

App-Entwicklung für iOS und OS X

SS 2017

Stephan Gimbel

Aktualisiert für iOS 10,
Xcode 8 und Swift 3



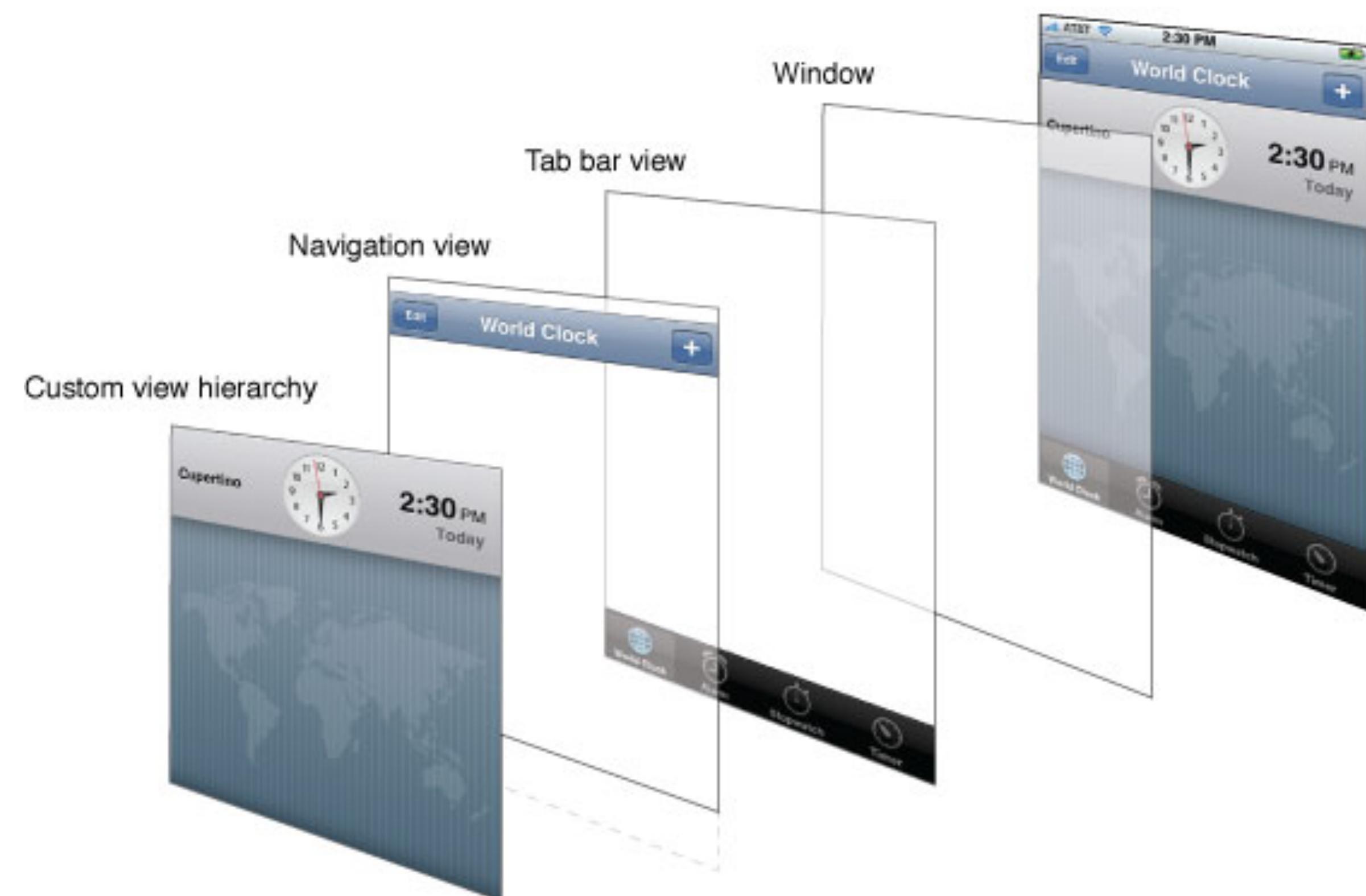
Heute

- **Views**
 - Zeichnen
- **Demo**
 - SmileyView

- Ein View (z.B. UIView Subclass) ist eine rechteckige Fläche, die...
 - ein Koordinatensystem definiert
 - zeichnet (drawing)
 - und Touch Events behandelt
- Hierarchie
 - Ein View hat nur einen `superview` ... `var superview: UIView?`
 - Er kann aber viele (oder keine) `subviews` haben ... `var subviews: [UIView]`
 - Die Reihenfolge im `subviews` Array ist wichtig: die welche später im Array stehen, sind über den früheren angeordnet
 - Ein View kann seine `subviews` zu seinen eigenen `bounds` clippen... oder auch nicht (Default ist dies nicht zu tun)
- UIWindow
 - Ist der UIView ganz, ganz oben in der View Hierarchie (und beinhaltet sogar den Status Bar)
 - Normalerweise haben wir nur ein UIWindow in einer iOS Applikation... es geht nur um Views, nicht um Windows

Views

- **UIWindow**
 - Ist der **UIView** ganz, ganz oben in der View Hierarchy (und beinhaltet sogar den Status Bar)
 - Normalerweise haben wir nur ein **UIWindow** in einer iOS Applikation... es geht nur um Views, nicht um Windows



Views

- Die View-Hierarchie wird normalerweise graphisch in Xcode erstellt
 - Sogar Custom Views werden normalerweise in die View Hierarchie mittels Xcode eingefügt

- Dies kann jedoch auch in Code geschehen:

```
func addSubview(_ view: UIView) // geschickt an den (baldigen) superview des Views  
func removeFromSuperview()      // geschickt an den View, der entfernt werden soll (nicht  
                                dessen Superview)
```

- Wo beginnt die View Hierarchie?

- Die (nutzbare) View Hierarchie beginnt in `var view: UIView` des Controllers.
- Es ist sehr wichtig dieses Property zu verstehen!
- Dieser View ändert seine bounds, wenn z.B. eine Rotation erfolgt.
- Es ist wahrscheinlich der `view`, zu dem wir subviews in Code hinzufügen werden (sofern wir das jemals machen)
- Alle unsere MVC's View's UIViews haben diesen `view` als Vorfahren.
- Wird automatisch für uns verbunden, wenn wir einen MVC in Xcode erstellen

Initialisierung eines UIView

- Wie immer gilt, wir versuchen initializer zu vermeiden (sofern möglich)
 - Es ist allerdings üblicher einen in UIView zu haben, als einen UIViewController Initializer zu haben

- Der Initializer eines UIView ist anders, wenn er aus einem Storyboard kommt

```
init(frame: CGRect) // Initializer, wenn der UIView in Code erstellt wird  
init(coder: NSCoder) // Initializer, wenn der UIView aus einem Storyboard kommt
```

- Wenn wir einen Initializer brauchen, implementieren wir beide ...

```
func setup() { ... }  
  
override init(frame: CGRect) {           // ein designated Initializer  
    super.init(frame: frame)  
    setup()  
}  
  
required init(coder aDecoder: NSCoder) { // ein required Initializer  
    super.init(coder: aDecoder)  
    setup()  
}
```

Initialisierung eines UIView

- Eine weitere Alternative zu Initializers in UIView...

`awakeFromNib()` // dies wird nur aufgerufen, wenn der UIView aus einem Storyboard kam

- Dies ist kein Initializer (es wird aufgerufen direkt nachdem die Initialisierung abgeschlossen ist)
- Alle Objekte, die von NSObject im Storyboard erben empfangen dies
- Die Reihenfolge ist nicht garantiert, also können wir hier keine Nachrichten an andere Objekte im Storyboard schicken

Datenstrukturen des Koordinatensystems

- **CGFloat**

- Dies verwenden wir immer statt Double oder Float, für alles was mit dem Koordinatensystem eines UIViews zu tun hat
- Wir können zu/von Double oder Float konvertieren mittels Initializers, z.B. `let cgf = CGFloat(aDouble)`

- **CGPoint**

- Einfaches struct mit zwei CGFloats darin: 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 darin: width and height

```
var size = CGSize(width: 100.0, height: 50.0)
size.width += 42.5
size.height += 75
```

- **CGRect**

- Ein struct mit einem CGPoint und einer CGSize darin...

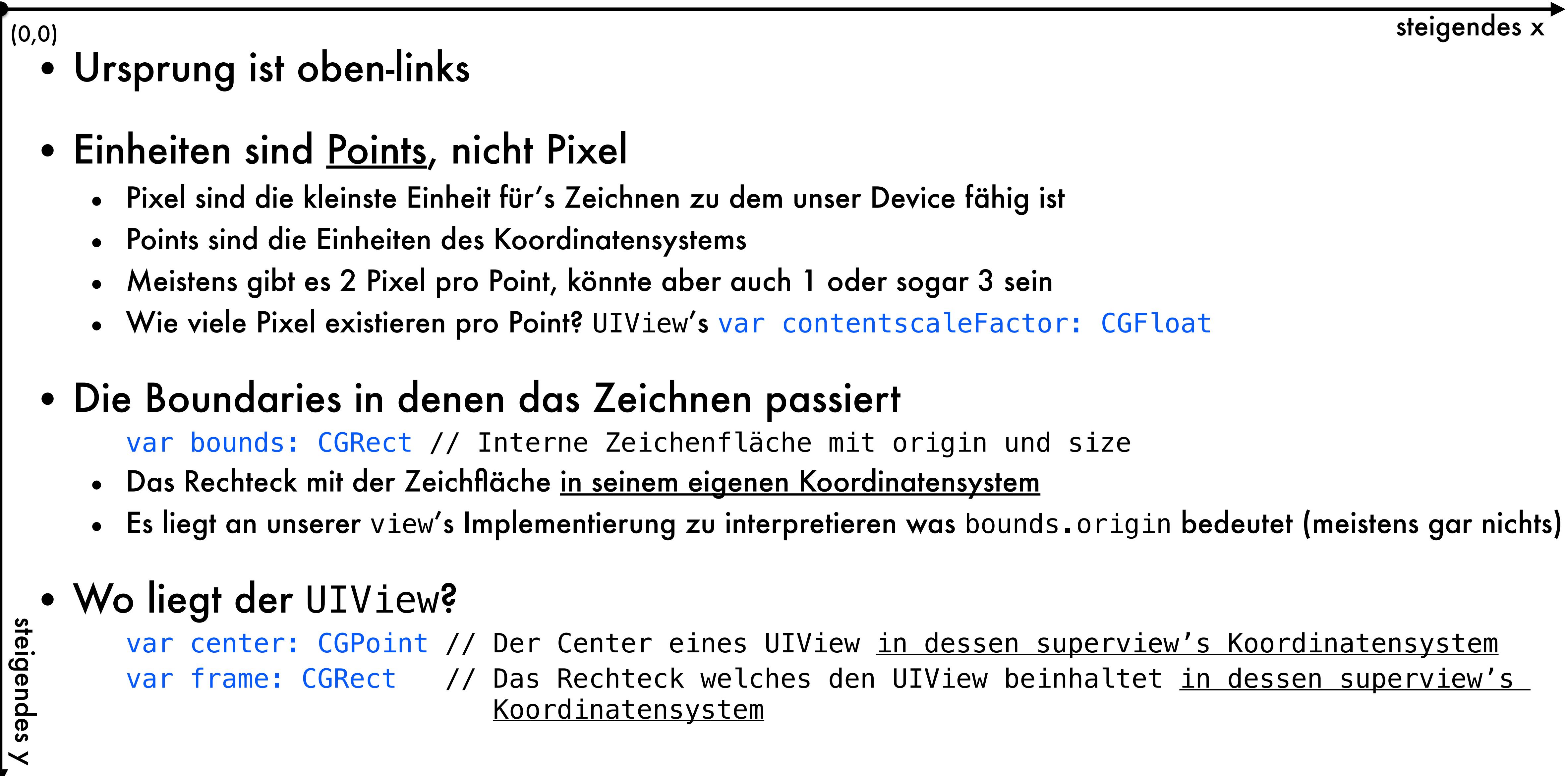
```
struct CGRect {  
    var origin: CGPoint  
    var size: CGSize  
}
```

```
let rect = CGRect(origin: aCGPoint, size: aCGSize) // es existieren weitere inits
```

- Ein CGRect hat viele nützliche Properties und Functions wie...

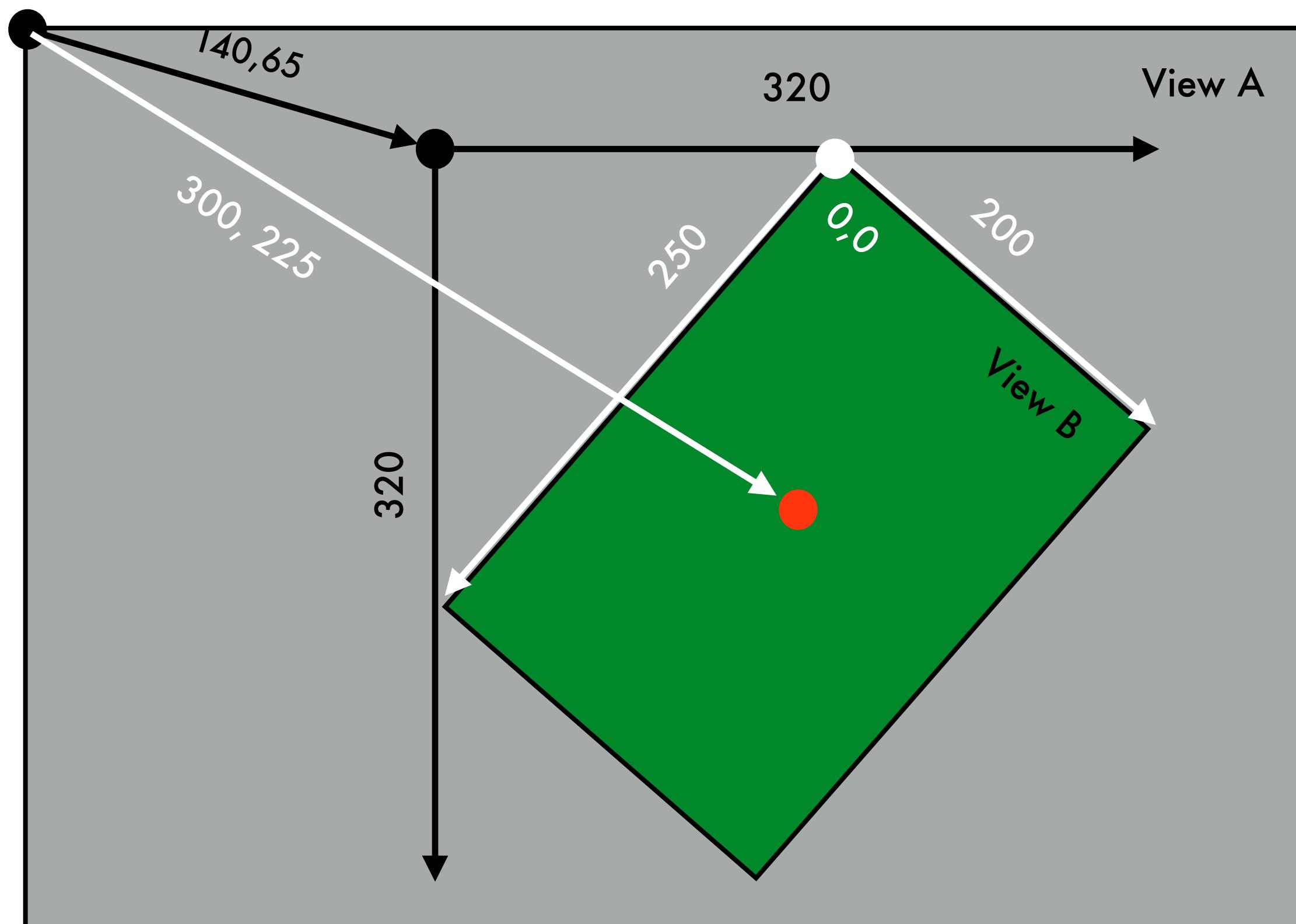
- var minX: CGFloat // linke Kante
- var midY: CGFloat // Mittelpunkt vertikal
- intersects(CGRect) -> Bool // schneidet dieses CGRect ein anderes?
- intersect(CGRect) // clip das CGRect zur Überschneidung
- contains(CGPoint) -> Bool // liegt CGPoint im CGRect?
- ... und viele, viele mehr (einfach ein CGRect erstellen und danach . tippen um mehr zu sehen)

Koordinatensystem des Views



bounds vs frame

- Wir verwenden **frame** und/oder **center** um einen **UIView** zu positionieren
 - Diese werden innerhalb des Koordinatensystems eines Views niemals zum zeichnen verwendet
 - Auch wenn wir denken, dass `frame.size` immer identisch zu `bounds.size` ist, liegen wir falsch...

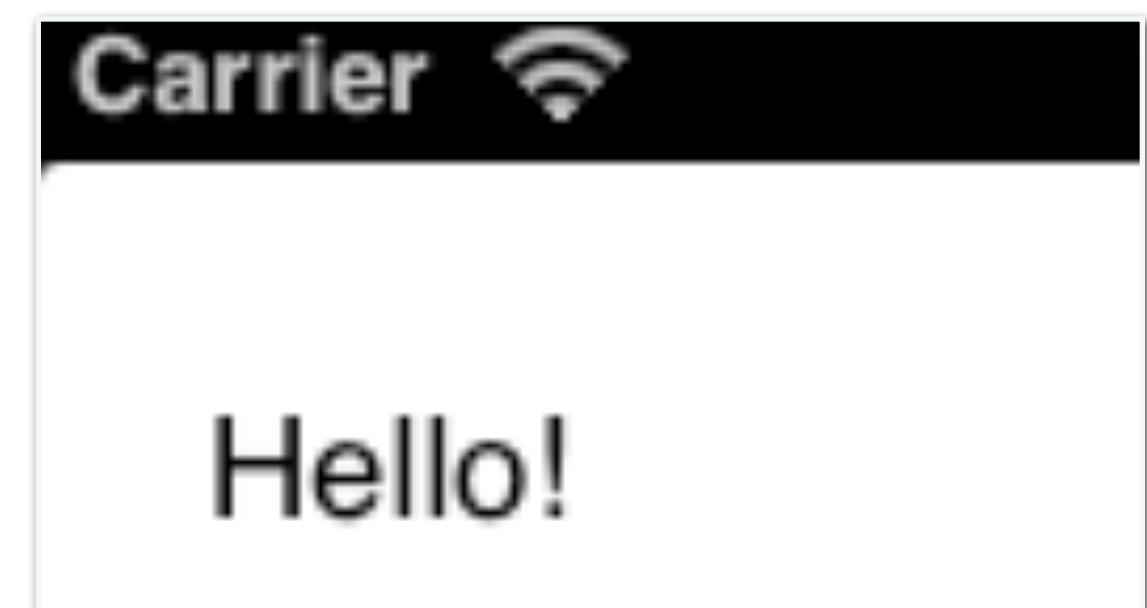


- Views können rotiert werden (und skaliert und verschoben)
View B bounds = ((0,0), (200,250))
View B frame = ((140, 65),(320,320))
View B center = (300,225)
- Mittelpunkt von View B in seinem eigenen Koordinatensystem ist...
(**bounds.midX**, **bounds.midY**) = (100,125)
- Views werden selten rotiert, wir missbrauchen daher nie frame oder center unter dieser Annahme!

Erstellen von Views

- Meistens werden Views im Storyboard erstellt
 - Xcode's Object Library hat einen generischen UIView, der per Drag & Drop verwendet werden kann
 - Danach muss der **Identity Inspector** verwendet werden, um dessen Klasse auf unsere Subclass zu setzen
- In seltenen Fällen erstellen wir einen UIView in Code
 - Wir können dafür den Frame Initializer verwenden... `let newView = UIView(frame: myViewFrame)`
 - Oder wir nutzen einfach `let newView = UIView()` // Frame ist dann CGRect.zero
- Beispiel

```
// wir nehmen an, dass dieser Code in einem UIViewController steht
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)
```



Custom Views

- **Wann erstellen wir unsere eigene UIView subclass?**
 - Wenn wir frei auf dem Screen zeichnen wollen
 - Wenn wir 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 betrachten wir das Zeichnen
- **Zum zeichnen, erstellen wir einfach eine UIView subclass und überschreiben draw(CGRect)**

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

 - Wir können außerhalb des `rect` zeichnen, dies wird aber nie verlangt
 - Das `rect` dient nur zur Optimierung
 - Es sind unsere UIView's `bounds` welche die gesamte Zeichenfläche beschreiben (das `rect` ist eine Teilfläche)
- **NIEMALS (!!) rufen wir draw(CGRect) direkt auf! NIEMALS!!!**
 - Wenn wir View neu zeichnen müssen, lassen wir dies das System wissen durch Aufruf von...
`setNeedsDisplay()`
`setNeedsDisplayInRect(_ rect: CGRect)` // rect ist die Fläche die neu gezeichnet werden muss
 - iOS ruft dann zu einem passenden Zeitpunkt `draw(CGRect)` für uns auf

Custom Views

- Wie implementieren wir nun draw(CGRect)?
 - Wir können uns entweder den drawing context holen und ihm sagen was wir zeichnen wollen, oder...
 - Wir können einen Pfad zum zeichnen erstellen unter Verwendung der UIBezierPath Klasse (so werden wir es machen)
- Core Graphics Konzepte
 1. Wir erhalten einen context um darin zu zeichnen (andere contexts beinhalten Drucken, Off-Screen Buffer, etc.)
 2. Die Funktion [UIGraphicsGetCurrentContext\(\)](#) gibt uns einen context den wir in draw(CGRect) nutzen können
 3. Erstellen von Pfaden (aus Linien, Bögen, etc.)
 4. Setzen von Zeichenattributen wie Farben, Fonts, Texturen, Linienbreite, Linienverbindung, etc.
 5. Stroke oder Fill des erstellten Pfades mit den gegebenen Attributen
- [UIBezierPath](#)
 - Identisch wie Konzept auf der vorletzten Seite, nur dieses mal in einer UIBezierPath Instanz
 - UIBezierPath zeichnet automatisch im “current” Kontext (draw(CGRect) macht das Setup für uns)
 - UIBezierPath hat Methoden zum zeichnen(lineto, arcs, etc.) und setzen von Attributen (linewidth, etc.)
 - Wir nutzen [UIColor](#) zum setzen von stroke und fill Farben
 - UIBezierPath hat Methoden für stroke und fill

Definieren eines Pfades

- Erstellen eines UIBezierPath

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

- Verschieben, Linien oder Arcs zum Pfad hinzufügen

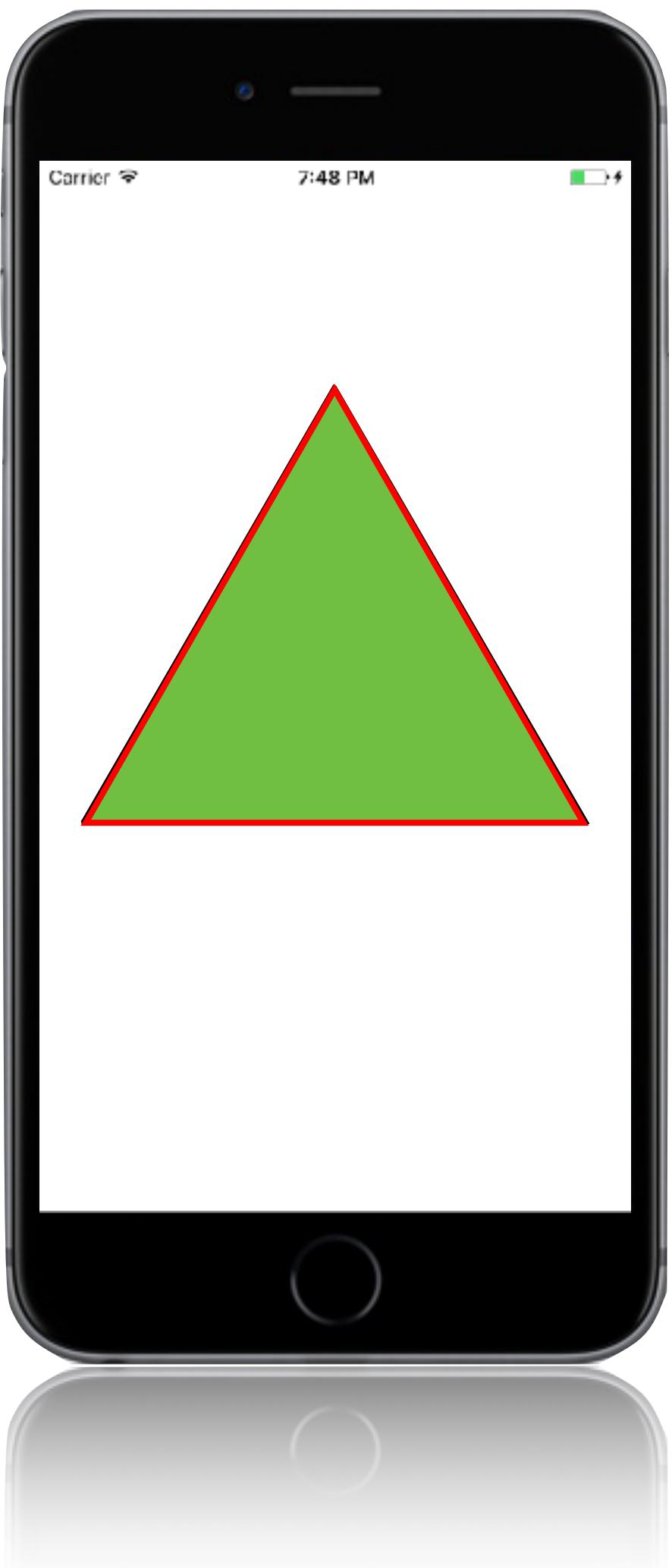
```
path.move(to: CGPoint(80, 50))  
path.addLine(to: CGPoint(140, 150))  
path.addLine(to: CGPoint(10, 150))
```

- Schließen des Pfades (sofern wir das wollen)

```
path.close()
```

- Da wir nun einen Pfad haben, die Attribute und stroke/fill setzen

```
UIColor.green.setFill() // setFill ist eine Methode aus UIColor  
UIColor.red.setStroke() // setStroke ist eine Methode aus UIColor  
path.lineWidth = 3.0 // linewidth ist ein Property in UIBezierPath  
path.fill() // fill ist eine Methode in UIBezierPath  
path.stroke() // stroke ist eine Methode in UIBezierPath
```



Zeichnen

- Wir können auch herkömmliche Formen mit UIBezierPath zeichnen

```
let roundRect = UIBezierPath(roundedRect: CGRect, cornerRadius: CGFloat)
```

```
let oval = UIBezierPath(ovalIn: CGRect)
```

... und andere

- Clippen von zeichnen zu einem UIBezierPath Pfad

```
addClip()
```

- Zum Beispiel kann so zu einem abgerundeten rect geclipppt werden um die Kanten einer Spielkarte zu erzwingen

- Hit Detection

```
func contains(_ points: CGPoint) -> Bool // ob ein Point innerhalb eines Pfad liegt
```

- Der Pfad muss geschlossen sein. Die Winding Rule kann mittels usesEvenOddFillRule Property gesetzt werden

- Etc.

- Noch viel mehr. In der Doku nachlesen!

UIColor

- **Farben werden mittels UIColor gesetzt**

- Es existieren Typ (aka static) vars für standard Farben, z.B. `let green = UIColor.green`
- Wir können diese auch aus RGB, HSB oder einem Pattern (unter Verwendung von UIImage) erstellen

- **Background Farbe für einen UIView**

```
var backgroundColor: UIColor // haben wir für unsere Calculator Buttons verwendet
```

- **Farben haben einen Alpha Wert (Transparenz)**

```
let semitransparentYellow = UIColor.yellow.withAlphaComponent(0.5)
```

- Alpha ist zwischen 0.0 (vollständig transparent) und 1.0 (opaque)

- **Wenn wir in unserem View mit Transparenz zeichnen wollen ...**

- Müssen wir dies das System wissen lassen durch setzen des UIView `var opaque = false`

- **Wir können den gesamten UIView transparent setzen...**

```
var alpha: CGFloat
```

View Transparenz

- **Was passiert, wenn die Views überlappen und Transparenz haben?**
 - Wie schon erwähnt, bestimmt die `subviews` Liste die Reihenfolge und damit wer vorne liegt
 - Tiefere (frühe im Array) können durch transparente Views darüber "durchscheinen"
 - Transparent ist teuer, sollten wir also sparsam einsetzen
- **Vollständiges verstecken eines Views ohne ihn aus der View Hierarchie zu entfernen**
`var hidden: Bool`
 - Ein hidden View zeichnet nichts auf den Screen und empfängt ebenfalls keine Events
 - Nicht so ungewöhnlich wie vielleicht gedacht um temporär einen View zu verstecken

Zeichnen von Text

- Normalerweise verwenden wir ein **UILabel** um Text auf dem Screen anzuzeigen
 - Es gibt aber sicherlich Gründe wenn wir Text in unserer `draw(CGRect)` zeichnen wollen
- Um in `draw(CGRect)` zu zeichnen, verwenden wir **NSAttributedString**

```
let text = NSAttributedString(string: "hello")
text.draw(at: aCGPoint)
let textSize: CGSize = text.size // wie viel Platz der String einnimmt
```
- Mutability erfolgt mittels **NSMutableAttributedString**
 - Ist nicht wie String (d.h. wo let immutable und var mutable bedeutet)
 - Wir verwenden einen anderen Klasse, wenn wir einen mutable attributed String erstellen möchten...

```
let mutableText = NSMutableAttributedString(string: "some string")
```
- **NSAttributedString ist kein String und auch kein NSString**
 - Wir können dessen Inhalt als String/NSString erhalten mittels dessen `string` oder `mutableString` Properties

Attributed String

- **Setzen von Attributed eines attributed String**

```
func setAttributes(_ attributes: [String:Any]?, range: NSRange)
func setAttributes(_ attributes: [String:Any]?, range: NSRange)
```

- Warnung! Dies ist eine pre-Swift API. NSRange ist keine Range
- Und eine Indexierung in den attributed String nutzt Int Indexing (nicht String.Index)
- Es ist vielleicht hilfreich String's utf16 var zu nutzen um einen String.UTF16View zu erhalten
- Ein UTF16View repräsentiert den String als Sequenz von 16 bit Unicode Characters
- Die Characters in einem UTF16View "stellen sich auf" mit attributed String's Charactern
- Aber UTF16View ist noch immer geindext durch String.Index
- Also brauchen wir die UTF16View Character und verwenden startIndex mit index(offsetBy:)

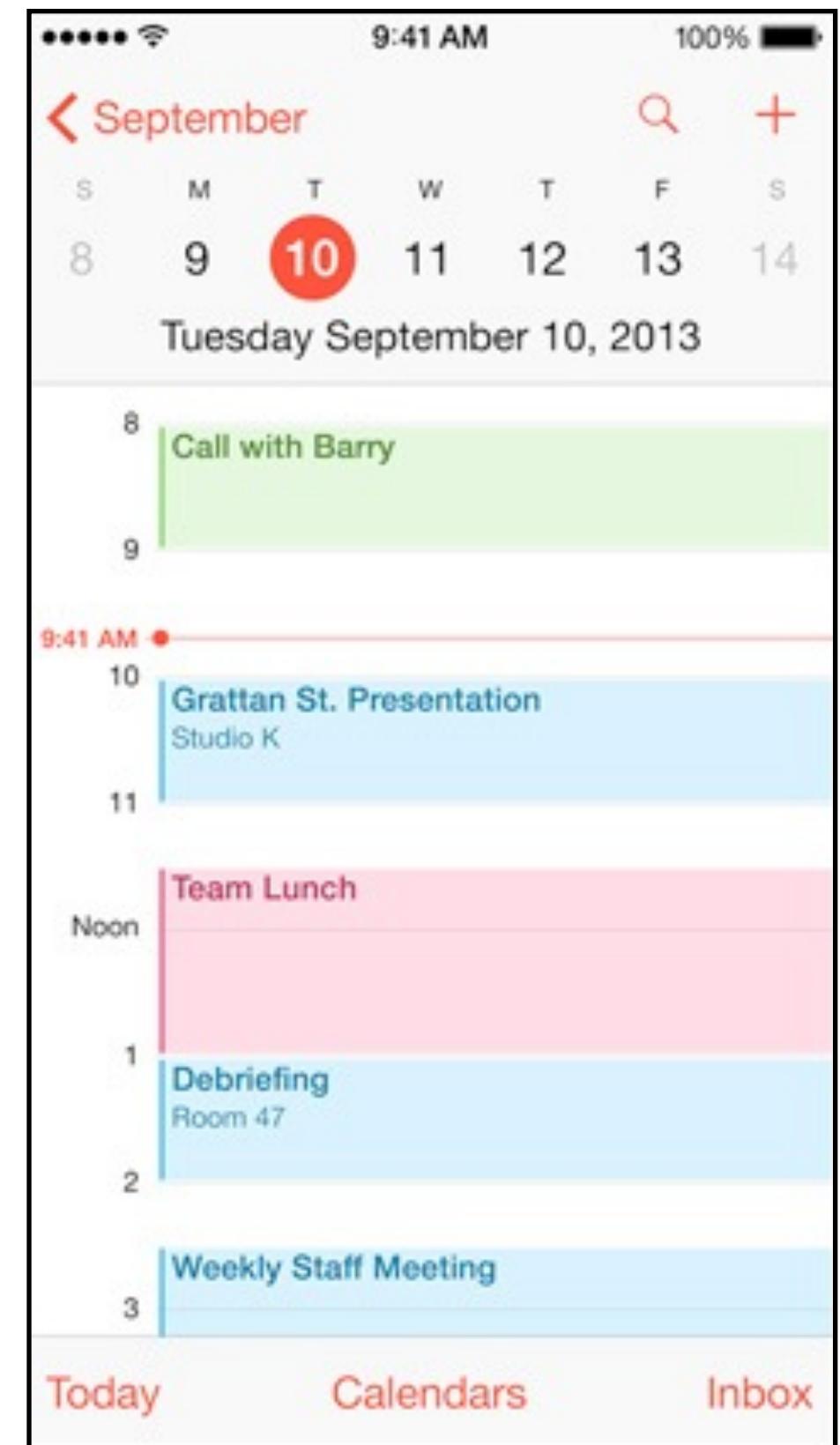
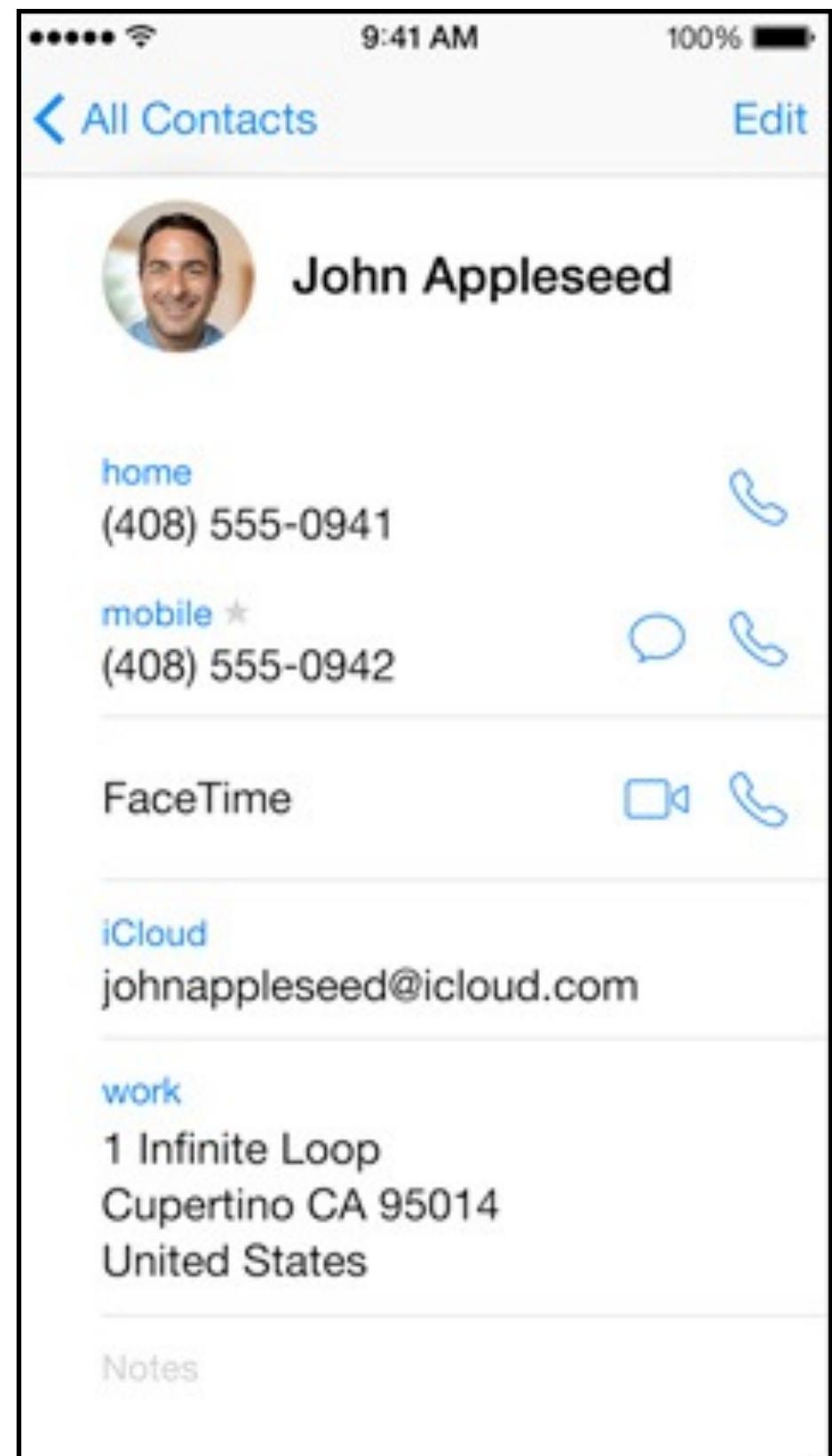
- **Attribute**

```
NSForegroundColorAttributeName : UIColor
NSStrokeWidthAttributeName : CGFloat
NSFontAttributeName : UIFont
```

- Siehe auch die Doku von UIKit für viel mehr

Fonts

- Es ist sehr wichtig Fonts in iOS richtig zu wählen
 - Dies ist fundamental für das Look and Feel des UI



Fonts

- **Der absolut beste Weg eine Font in Code zu erhalten**
 - Erhalten einer preferred Font für einen gegebenen Text Style (z.B. Body, etc.) unter Verwendung dieser UIFont Typ Methode...
`static func preferredFont(forTextStyle: UIFontTextStyle) -> UIFont`
 - Einige der Styles (siehe UIFontDescriptor Dokumentation für mehr)...
`UIFontTextStyle.headline`
 `.body`
 `.footnote`
- **Es existieren auch “System Fonts”**
 - Diese tauchen normalerweise auf Dingen wie Buttons auf
`static func systemFont(ofSize: CGFloat) -> UIFont`
`static func boldSystemFont(ofSize: CGFloat) -> UIFont`
 - Wir verwenden diese nicht für den User's Content. Dafür verwenden wir preferred Fonts.
- **Andere Wege um Fonts zu erhalten**
 - Schauen Sie sich UIFont und UIFontDescriptor für mehr an, Sie sollten dies aber nicht häufig brauchen

Zeichnen von Bildern

- Es existiert ein **UILabel-Equivalent für Bilder**

[UIImageView](#)

- Aber nochmal, wir wollen vielleicht das Bild in unserer `draw(CGRect)` zeichnen...

- Erstellen eines **UIImage Objektes**

```
let image: UIImage? = UIImage(named: "foo") // dies ist ein Optional
```

- Wir haben `foo.jpg` zu unserem Projekt in den [Assets.xcassets](#) File hinzugefügt (haben wir bisher ignoriert)

- Wir können es ebenfalls aus Dateien im Dateisystem erstellen

- (wir haben uns noch nicht darüber unterhalten, wie wir Dateien aus dem Dateisystem erhalten.. dennoch...)

```
let image: UIImage? = UIImage(contentsOfFile: aString)
```

```
let image: UIImage? = UIImage(data: aData) // raw jpg, png, tiff, etc. Image Data
```

- Wir können sogar **keines** erstellen durch **zeichnen mittels Core Graphics**

- Siehe Doku für `UIGraphicsBeginImageContext(CGSize)`

Zeichnen von Bildern

- Sobald wir ein UIImage haben, können wir die Bits auf den Screen bringen

```
let image: UIImage = ...
image.draw(at Point: aCGPoint)          // obere links Ecke auf aCGPoint gesetzt
image.draw(in rect: aCGRect)            // skaliert das Bild passend ins aCGRect
image.drawAsPattern(in rect: aCGRect) // erstellt Tiles des Bildes in aCGRect
```

Redraw bei bounds Change?

- Per Default, gibt es kein redraw, wenn die UIView bounds sich ändern
 - Stattdessen werden die "Bits" des existierenden Bildes skaliert zu der neuen bounds Größe
- Dies ist oft nicht was wir wollen...
 - Glücklicherweise gibt es ein UIView Property um dies zu kontrollieren! Kann ebenfalls in Xcode gesetzt werden

```
let contentMode: UIViewContentMode
```
- UIViewContentMode
 - Nicht den View skalieren, einfach nur irgendwo platzieren...
`.left/.right/.top/.bottom/.topRight/.topLeft/.bottomRight/.bottomLeft/.center`
 - Skalieren der "Bits" des View...
`.scaleToFill/.scaleAspectFill/.scaleAspectFit` // .scaleToFill ist der Default
 - Redraw durch erneuten Aufruf von `draw(CGRect)` (teuer, aber für manchen Inhalt besserer Ergebnisse)...
`.redraw`