Erfahrungen mit Autolayout 2

UI Erstellung ohne Storyboard

Cocoaheads Saar

Worum geht es?

- Erfahrungen beim „Lernen“ von Autolayout

- kein IB, keine XIBs, alles im Code

- gute Informationen im Internet: übel wenig, besonders im Bereich UI für MacOS X

Teil 1 (letztes Mal)

- Grundlagen von Autolayout
- Kurze Zusammenfassung heute
- Github für die Folien von Teil 1

Teil 2 (heute)

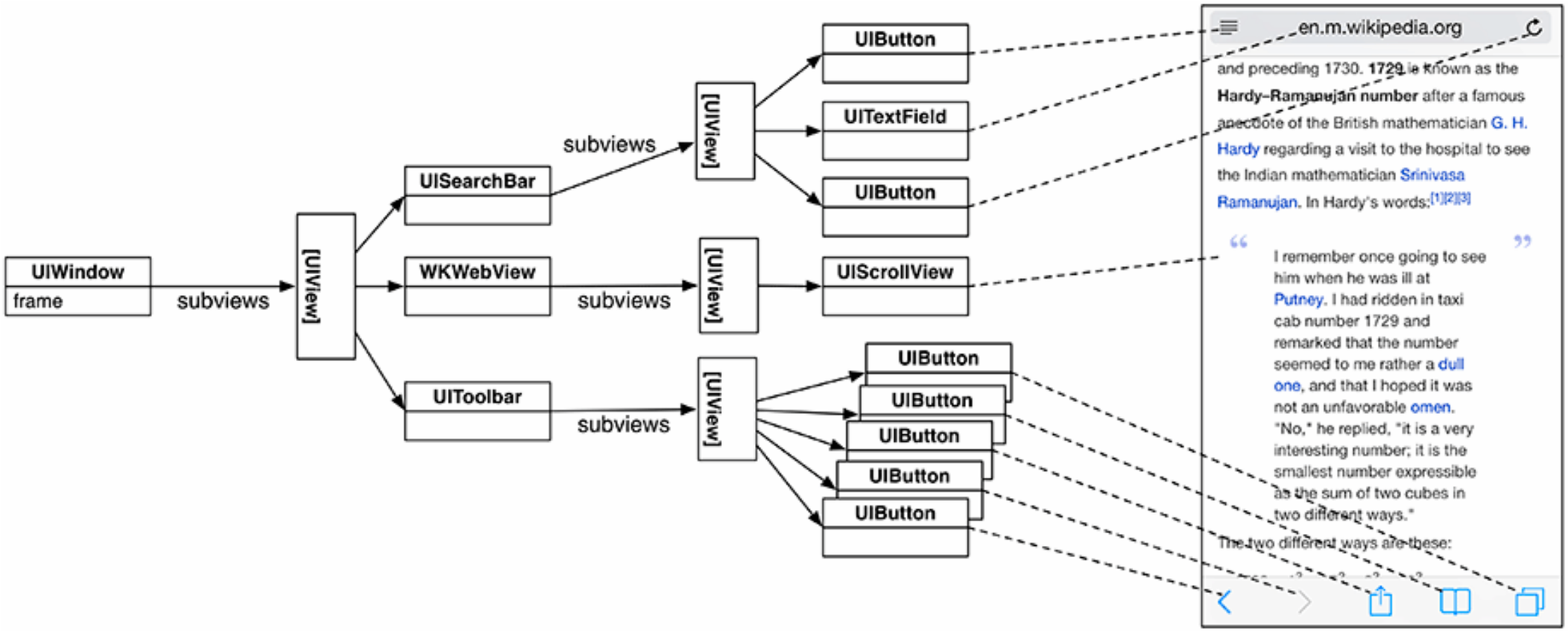
- Schnellkurs aus Teil 1
- Wie sieht das wirklich im Code aus?
- Entwicklung einer Objektbibliothek

Zusammenfassung

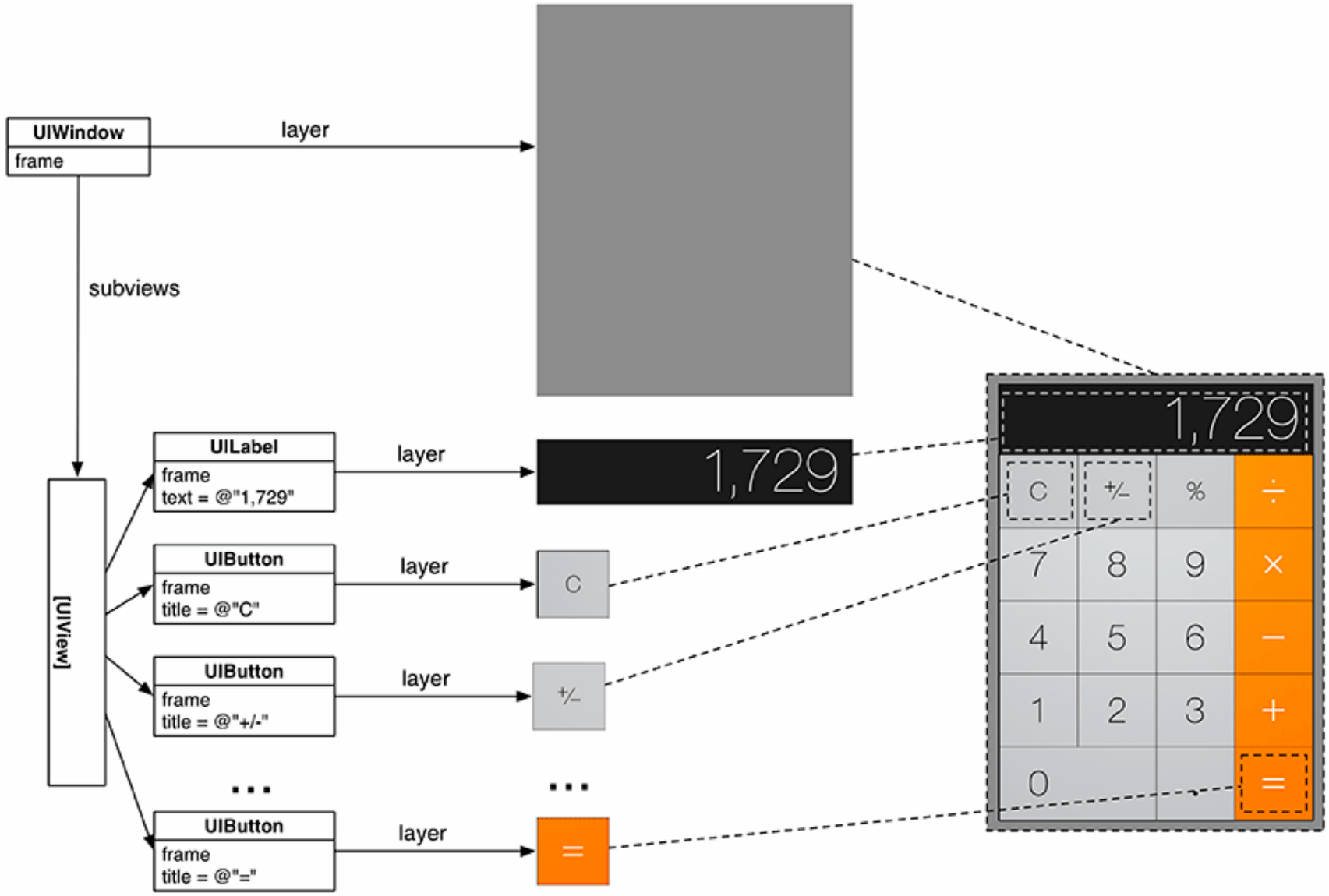
Autolayout 1

Cocoaheads Saar

View Hierarchie bei iOS

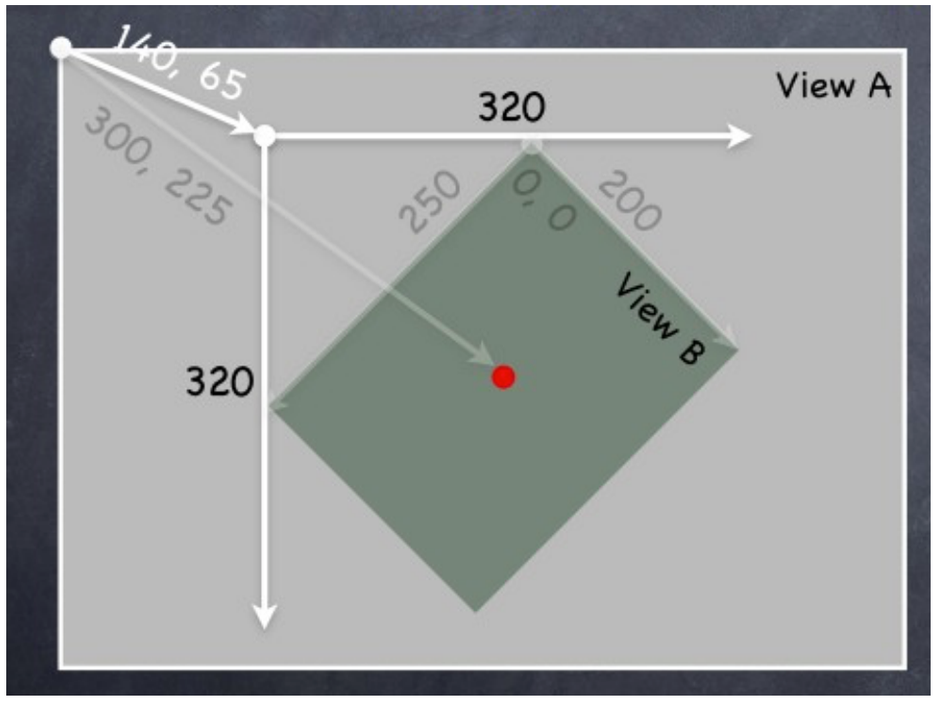


Darstellung

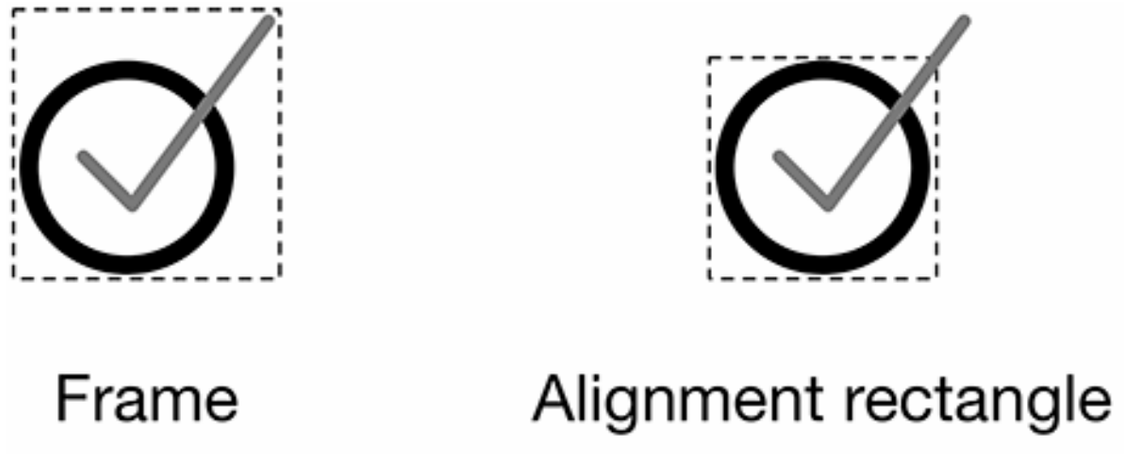


Frame und Bounds, Alignment Rectangle

- superview (View A)
- subview (View B)



- View B:
  bounds = ((0,0),(200,250))
  frame = ((140,65),(320,320))



Constraints

- Constraints sind einfache lineare Gleichungen der Form :

  Item1.Attribut1 = Multiplier x Item2.Attribut2 + Constant

- Item1 und Item2 sind die Views

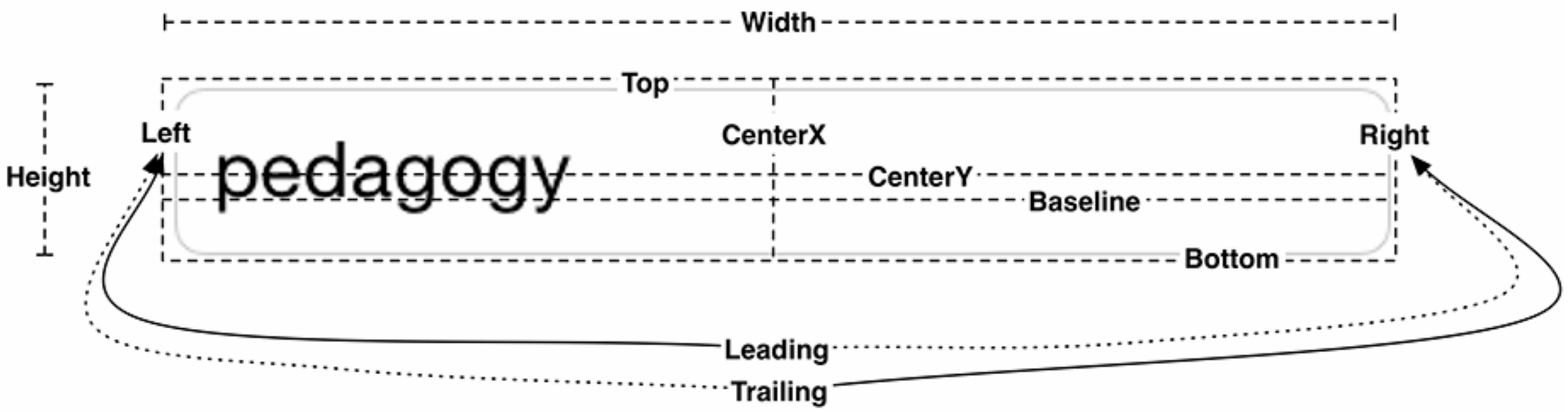
- Attribute in Enum NSLayoutAttribute definiert (nächste Folie)

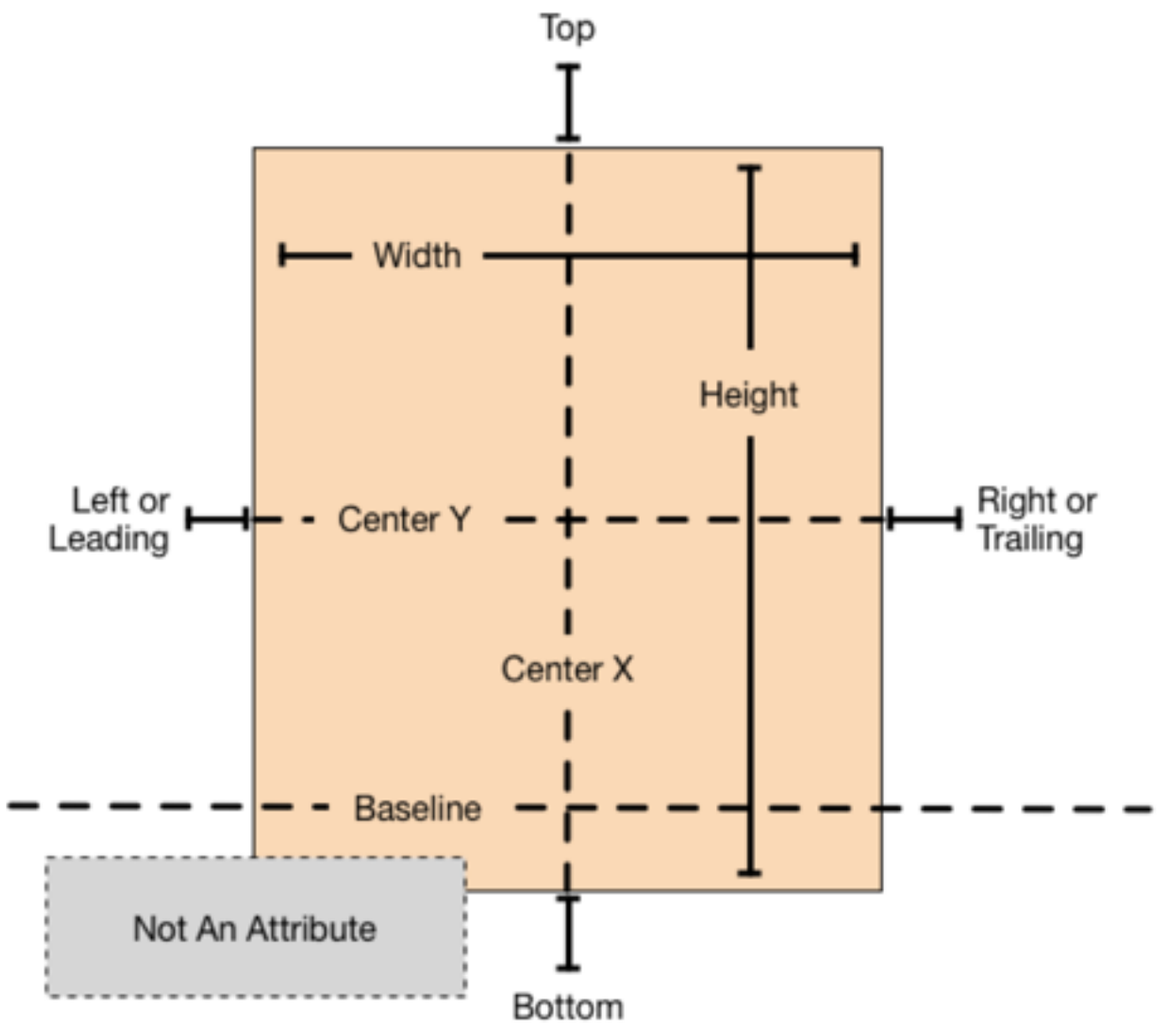
- Multiplier und Constant sind CGFloats

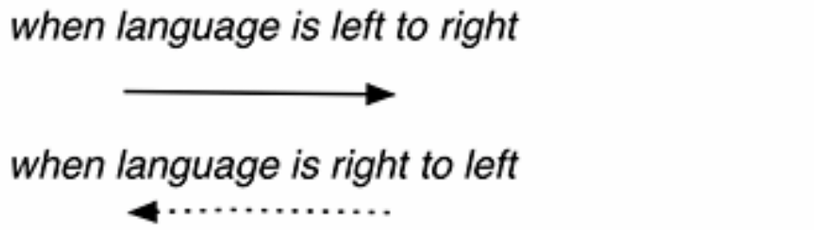
- = kann auch >= / > / < / <= sein
  Blue.leading >= 1.0 * Red.trailing + 8.0
  (minimaler Abstand zwischen Blue und Red ist 8)

- NotAnAttribute bei keiner Abhängigkeit von anderer View
  View.height = 0.0 * nil.NotAnAttribute + 40.0
  (Höhe konstant 40)

- Jede Constraints kann eine Priorität von 1-1000 haben


Width
Top
Left
CenterX
Right
Height
pedagogy
CenterY
Baseline
Bottom
Leading
Trailing


Top
Width
Height
Left or
Leading
Center Y
Right or
Trailing
Center X
Baseline
Not An Attribute
Bottom


when language is left to right
when language is right to left

- Constraints werden genauso geschrieben, wie wir sie eben definiert haben

- textLabel.Baseline = 1.0 x textField.Baseline + 0.0 wird zu :

superView.addConstraint(NSLayoutConstraint(item: textField, attribute: .Baseline, relatedBy: .Equal, toItem: textLabel, attribute: .Baseline, multiplier: 1, constant: 0))

TopLayoutGuide und Co.

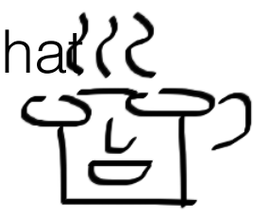
- ToolBar, StatusBar, TabBar und NavigationBar reduzieren die nutzbare Größe und sind beim Drehen des Gerätes u.U. sichtbar oder unsichtbar

- Um das Problem zu lösen, erstellt der **UIViewController** automatisch zwei unsichtbare Views, TopLayoutGuide und BottomLayoutGuide, fügt sie in die View Hierarchie ein und updated automatisch die Größe

- die untere Seite des TopLayoutGuides ist automatisch gleich dem Bottom der untersten Bar oben, entsprechend die obere Seite des BottomLayoutGuides dem Top der obersten Bar unten

- Da dies Views sind, kann man relativ zu BottomLayoutGuide.top bzw. TopLayoutGuides.bottom Views positionieren, deren Position automatisch angepasst

- **Achtung**: das geht erst ab iOS 9. Es gibt immer noch bei jeder UIView ein LayoutMargins, ein UIEdgeInset, was die vier Werte top, left, bottom, right hat



Constraints mit Anchors

```
let margins = layoutMarginsGuide
var layoutConstraints = [NSLayoutConstraint]()
layoutConstraints.append(textLabel.centerYAnchor.constraintEqualToAnchor(textField.centerYAnchor))
layoutConstraints.append(textField.leftAnchor.constraintEqualToAnchor(textLabel.rightAnchor, constant: 10.0))
layoutConstraints.append(textLabel.leftAnchor.constraintEqualToAnchor(margins.leftAnchor))
layoutConstraints.append(textField.rightAnchor.constraintEqualToAnchor(margins.rightAnchor))
layoutConstraints.append(textLabel.widthAnchor.constraintEqualToConstant(textSize))

NSLayoutConstraint.activateConstraints(layoutConstraints)
```


Visual Format

- Eine visuelle Art, die Positionen und Größen zu beschreiben.

- Vorteil: Relationen zwischen mehr als zwei Views beschreibbar und die Anzahl der Constraints ist sehr viel geringer

- Zuerst definiert man sich ein Array mit den Views, für die man Constraints beschreiben will

- Dann beschreibt man die Constraints in ASCII Art

```
let views = ["textLabel" : textLabel, "textField" : textField]
let metrics = ["textSize" : textSize]

var layoutConstraints = [NSLayoutConstraint]()

        layoutConstraints +=
NSLayoutConstraint.constraintsWithVisualFormat("H:|-[textLabel(textSize)]-
[textField]-|", options: [], metrics: metrics, views: views)

        NSLayoutConstraint.activateConstraints(layoutConstraints)
```


Syntax

- [view1]-[view2] : Standardabstand 8
- [view(>=50)] : View mindestens 50 Points breit
- |-10-[view]-10-| : View 10 Points von den Margins
- V:[view1]-10-[view2] : View2 10 Points unter View1
- [view1(==view2)] : View1 genauso breit wie View2
- [view(>=10,<=100] : View zwischen 10 und 100 Points
- [view (100@20)] : View 100 Points, Priorität 20

Status

- Damit sind die Grundlagen für Autolayout per Code erledigt

- Es fehlt noch, wie man den Code organisiert, weil man jetzt sehr viel mehr Codezeilen hat

- Und es fehlt, wie man die einzelnen Views (Buttons, Labels, Bars usw.) erzeugt

Grundprinzipien

- Frames sind immer CGRectMake(0, 0, 0, 0) oder abgekürzt .zero
- NUR die Constraints definieren Größe und Position, damit Auto Layout bei unterschiedlichen Größen oder Rotation auch anpassen kann
- translatesAutoresizingMaskIntoConstraints wird für ALLE Views auf false gesetzt, weil es sonst zu Konflikten und komischen Verhalten kommt
- IntrinsicContentSize beachten
- Constraints sind natürlich mischbar (Anchor mit Visualformat z.B.)

Organisation des Codes

- Bereits ein einfaches Label erzeugt einige Zeilen Code:

```
textLabel = UILabel(frame: .zero)
textLabel.translatesAutoresizingMaskIntoConstraints = false
textLabel.text = "TestIabel"
```

- Diese Zeilen sind IMMER dann nötig, wenn man ein Label verwendet und das geschieht erfahrungsgemäß sehr oft

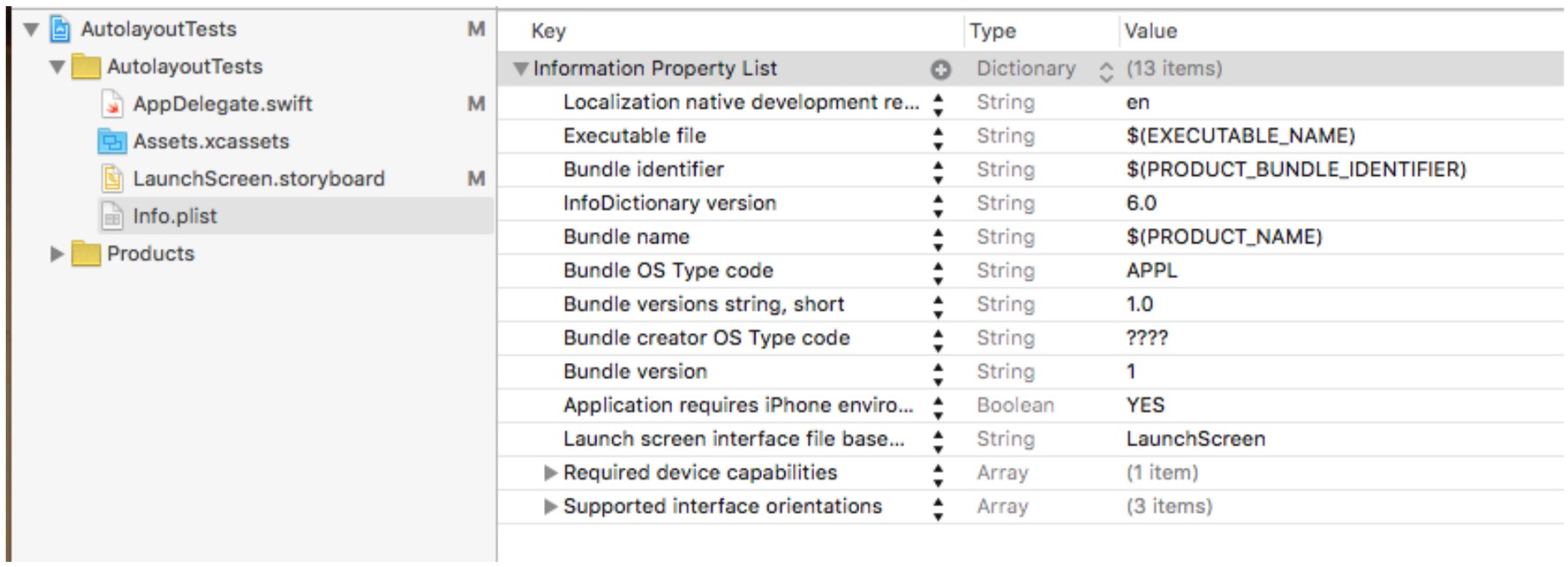
- also sollte man sich eine Objektbibliothek erstellen und daraus die Views erzeugen

- weiterer Vorteil: Fehleranfälligkeit sinkt

- Baukastenbibliothek mit class func

Vorarbeit

- Neues Projekt: iOS App, Single View Template
- Entfernen des Storyboards:
  - Info.plist: Entfernen des Eintrags „Main Storyboard file base name“
  - Löschen von Main.storyboard und Viewcontroller.swift
  - App startet trotzdem, natürlich mit weißem Screen



Organisation des Codes

- MVC Design
- Model, View und Controller sind jeweils eigene Dateien
- Model entfällt hier erstmal noch

Basis MainView

```
class MainView: UIView {

   // views deklarieren

    override init(frame: CGRect) {

        // views erzeugen

        super.init(frame: frame)

        backgroundColor = .whiteColor()

        // Views als Subviews hinzufügen

        // Constraints erstellen
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Cocoaheads Saar

Basis MainViewController

```
class MainViewController: UIViewController {

    var mainView: MainView {
        return view as! MainView
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // weiterer View Code (z.B. Target setzen)
    }

    override func loadView() {

        let contentView = MainView(frame: .zero)
        view = contentView
    }

    override func viewWillLayoutSubviews() {
      // eventuell fehlende Constraints (z.B. wegen topLayoutGuide)
    }

}
```


Basis AppDelegate.swift

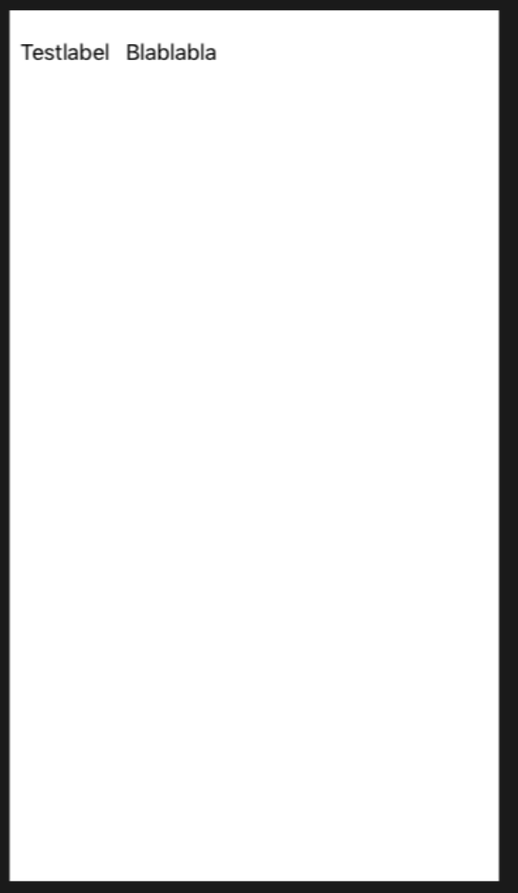
```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow? = UIWindow(frame: UIScreen.mainScreen().bounds)

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[NSObject: AnyObject]?) -> Bool {

        let mainViewController = MainViewController()

        window?.rootViewController = mainViewController
        window?.makeKeyAndVisible()

        return true
    }

}
```


Layouten per Playground

- Für Swift kann man im Playground Layouten
- Es ist zwar nicht der fertige Code, den man dann im Projekt verwendet, aber zum Austesten, ob die Constraints passen, reicht es vollkommen aus
- Da man kein Device hat, muss man sich als erstes eine UIView erzeugen, die der Größe eines Devices entspricht (http://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions)
- Wichtig: das ist keine Simulation eines Devices, es fehlt also z.B. die StatusBar
- Natürlich kann man auch mit den Klassen arbeiten, die wir vorher definiert haben
- man muss den AssistentEditor aktivieren, um das Resultat zu sehen

```
1  //: Playground - noun: a place where people can play
2
3  import UIKit
4  import XCPlayground
5  let mainFrame = UIView(frame: CGRect(x: 0, y: 0, width: 375, height: 667)) // iPhone 6
6  mainFrame.backgroundColor = .whiteColor()
7
8  XCPlaygroundPage.currentPage.liveView = mainFrame
9
10 var textLabel: UILabel
11 var textField: UITextField
12
13 // views
14 textLabel = UILabel(frame: .zero)
15 textLabel.translatesAutoresizingMaskIntoConstraints = false
16 textLabel.text = "Testlabel"
17 let textSize = textLabel.intrinsicContentSize().width
18
19 textField = UITextField(frame: .zero)
20 textField.text = "Blablabla"
21
22 textField.translatesAutoresizingMaskIntoConstraints = false
23 textField.borderStyle = .Line
24
25 mainFrame.addSubview(textLabel)
26 mainFrame.addSubview(textField)
27
28 var layoutConstraints = [NSLayoutConstraint]()
29 layoutConstraints.append(textLabel.centerYAnchor.constraintEqualToAnchor(textField.centerYAnchor))
30 layoutConstraints.append(textField.leftAnchor.constraintEqualToAnchor(textLabel.rightAnchor, constant: 10.0))
31 layoutConstraints.append(textLabel.leftAnchor.constraintEqualToAnchor(mainFrame.leftAnchor, constant: 8.0))
32 layoutConstraints.append(textField.rightAnchor.constraintEqualToAnchor(mainFrame.rightAnchor, constant: 8.0))
33 layoutConstraints.append(textLabel.widthAnchor.constraintEqualToConstant(textSize))
34 layoutConstraints.append(textLabel.topAnchor.constraintEqualToAnchor(mainFrame.topAnchor, constant: 20.0))
35 layoutConstraints.append(textField.topAnchor.constraintEqualToAnchor(mainFrame.topAnchor, constant: 20.0))
36
37 NSLayoutConstraint.activateConstraints(layoutConstraints)
38
39 mainFrame
40 textField.frame
41 textField.bounds
42
43 textLabel.frame
44 textLabel.bounds
45
46
```

```
UIView
UIView

<UILabel: 0x7ffa1ca0f9a0; fra...
<UILabel: 0x7ffa1ca0f9a0; fra...
<UILabel: 0x7ffa1ca0f9a0; fra...
69

<UITextField: 0x7ffa1c912f50;...
<UITextField: 0x7ffa1c912f50;...

<UITextField: 0x7ffa1c912f50;...
<UITextField: 0x7ffa1c912f50;...

UIView
UIView

[]
[{NSObject}]
[{NSObject}, {NSObject}]
[{NSObject}, {NSObject}, {NS...
[{NSObject}, {NSObject}, {NS...
[{NSObject}, {NSObject}, {NS...
[{NSObject}, {NSObject}, {NS...
[{NSObject}, {NSObject}, {NS...


UIView
{x 87 y 20 w 296 h 25}
{x 0 y 0 w 296 h 25}

{x 8 y 20 w 69 h 25}
{x 0 y 0 w 69 h 25}
```

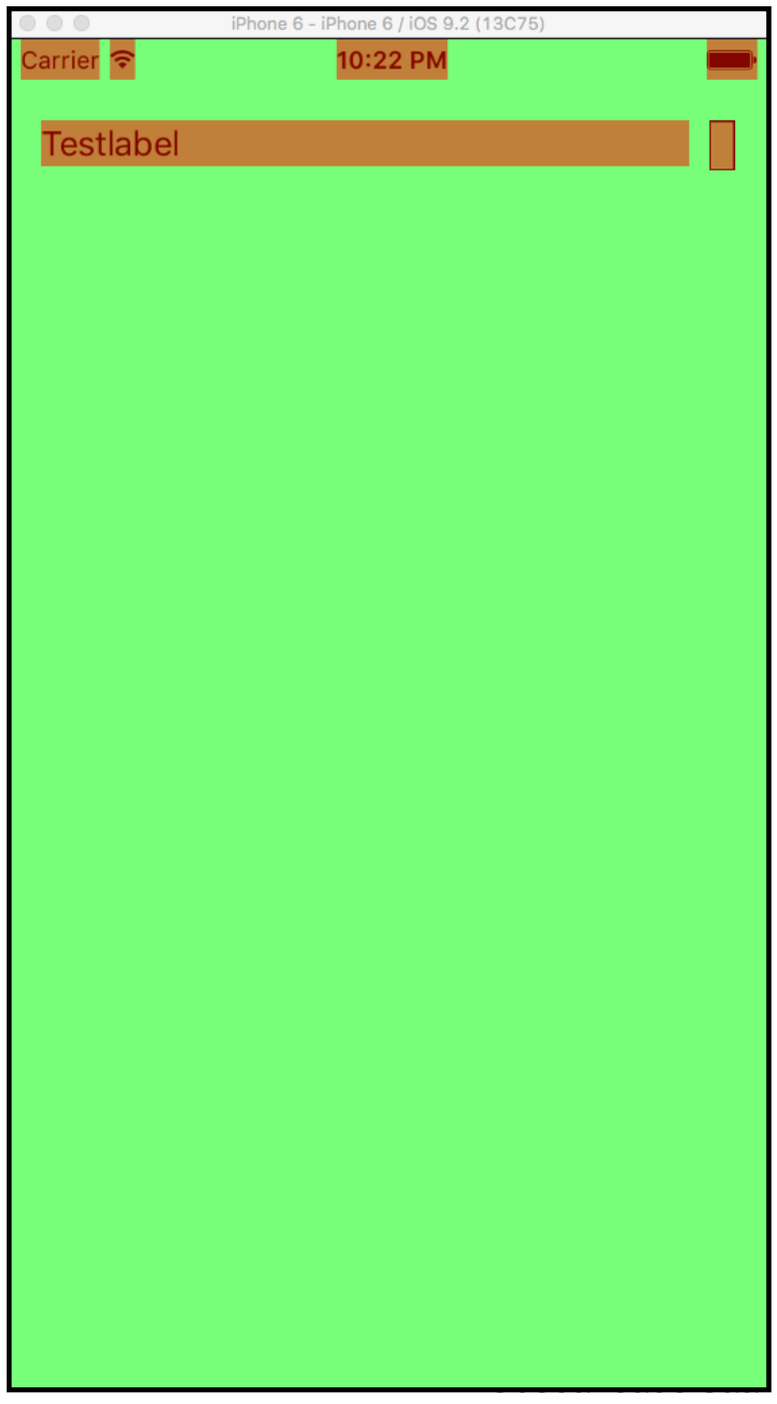



Wie findet man Fehler?

- Man startet die App im Simulator

- Im Menü „Debug“ wählt man „Color Blended Layers“

- das geht NUR für iOS



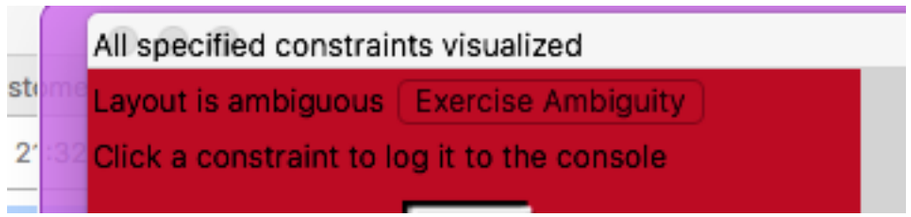
Fehlersuche bei MacOS X

- AppDelegate:

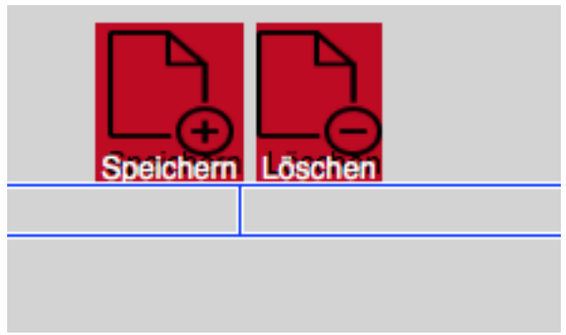
```
// show constraints
NSUserDefaults.standardUserDefaults().setBool(true, forKey:
„NSConstraintBasedLayoutVisualizeMutuallyExclusiveConstraints")
```

- Visuelle Hilfe bei Constraintfehlern

- Sobald ein Fenster mit Constraintfehler angezeigt wird, wird das gesamte Fenster lila hinterlegt



- Zusätzlich wird die Stelle des Fehlers mit blauen Linien markiert



- Klickt man mit der Maus auf die Linien, erscheinen in der Konsole von Xcode die betroffenen Constraints

```
2016-02-24 21:35:05.598 Uprojectme[12614:265476] Clicked on overlapping visualized constraints: (
    "<NSLayoutConstraint:0x60800008a280 V:[NSStackView:0x100f39450]-(NSSpace(20))-|   (Names: '|':NSStackView:0x100f35660 )> (Actual Distance - pixels):20",
    "<NSLayoutConstraint:0x608000094f50 'NSStackView.Edge.Bottom' V:[NSStackView:0x100f39450]-(0)-|   (Names: '|':NSStackView:0x100f35660 )> (Actual Distance - pixels):20"
)
```


Genug Theorie

jetzt geht es endlich los

einfache Views

- Labels, TextFields, SegmentedControlls, Slider, TextViews sind einfach

- Buttons eigentlich auch (Action nur dazu)

UIPickerviews

- UIPickerviews brauchen ein Delegate und eine Datasource
- die werden im UIViewController gesetzt. Dazu muss dieser die entsprechenden Protokolle akzeptieren
- Zur Übersicht implementiert das Delegate und die Datasource als Extension des Viewcontrollers
- extension ViewController: UIPickerViewDataSource {

```
func pickerView(pickerView: UIPickerView, viewForRow row: Int, forComponent component: Int, reusingView view: UIView?) -> UIView { }
func numberOfComponentsInPickerView(colorPicker: UIPickerView) -> Int { }
func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int { }
}
```

- extension ViewController: UIPickerViewDelegate {

```
func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? { }
func pickerView(pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) { }
}
```


UITabBar

- Man implementiert diese im AppDelegate
- Für jede Tab erstellt man eine Datei für View und eine ViewController, in denen dann die UI der einzelnen Tabs erstellt wird
-

```
let tabBarController = UITabBarController()
let mainViewController = MainViewController()
mainViewController.title = "Simple"
let mainWithAnchorsViewController = MainWithAnchorsViewController()
mainWithAnchorsViewController.title = "Anchors"
…
tabBarController.viewControllers = [
   UINavigationController(rootViewController: mainViewController),
   UINavigationController(rootViewController: mainWithAnchorsViewController)
   …
]
window?.rootViewController = tabBarController
window?.makeKeyAndVisible()
```


UINavigationController

- Wird ebenfalls in der AppDelegate vor der TabBar erzeugt
- In den einzelnen ViewController Dateien können dann Buttons und Funktionalität hinzugefügt werde
-

```
let nav = UINavigationController()
let mainView = UIViewController(nibName: nil, bundle: nil)
nav.viewControllers = [mainView]
nav.navigationBar.titleTextAttributes = [NSForegroundColorAttributeName: UIColor.whiteColor()]
UINavigationBar.appearance().barTintColor = UIColor.redColor()
UINavigationBar.appearance().tintColor = UIColor.whiteColor()
UINavigationBar.appearance().barStyle = UIBarStyle.BlackTranslucent
```

ViewController:

```
let leftBarButton = UIBarButtonItem(barButtonSystemItem: UIBarButtonSystemItem.Cancel, target: self, action: "leftButtonPressed:")
self.navigationItem.setLeftBarButtonItem(leftBarButton, animated: true)
let rightBarButton = UIBarButtonItem(barButtonSystemItem: UIBarButtonSystemItem.Save, target: self, action: "rightButtonPressed:")
self.navigationItem.setRightBarButtonItem(rightBarButton, animated: true)
```


UIStackviews

- StackViews sind Container für Views, für die man Constraints definieren kann.
- Beispiel: textField und textLabel sollen in einen Stackview. Nötige Constraints für das obige sind dann:

1. Stackview geht von rechten bis linken Margin
2. Stackview hat Abstand 20 von oben

- Im Stackview sollen dann textField und textLabel horizontal ausgerichtet sein, textLabel hat Breite textSize und den Rest macht Auto Layout
- besonders wichtig bei mehr als zwei Views im Stackview
- ein Stackview kann auch aus anderen Stackviews bestehen
- Achtung: geht erst ab iOS 9

MainStackView

```
var textLabel: UILabel
var textField: UITextField
var textStackView: UIStackView

override init(frame: CGRect) {
    // views
    textLabel = UILabel(frame: .zero)
    textLabel.translatesAutoresizingMaskIntoConstraints = false
    textLabel.text = "Testlabel"
    let textSize = textLabel.intrinsicContentSize().width
    textField = UITextField(frame: .zero)
        textField.borderStyle = .Line
        textField.translatesAutoresizingMaskIntoConstraints = false

        // stack view
        textStackView = UIStackView(arrangedSubviews: [textLabel, textField])
        textStackView.translatesAutoresizingMaskIntoConstraints = false
        textStackView.axis = .Horizontal
        textStackView.spacing = 10
        textStackView.distribution = .Fill

        super.init(frame: frame)
        backgroundColor = .whiteColor()
        addSubview(textStackView)

        let views = ["textStackView" : textStackView]
        var layoutConstraints = [NSLayoutConstraint]()
        layoutConstraints += NSLayoutConstraint.constraintsWithVisualFormat("H:|-[textStackView]-|", options: [],
metrics: nil, views: views)
        layoutConstraints.append(textLabel.widthAnchor.constraintEqualToConstant(textSize))
        NSLayoutConstraint.activateConstraints(layoutConstraints)
   }
```


MainStackViewController

```
override func viewWillLayoutSubviews() {

    let topLayoutGuide = self.topLayoutGuide

        let views = ["topLayoutGuide": topLayoutGuide, "textStackView":
mainStackView.textStackView] as [String:AnyObject]

NSLayoutConstraint.activateConstraints(NSLayoutConstraint.constraintsWithVisual
Format("V:[topLayoutGuide]-[textStackView]", options: [], metrics: nil, views:
views))

    }
```


UITableViews

- gleich richtig: mit Custom Cell und eigenem DataProvider
- View und ViewController zur „globalen“ Kontrolle des Verhaltens
- DataProvider und Custom Cell zur Kontrolle der Daten und der Cell
- je eine eigene Datei für DataProvider und für Custom Cell
- Scrollen automatisch

DataProvider

- Registrieren der Custom Cell als gültige TableViewCell

```
func registerCellsForTableView(tableView: UITableView) { }
```

- Zusätzlich werden die nötigen Methoden der Protokolle implementiert

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int { }
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell { }
func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) { }
func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat { }
```

- Kommen die Daten von Core Data, wird hier auch der FetchedResultController definiert

Custom Cell

- Struktur wie die Implementierung einer normalen View

- Damit definiert man, wo welche View (in der Regel Label oder TextField oder Image) in der Cell angeordnet wird

- Dann wird dies automatisch für jede Cell der TableView angewendet

Zusammenbau

- der DataProvider und die Custom Cell sind definiert
- jetzt muss der View Controller alles zusammensetzen
- im loadView() wird eine Instanz des DataProvider erzeugt
- im viewDidLoad() Datasource und Delegate gesetzt. Ausserdem die Custom Cell als gültige Cell registriert

Detailansicht

- das werden logischerweise wieder zwei neue Dateien (View und View Controller)

- bei kleinen Displays und vielen Views wird das größer als der Bereich, der sichtbar ist. Dann braucht man eine ScrollView

UIScrollview

- Am einfachsten erzeugt man eine ContainerView, in die man die einzelnen Viewelemente als Subviews einfügt und per Constraints positioniert

- die ContainerView wird als Subview in die Scrollview eingefügt und per Constraints positioniert

- den Rest macht Autolayout

Mac OSX UI

- Theoretisch wie bei iOS, praktisch ganz anders
- Man verwendet ebenfalls eine Objektbibliothek
- Problem der vielen Fenster
- bei iOS UI per Code gibt es noch einigermaßen viele Infos im Internet
- bei Mac OSX UI per Code findet man fast nichts mehr

Noch viel lernen du musst :)

–Yoda