optional liefert, kann dies entweder so weiter verwendet werden oder es muss ein Typecast in einen non-optional Typ mittels des nil coalescing Operators erfolgen, z.B.:

```
let array = defaults.objectForKey("SavedArray") as? [String] ?? [String]()
```

# Views

---

Ein View (z.B. **UIView** Subclass) ist eine rechteckige Fläche, die

- ein Koordinatensystem definiert
- zum zeichnen (drawing)
- und Touch Events behandelt

Views haben eine **Hierarchie**, wobei jeder View einen Superview hat...

```
var superview: UIView? { get }
```

Views können mehrere (oder keine) Subviews haben

```
var subviews: [UIView] { get }
```

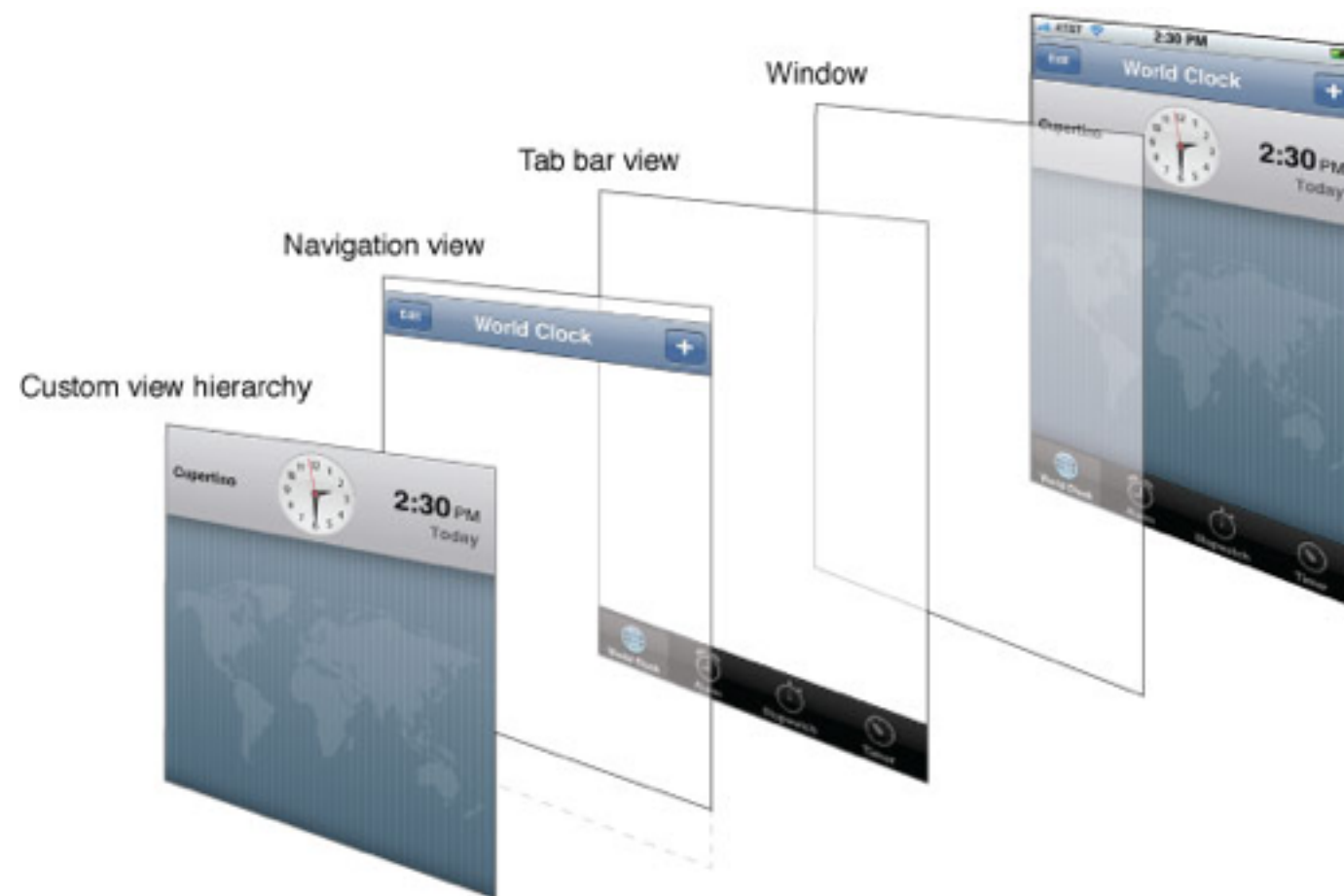
Die Reihenfolge im Subviews-Array ist wichtig! Die Views die später im Array angeordnet sind, liegen über denen die früher im Array sind.

Ein View kann seine Subviews in seine eigenen Bounds clippen (default ist dies nicht zu tun)

# Views

**UIWindow** ist der **UIView**, der ganz, ganz oben in der View-Hierarchie liegt (inkl. Status Bar).

Normalerweise existiert nur ein UIWindow in einer iOS Applikation\*, es geht also i.d.R. immer um UIViews und nicht UIWindows.



# Views

---

Die View-Hierarchie wird normalerweise graphisch in Xcode erstellt. Dies gilt auch für Custom Views.

Dies kann jedoch auch in Code geschehen:

```
addSubview(aView: UIView)
```

```
removeFromSuperview() // geschickt an den View, nicht dessen Superview!
```

Wo beginnt die View Hierarchie?

Die (nutzbare) View Hierarchie beginnt im `var view:UIView` des Controllers.

Es ist wichtig dieses Property zu verstehen! Dieser View ändert seine Bounds, wenn z.B. eine Rotation erfolgt.

Es ist wahrscheinlich der View, zu dem Sie Subviews in Code hinzufügen (sofern Sie das jemals machen).

Alle Views aus dem MVC haben diesen View als Vorfahren.

Geschieht automatisch, wenn der MVC in Xcode erstellt wird.

# Views

---

Wie immer gilt, versuchen Sie Initializer zu vermeiden (sofern möglich)  
Es ist allerdings häufiger der Fall als z.B. einen UIViewController Initializer zu haben.

Der UIView Initializer unterscheidet sich, wenn er aus einem Storyboard kommt.

`init(frame: CGRect)` // Erstellt in Code

`init(coder: NSCoder)` // aus dem Storyboard

Sofern Sie einen Initializer benötigen, erstellen Sie beide!

```
override init(frame: CGRect) {  
    super.init(frame: frame)  
    setup()  
}  
  
required init(coder aDecoder: NSCoder) {  
    super.init(coder: aDecoder)  
    setup()  
}
```



# Views

Wie immer gilt, versuchen Sie Initializer zu vermeiden (sofern möglich)  
Es ist allerdings häufiger der Fall als z.B. einen UIViewController Initializer zu haben.

Der UIView Initializer unterscheidet sich, wenn er aus einem Storyboard kommt.

`init(frame: CGRect)` // Erstellt in Code

`init(coder: NSCoder)` // aus dem Storyboard

Sofern Sie einen Initializer benötigen, erstellen Sie beide!

```
override init(frame: CGRect) {  
    super.init(frame: frame)  
    setup()  
}
```

```
required init(coder aDecoder: NSCoder) {  
    super.init(coder: aDecoder)  
    setup()  
}
```

Jede Subclass dieser Klasse muss diesen Initializer implementieren.

# Views

---

Eine Alternative zu Initializers von UIViews ist...

`awakeFromNib()`

Wird nur aufgerufen wenn der View aus einen Storyboard kommt.

Es ist kein Initializer (!), wird aber unmittelbar aufgerufen nachdem die Initialisierung abgeschlossen ist.

Alle Objekte die in einem Storyboard von NSObject erben, rufen dies auf (sofern sie es implementieren).

# Views

---

## Datenstrukturen für das Koordinatensystem

### CGFloat

Immer CGFloat statt Double oder Float verwenden, für alles was mit dem Koordinatensystem eines UIViews zu tun hat.

Von Double oder Float kann konvertiert werden

```
let cfg = CGFloat(aDouble)
```

### CGPoint

Einfaches Struct mit zwei CGFloats: x und y.

```
var point = CGPoint(x: 37.0, y: 55.2)
```

```
point.x -= 30
```

```
point.y += 20.0
```

### CGSize

Ebenfalls ein Struct mit zwei CGFloats: width and height.

```
var size = CGSize(width: 100.0, height: 50.0)
```

```
size.width += 42.5
```

```
size.height += 75
```

# Views

---

## CGRect

Ein Struct mit einem CGPoint und einer CGSize

```
struct CGRect {  
    var origin: CGPoint  
    var size: CGSize  
}
```

```
let rect = CGRect(origin: aCGPoint, size: aCGSize)
```

Es existieren weitere Initializer...

... und weitere Convenient Properties und Funktionen für CGRect wie...

```
var minX: CGFloat          // linke Kante  
var midY: CGFloat          // Mittelpunkt vertikal  
intersects(CGRect) -> Bool // schneidet dieses CGRect ein anderes?  
intersect(CGRect)         // clip das CGRect zur Überschneidung  
contains(CGPoint) -> Bool  // liegt CGPoint im CGRect?
```

... und viele, viele mehr (siehe Code-Completion)

# Views

(0,0)

## Koordinatensystem

Ursprung ist oben-links

Einheiten sind Points, nicht Pixel

Pixel sind die kleinste Einheit zum zeichnen auf einem Device.

Points sind die Einheiten des Koordinatensystems

Meistens gibt es 2 Pixel pro Point, könnte aber auch 1 oder etwas anders sein.

Wie viele Pixel existieren pro Point?

`var contentScaleFactor: CGFloat` aus **UIView**

In den Boundaries wird gezeichnet

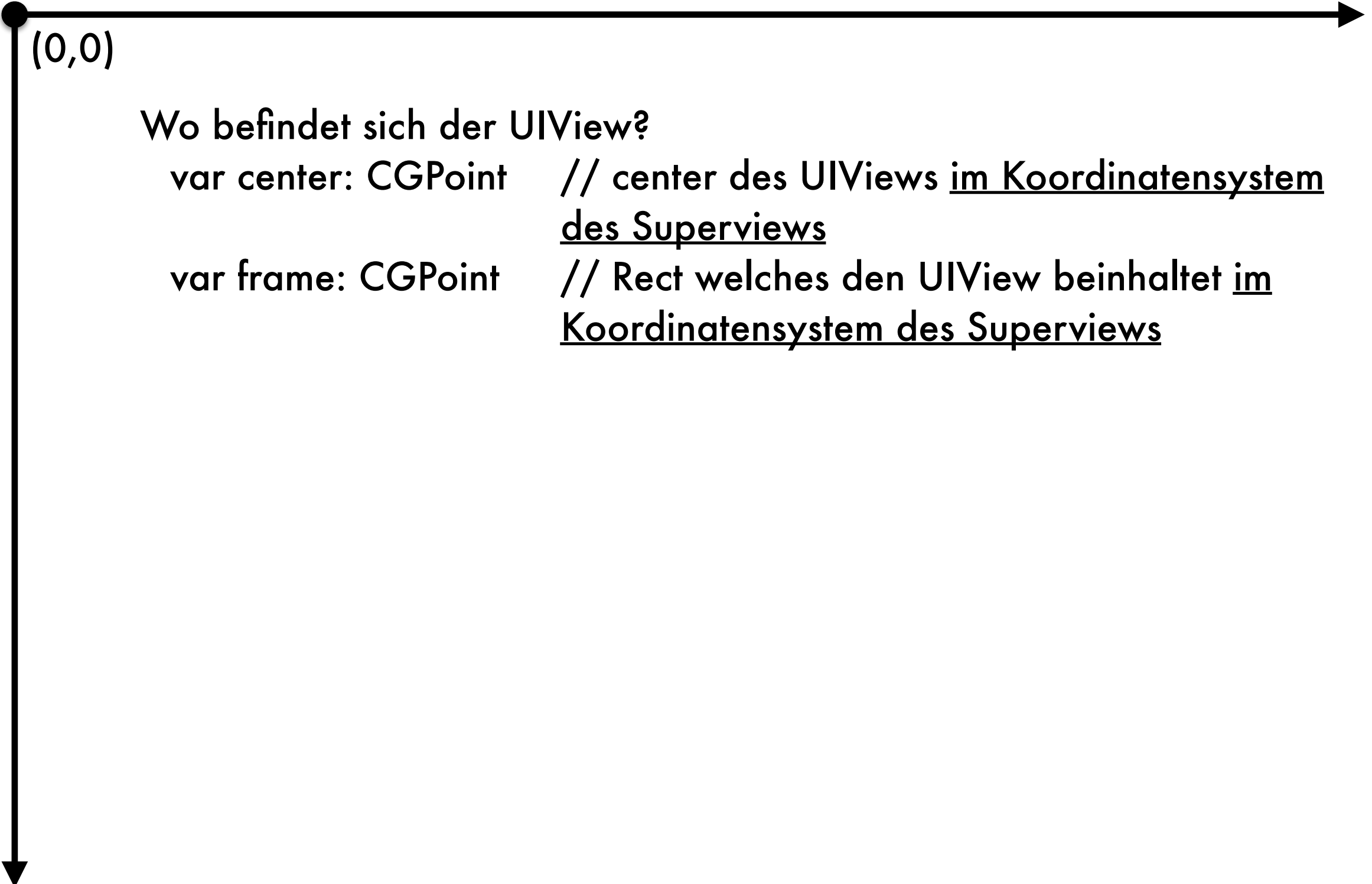
`var bounds: CGRect` // Interne Zeichenfläche mit origin und size

Das Rechteck mit der Zeichenfläche in seinem eigenen Koordinatensystem

Die Implementierung des Views muss implementieren was

`bounds.origin` bedeutet (meistens gar nichts)

# Views



(0,0)

Wo befindet sich der UIView?

var center: CGPoint // center des UIViews im Koordinatensystem  
des Superviews

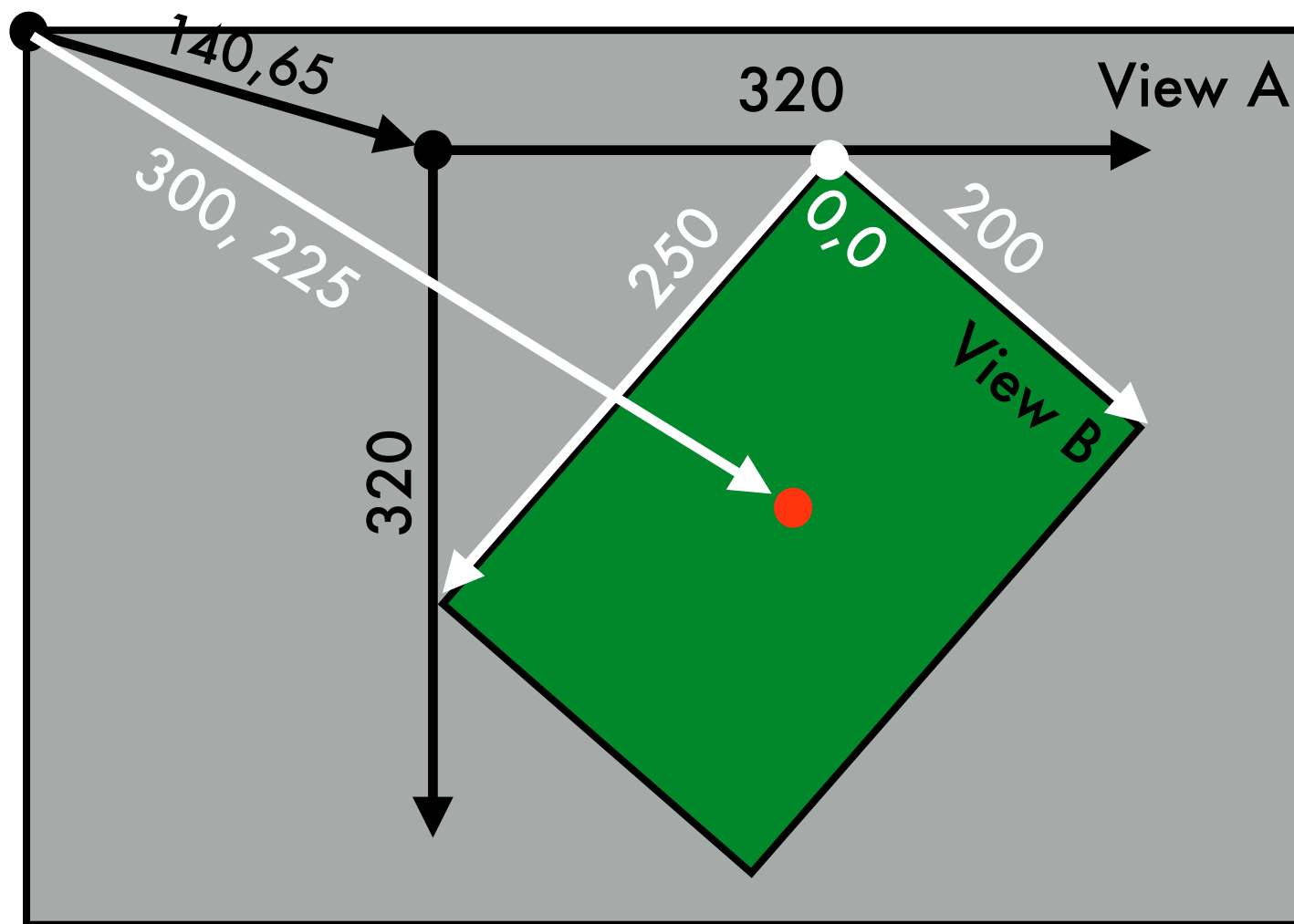
var frame: CGRect // Rect welches den UIView beinhaltet im  
Koordinatensystem des Superviews

# Views

## Bounds vs Frames

Verwenden von **frame** und/oder **center** um einen View zu positionieren.

Dies wird nie verwendet um im Koordinatensystem eines Views zu zeichnen.  
Vielleicht vermuten Sie, dass `frame.size` immer gleich `bounds.size` ist, aber die liegen Sie falsch...



Views können rotiert werden (und skaliert und verschoben)

View B bounds =  $((0,0), (200,250))$

View B frame =  $((140,65), (320,320))$

View B center =  $(300,225)$

Mittelpunkt von View B in seinem eigenen Koordinatensystem ist...

$(\text{bounds.midX}, \text{bounds.midY}) = (100,125)$

Views werden selten rotiert, missbrauchen Sie aber nie `frame` oder `center` unter diesen Annahme!

# Views

Meistens werden Views im Storyboard erstellt

Xcode's Object Library hat einen generischen UIView, der drag&dropped werden kann.

Danach muss der **Identity Inspector** verwendet werden um dessen Klasse auf Ihre Subclass zu setzen.

In sehr seltenen Fällen erstellen Sie UIViews in Code

Unter Verwendung des Frame Initializers

```
let newView = UIView(frame:myViewFrame)
```

Oder einfacher `let newView = UIView()` //Frame ist dann CGRectZero

## Beispiel

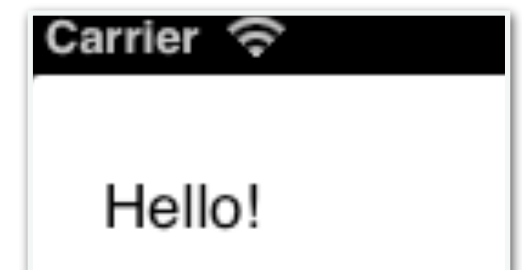
// Code in UIViewController

```
let labelRect = CGRect(x: 20, y:20, width: 100, height: 50)
```

```
let label = UILabel(frame: labelRect) // UILabel ist Subclass von UIView
```

```
label.text = "Hello!"
```

```
view.addSubview(label)
```





# Views

---

## Custom Views

Manchmal ist es sinnvoll eine eigene UIView Subclass zu erstellen.

Wenn frei auf dem Screen gezeichnet werden soll.

Wenn Touch-Events auf eine spezielle Art und Weise behandelt werden sollen (z.B. anders als ein Button oder Slider)

Touch-Events behandeln wir später, zuerst das Zeichnen

Um zu zeichnen, einfach eine UIView Subclass erstellen und drawrect überschreiben.

`override func drawrect(regionThatNeedsToBeDrawn: CGRect)`

Sie können außerhalb von `regionThatNeedsToBeDrawn` zeichnen, dies wird aber nicht verlangt

`regionThatNeedsToBeDrawn` ist nur zur Optimierung

# Views

---

Rufen Sie NIEMALS (!!!) drawRect direkt auf! **NIEMALS!!!**

Wenn Ihr View neu gezeichnet werden soll, lassen Sie es das System wissen durch Aufruf von...

`setNeedsDisplay()`

`setNeedsDisplayInRect(regionThatNeedsToBeRedrawn: CGRect)`

iOS ruft dann zu einem passenden Zeitpunkt drawRect für Sie auf.

# Views

---

Wie wird drawRect implementiert?

Unter Verwendung einer C-ähnlichen (nicht objekt-orientierten) API mit dem Namen Core Graphics

Oder objekt-orientiert mit der **UIBezierPath** Klasse

Core Graphics Konzepte

Sie erhalten einen Kontext in den gezeichnet werden kann (oder gedruckt, oder..., oder...)

Die Funktion `UIGraphicsGetCurrentContext()` gibt den Kontext zurück, den Sie in drawRect verwenden können

Erstellung von Pfaden (Winkel, Bogen, etc.)

Setzen von Attributen wie Farben, Schriftart, Texturen, Linienbreite, etc.

Stroke oder Fill für die erstellten Pfade mit den gegebenen Attributen

# Views

---

## UIBezierPath

Identisch wie Konzept auf letzter Seite, nur dieses mal in einer UIBezierPath Instanz

UIBezierPath zeichnet automatisch im "current" Kontext (drawRect macht das Setup)

Bietet Methoden zum Hinzufügen zum UIBezierPath (lineto, arcs, etc.) und setzen von linewidth, etc.

Methoden für stroke und fill des UIBezierPath

# Views

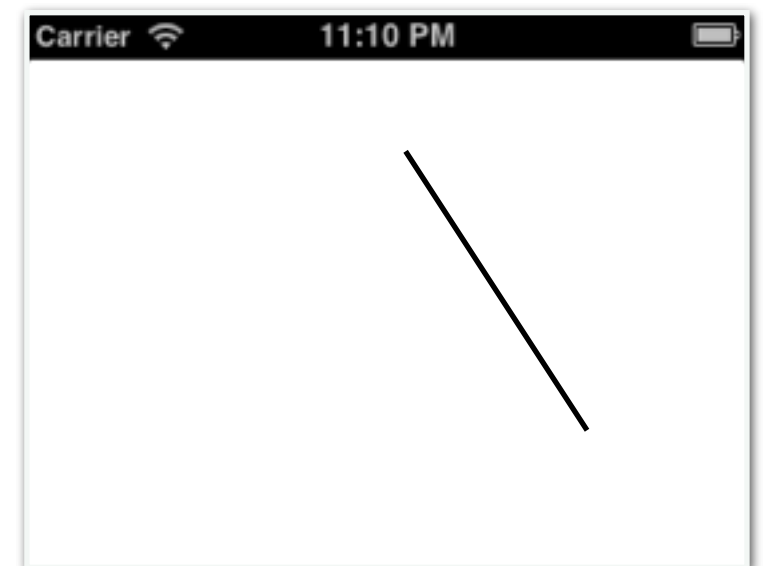
---

## Erstellen eines UIBezierPath

```
let path = UIBezierPath()
```

## Verschieben, Linien oder Arcs zum Pfad hinzufügen

```
path.moveToPoint(CGPoint(80, 50))  
path.addLineToPoint(CGPoint(140, 150))
```



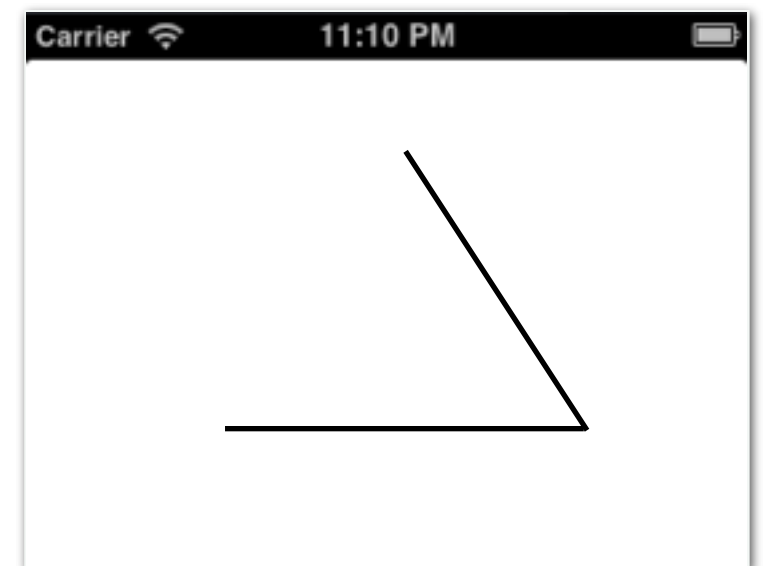
# Views

## Erstellen eines UIBezierPath

```
let path = UIBezierPath()
```

## Verschieben, Linien oder Arcs zum Pfad hinzufügen

```
path.moveToPoint(CGPoint(80, 50))  
path.addLineToPoint(CGPoint(140, 150))  
path.addLineToPoint(CGPoint(10, 150))
```



# Views

## Erstellen eines UIBezierPath

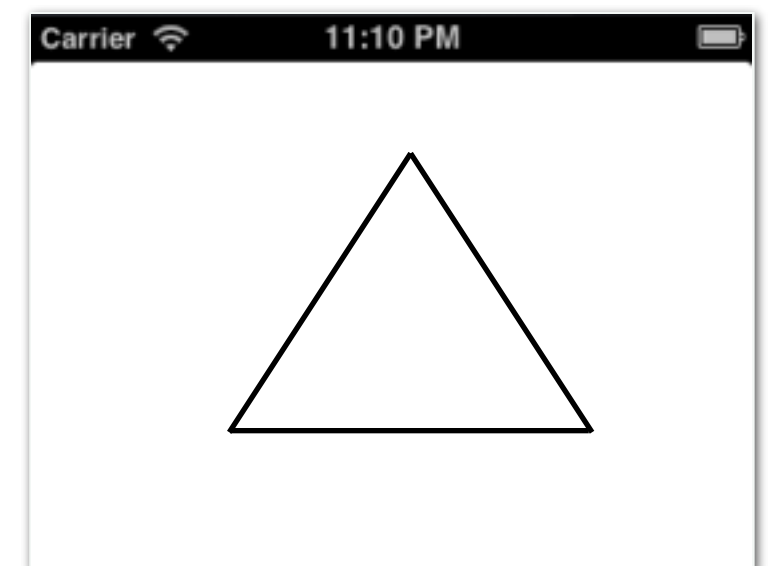
```
let path = UIBezierPath()
```

## Verschieben, Linien oder Arcs zum Pfad hinzufügen

```
path.moveToPoint(CGPoint(80, 50))  
path.addLineToPoint(CGPoint(140, 150))  
path.addLineToPoint(CGPoint(10, 150))
```

## Schließen des Pfades (sofern gewünscht)

```
path.closePath()
```



# Views

## Erstellen eines UIBezierPath

```
let path = UIBezierPath()
```

## Verschieben, Linien oder Arcs zum Pfad hinzufügen

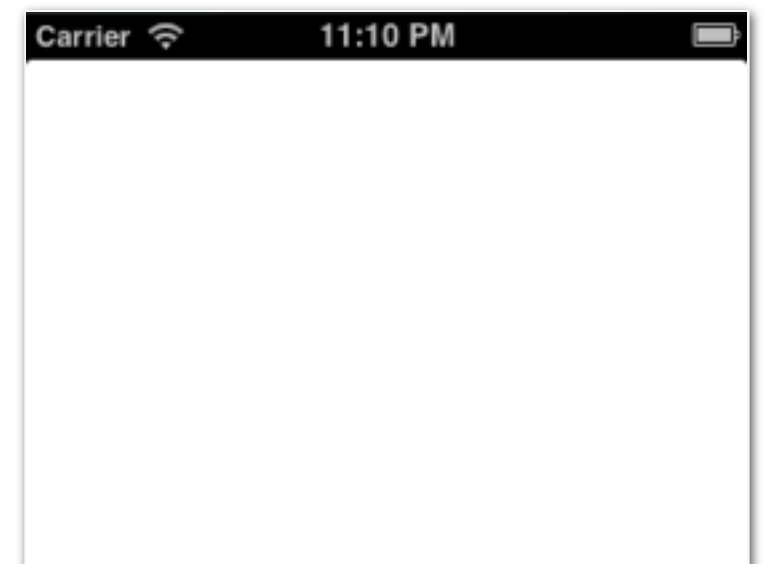
```
path.moveToPoint(CGPoint(80, 50))  
path.addLineToPoint(CGPoint(140, 150))  
path.addLineToPoint(CGPoint(10, 150))
```

## Schließen des Pfades (sofern gewünscht)

```
path.closePath()
```

Bisher ist aber noch nichts passiert... die Attribute für das Zeichnen müssen noch gesetzt werden

```
UIColor.greenColor().setFill() // Methode aus UIColor  
UIColor.redColor().setStroke() // Methode aus UIColor  
path.lineWidth = 3.0 // Property von UIBezierPath
```





# Views

## Erstellen eines UIBezierPath

```
let path = UIBezierPath()
```

## Verschieben, Linien oder Arcs zum Pfad hinzufügen

```
path.moveToPoint(CGPoint(80, 50))  
path.addLineToPoint(CGPoint(140, 150))  
path.addLineToPoint(CGPoint(10, 150))
```

## Schließen des Pfades (sofern gewünscht)

```
path.closePath()
```

Bisher ist aber noch nichts passiert... die Attribute für das Zeichnen müssen noch gesetzt werden

```
UIColor.greenColor().setFill() // Methode aus UIColor  
UIColor.redColor().setStroke() // Methode aus UIColor  
path.lineWidth = 3.0 // Property von UIBezierPath  
path.fill()
```



# Views

## Erstellen eines UIBezierPath

```
let path = UIBezierPath()
```

## Verschieben, Linien oder Arcs zum Pfad hinzufügen

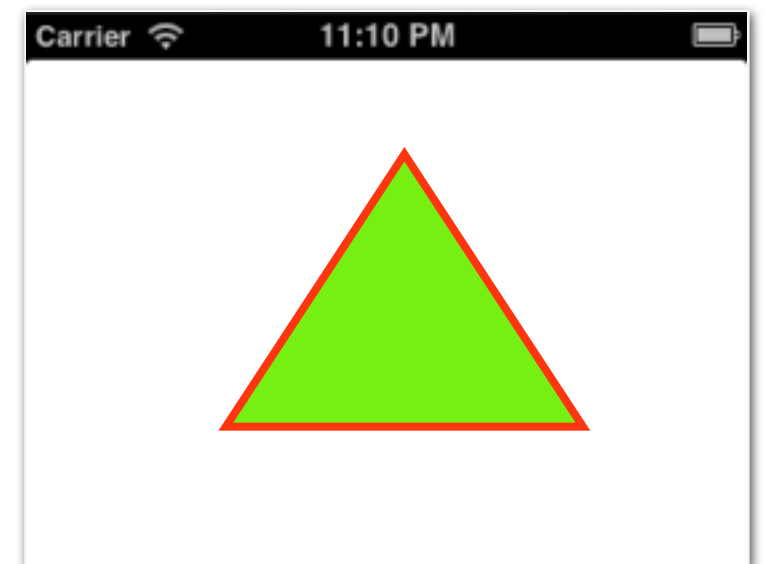
```
path.moveToPoint(CGPoint(80, 50))  
path.addLineToPoint(CGPoint(140, 150))  
path.addLineToPoint(CGPoint(10, 150))
```

## Schließen des Pfades (sofern gewünscht)

```
path.closePath()
```

Bisher ist aber noch nichts passiert... die Attribute für das Zeichnen müssen noch gesetzt werden

```
UIColor.greenColor().setFill() // Methode aus UIColor  
UIColor.redColor().setStroke() // Methode aus UIColor  
path.lineWidth = 3.0 // Property von UIBezierPath  
path.fill()  
path.stroke()
```



# Views

## Viele verschiedene Formen für UIBezierPath

```
let roundRect = UIBezierPath(roundedRect: aCGRect,  
                             cornerRadius: aCGFloat)
```

```
let oval = UIBezierPath(ovalInRect: aCGRect)
```

... und viele andere

## Clipping von Zeichnungen zu einem UIBezierPath

```
addClip()
```

Zum Beispiel, clippen eines gerundeten Rechtecks für die Ecken einer Spielkarte

## Kollisionserkennung

```
func containsPoint(CGPoint) -> Bool
```

Der Pfad muss geschlossen sein. Winding kann mittels `usesEvenOddFillRule` Property gesetzt werden.

Und noch viel, viel mehr. Siehe Doku.

