App-Entwicklung für iOS und OS X

SS 2016

Stephan Gimbel

Extensions

Hinzufügen von Methoden und Properties zu Klassen (auch wenn der Quellcode nicht zur Verfügung steht)
Früher/Alternativ: Catergory Pattern

Ein paar Einschränkungen

- Keine Re-Implementierung von Methoden oder Properties die schon existieren (nur hinzufügen von neuen)
- Die Properties die hinzugefügt werden, dürfen keinen Storage haben

Wird leider oft missbraucht

- Sollte für Klarheit und Lesbarkeit sorgen, nicht zur Verschleierung
- Nicht als Ersatz für gutes objekt-orientiertes Design verwenden
- Am besten (für Beginner) für kleine, gekapselte Hilfsfunktionen verwenden
- Kann verwendet werden um Code zu organisieren, benötigt aber eine gute Software-Architektur
- Im Zweifelsfall (jetzt noch) nicht benutzen

Protocols

Möglichkeit einer minimalen API

Statt einen Caller zu zwingen eine Klasse/Struct zu übergeben, kann gezielt danach gefragt werden was gebraucht wird

Es müssen nur die Funktionen und Properties spezifiziert werden

Ein Protocol ist ein Typ wie jeder andere Typ, bis auf...

- Es hat keinen Speicher oder Implementierung mit sich assoziiert
- Jeder Speicher oder Implementierung, die das Protocol implementieren muss ist ein Implementierung-Typ
- Ein Implementierungstyp kann eine Klasse, Struct oder Enum sein
- Ansonsten kann ein Protocol als Typ genutzt werden um Variablen zu deklarieren, als Funktionsparameter, etc.

Drei Aspekte eines Protokolls

1. Protocol Declaration (welche Funktionen/Properties sind Teil des Protokolls)
2. Declaration in der eine Klasse, Struct oder Enum bekannt gibt, dass ein Protokoll implementiert wird
3. Die eigentliche Implementierung in besagter Klasse, Struct oder Enum

Protocols

Deklaration des Protokolls...

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Protocols

Deklaration des Protokolls…

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Jeder der `SomeProtocol` implementiert, muss auch `InheritedProtocol1` und `2` implementieren.

Protocols

Deklaration des Protokolls…

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Jeder der `SomeProtocol` implementiert, muss auch `InheritedProtocol1` und 2 implementieren.

Spezifizieren ob ein Property get only oder `get` und `set` ist

Protocols

Deklaration des Protokolls…

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Jeder der `SomeProtocol` implementiert, muss auch `InheritedProtocol1` und 2 implementieren.

Spezifizieren ob ein Property get only oder `get` und `set` ist.

Eine Funktion die den Receiver verändert, muss als solche mit `mutating` gekennzeichnet werden.

Protocols

Deklaration des Protokolls…

```
protocol SomeProtocol : class, InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Jeder der `SomeProtocol` implementiert, muss auch `InheritedProtocol1` und 2 implementieren.

Spezifizieren ob ein Property get only oder `get` und `set` ist.

Eine Funktion die den Receiver verändert, muss als solche mit `mutating` gekennzeichnet werden.
(außer das Protokoll wird zum implementieren für Klassen eingeschränkt mit dem class Schlüsselwort)

Protocols

Deklaration des Protokolls…

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Jeder der `SomeProtocol` implementiert, muss auch `InheritedProtocol1` und 2 implementieren.

Spezifizieren ob ein Property get only oder `get` und `set` ist.

Eine Funktion die den Receiver verändert, muss als solche mit `mutating` gekennzeichnet werden.
(außer das Protokoll wird zum implementieren für Klassen eingeschränkt mit dem `class` Schlüsselwort)

Es kann ebenfalls spezifiziert werden dass ein Implementieren einen gegebenen Initializer implementieren müssen.

Protocols

Implementieren eines Protokolls...

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol  &
    // AnotherProtocol
}
```

Konformität zu einem Protokoll stehen nach der Superclass für eine Klasse.

Protocols

Implementieren eines Protokolls...

```
enum SomeClass : SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol  &
    // AnotherProtocol
}
```

Konformität zu einem Protokoll stehen nach der Superclass für eine Klasse.

(Hoffentlich) Offensichtlich haben Enums und Structs keine Superclass.

Protocols

Implementieren eines Protokolls...

```
struct SomeClass : SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol  &
    // AnotherProtocol
}
```

Konformität zu einem Protokoll stehen nach der Superclass für eine Klasse.

(Hoffentlich) Offensichtlich haben Enums und Structs keine Superclass.

Eine beliebige Anzahl von Protokollen können von einer Klasse, Struct oder Enum implementiert werden.

Protocols

Implementieren eines Protokolls...

```
class SomeClass : SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here, including ...
    required init(...)
}
```

Konformität zu einem Protokoll stehen nach der Superclass für eine Klasse.

(Hoffentlich) Offensichtlich haben Enums und Structs keine Superclass.

Eine beliebige Anzahl von Protokollen können von einer Klasse, Struct oder Enum implementiert werden.

In einer Klasse müssen `init`s als `required` markiert werden (ansonsten könnte eine Subclass nicht konform zum Protokoll sein).

Protocols

Implementieren eines Protokolls...

```
extension Something : SomeProtocol {
    // implementation of SomeProtocol here
    // no stored properties
}
```

Konformität zu einem Protokoll stehen nach der Superclass für eine Klasse.

(Hoffentlich) Offensichtlich haben Enums und Structs keine Superclass.

Eine beliebige Anzahl von Protokollen können von einer Klasse, Struct oder Enum implementiert werden.

In einer Klasse müssen `inits` als `required` markiert werden (ansonsten könnte eine Subclass nicht konform zum Protokoll sein).

Es ist erlaubt Protokoll Konformität mittels einer Extension hinzuzufügen.

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
```

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.moveTo(...)
```

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
```

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
thingToMove = square
```

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] =[prius, square]
```

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] =[prius, square]

func slide(slider: Moveable) {
   let positionToSlideTo = ...
   slider.moveTo(positionToSlideTo)
}
slide(prius)
slide(square)
```

Protocols

Verwendung von Protokollen

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

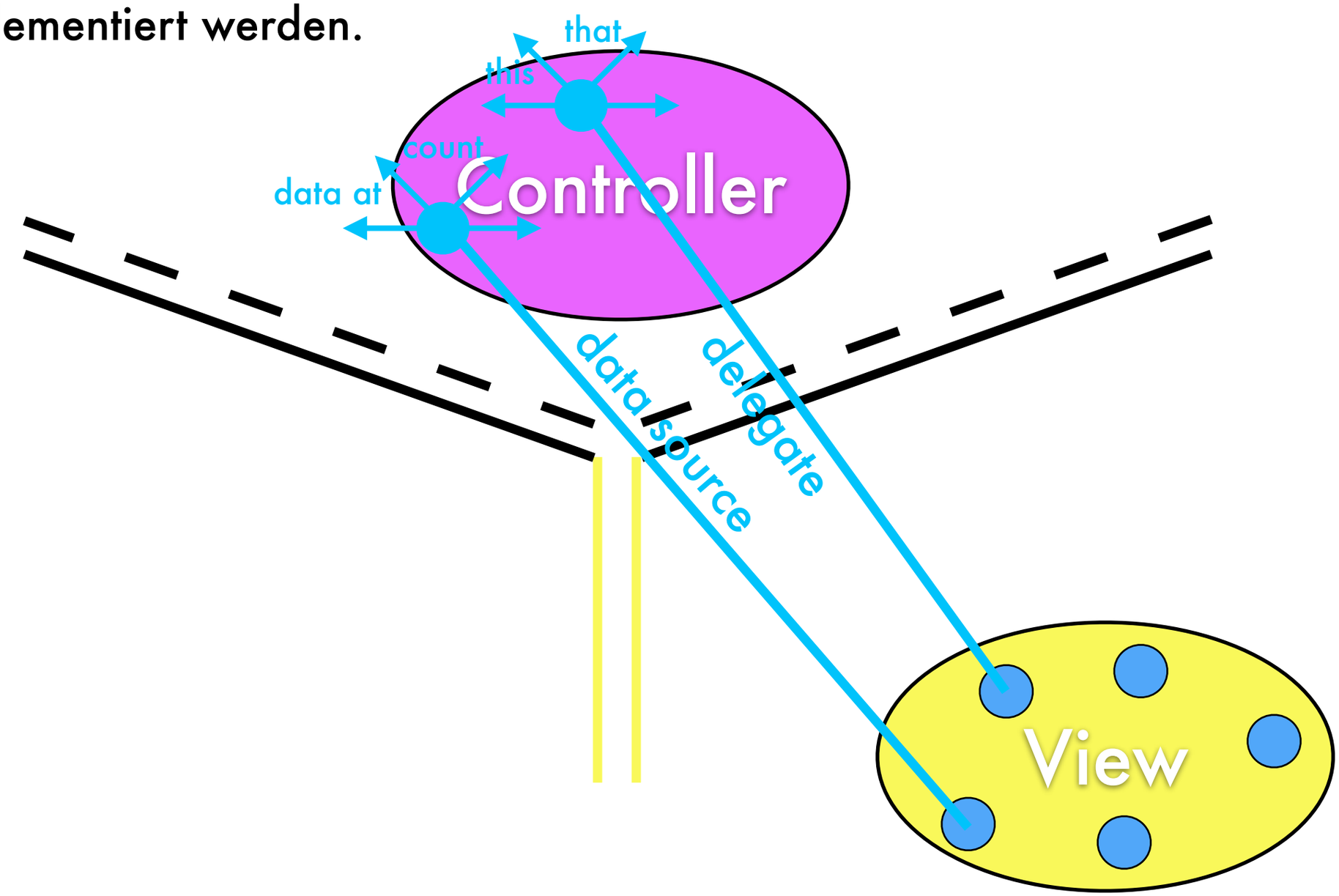
```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] =[prius, square]

func slide(slider: Moveable) {
   let positionToSlideTo = ...
   slider.moveTo(positionToSlideTo)
}
slide(prius)
slide(square)
func slipAndSlide(x: protocol<Slippery,Moveable>)
slipAndSlide(prius)
```

Delegation

Wichtiger Ort um Protokolle zu verwenden

So kann “blinde” Kommunikation zwischen View und einem Controller implementiert werden.



Delegation

Wichtiger Ort um Protokolle zu verwenden

So kann “blinde” Kommunikation zwischen View und einem Controller implementiert werden.

To sum it up…

1. Erstellen eines Delegation Protocols (definiert was der View vom Controller erwartet)
2. Erstellen eines delegate Property im View dessen Typ das des Delegation Protocols ist
3. Verwendung des delegate Property im View um Dinge zu erhalten/ auszuführen die der View nicht besitzt oder kontrollieren kann
4. Controller gibt an, dass er ein Protokoll implementiert
5. Controller setzt sich selbst als das Delegate des Views durch setzen des Property in #2
6. Implementierung des Protokolls im Controller

Jetzt ist der View an den Controller gebunden, weiss aber immer noch nicht was der Controller eigentlich ist. Daher ist der View immer noch generisch/ wiederverwendbar.

Praktikum

Controller

```
func happinessForFaceView... {
   ... happiness ...
}
```

View

```
protocol FaceViewDataSource: class {
   ...
}
```

```
weak var dataSource: FaceViewDataSource?
```

```
override func drawRect(rect: CGRect) {
   ...
   let happiness = dataSource...
   ...
}
```

Praktikum

Controller

```
func happinessForFaceView... {
   ... happiness ...
}
```

View

```
protocol FaceViewDataSource: class {
   ...
}
```

```
weak var dataSource: FaceViewDataSource?
```

```
override func drawRect(rect: CGRect) {
   ...
   let happiness = dataSource...
   ...
}
```

Gestures

Wir wissen nun wie in UIViews gezeichnet wird, aber wie werden Touch Gesten abgearbeitet?

Wir können uns von Touch Events benachrichtigen lassen (touch down, moved, up, etc.).

Oder wir können auf vordefinierte Gesten reagieren. Letzteres wollen wir machen.

Gesten werden von einer Instanz von `UIGestureRecognizer` erkannt

Die Basisklasse ist abstrakt. Wir verwenden nur konkrete Subclasses zur Erkennung.

Zwei Dinge um einen Gesture Recognizer nutzen zu können

1. Hinzufügen eines Gesture Recognizers zu einem UIView (damit der UIView die Geste erkennen kann)
2. Zur Verfügung stellen einer Funktion, welche die Geste abarbeitet (nicht notwendigerweise im UIView)

Gestures

1. Hinzufügen eines Gesture Recognizers zu einem UIView (damit der UIView die Geste erkennen kann)
2. Zur Verfügung stellen einer Funktion, welche die Geste abarbeitet (nicht notwendigerweise im UIView)

Normalerweise wird Punkt #1 vom Controller erledigt

Manchmal erledigt der UIView dies, aber nur dann, wenn der View ohne die Geste nicht existieren kann.

Punkt #2 wird entweder vom UIView oder einem Controller zur Verfügung gestellt

Hängt von der Situation ab.

Wenn eine Geste Dinge beeinflusst, die der View besitzt oder nur diesen betreffen, dann kann dies im UIView geschehen.

Wenn eine Geste Dinge beeinflusst, die der View nicht besitzt oder die ihn indirekt betreffen, dann geschieht dies i.d.R. im Controller.

Gestures

Hinzufügen eines Gesture Recognizers zu einem UIView
Wenn der UIView des Controllers z.B. eine “Pan” Geste erkennen soll.

```
@IBOutlet weak var pannableView: UIView {
   didSet {
     let recognizer = UIPanGestureRecognizer(target: self, action:
                                                    #selector(ClassName.pan))
     pannableView.addGestureRecognizer(recognizer)
   }
}
```

Dies ist ein normales Outlet zum UIView der die Geste erkennen soll.

Gestures

Hinzufügen eines Gesture Recognizers zu einem UIView
Wenn der UIView des Controllers z.B. eine “Pan” Geste erkennen soll.

```
@IBOutlet weak var pannableView: UIView {
    didSet {
      let recognizer = UIPanGestureRecognizer(target: self, action:
                                                #selector(ClassName.pan))
      pannableView.addGestureRecognizer(recognizer)
    }
}
```

Dies ist ein normales Outlet zum UIView der die Geste erkennen soll.

Wir verwenden einen Property Observer wenn das Outlet von iOS gehooked wird.

Gestures

Hinzufügen eines Gesture Recognizers zu einem UIView
Wenn der UIView des Controllers z.B. eine “Pan” Geste erkennen soll.

```
@IBOutlet weak var pannableView: UIView {
   didSet {
     let recognizer = UIPanGestureRecognizer(target: self, action:
                                                    #selector(ClassName.pan))
     pannableView.addGestureRecognizer(recognizer)
   }
}
```

Dies ist ein normales Outlet zum UIView der die Geste erkennen soll.

Wir verwenden einen Property Observer wenn das Outlet von iOS gehooked wird.

Erstellen einer konkreten Subclass von UIGestureRecognizer (für Pans).

Gestures

Hinzufügen eines Gesture Recognizers zu einem UIView
Wenn der UIView des Controllers z.B. eine “Pan” Geste erkennen soll.

```
@IBOutlet weak var pannableView: UIView {
   didSet {
     let recognizer = UIPanGestureRecognizer(target: self, action:
                                                    #selector(ClassName.pan))
     pannableView.addGestureRecognizer(recognizer)
   }
}
```

Dies ist ein normales Outlet zum UIView der die Geste erkennen soll.

Wir verwenden einen Property Observer wenn das Outlet von iOS gehooked wird.

Erstellen einer konkreten Subclass von UIGestureRecognizer (für Pans).

Das Target wird benachrichtigt, wenn die Geste erkannt wird (in diesem Fall, der Controller selbst).

Gestures

Hinzufügen eines Gesture Recognizers zu einem UIView
Wenn der UIView des Controllers z.B. eine “Pan” Geste erkennen soll.

```
@IBOutlet weak var pannableView: UIView {
   didSet {
     let recognizer = UIPanGestureRecognizer(target: self, action:
                                                  #selector(ClassName.pan))
     pannableView.addGestureRecognizer(recognizer)
   }
}
```

Dies ist ein normales Outlet zum UIView der die Geste erkennen soll.

Wir verwenden einen Property Observer wenn das Outlet von iOS gehooked wird.

Erstellen einer konkreten Subclass von UIGestureRecognizer (für Pans).

Das Target wird benachrichtigt, wenn die Geste erkannt wird (in diesem Fall, der Controller selbst).

Die Action ist die Funktion die aufgerufen wird, wenn die Geste erkannt wird.

Gestures

Hinzufügen eines Gesture Recognizers zu einem UIView
Wenn der UIView des Controllers z.B. eine “Pan” Geste erkennen soll.

```
@IBOutlet weak var pannableView: UIView {
   didSet {
     let recognizer = UIPanGestureRecognizer(target: self, action:
                                                    #selector(ClassName.pan))
     pannableView.addGestureRecognizer(recognizer)
   }
}
```

Dies ist ein normales Outlet zum UIView der die Geste erkennen soll.

Wir verwenden einen Property Observer wenn das Outlet von iOS gehooked wird.

Erstellen einer konkreten Subclass von UIGestureRecognizer (für Pans).

Das Target wird benachrichtigt, wenn die Geste erkannt wird (in diesem Fall, der Controller selbst).

Die Action ist die Funktion die aufgerufen wird, wenn die Geste erkannt wird.

Hier wird der Gesture Recognizer hinzugefügt und der UIView erkennt ab jetzt Gesten innerhalb seiner Bounds.

Gestures

Ein Handler für eine Geste braucht Gesten-spezifische Informationen

Jede konkrete Subclass stellt spezielle Funktionen zur Verfügung diesen Typ von Geste zu verarbeiten.

Für UIPanGestureRecognizer sind dies z.B. 3 Funktionen

```
func translationInView(view: UIView) -> CGPoint
```

Kummulativ seit dem Start der Erkennung

```
func velocityInView(view: UIView) -> CGPoint
```

Wie schnell der Finger sich bewegt (in Points/s)

```
func setTranslation(translation: CGPoint, inView: UIView)
```

Dies erlaubt die Translation zu resetten die bisher stattgefunden hat. Durch das resetten zu Null können wir die inkrememtelle statt der kumulativen Distanz erhalten.

Gestures

Die abstrakte Superclass stellt ebenfalls Statusinformationen zur Verfügung

```
var state: UIGestureRecognizerState { get }
```

Dies steht auf `.Possible` bis die Erkennung beginnt.

Für eine diskrete Geste (z.B. Swipe), wechselt dies zu `.Recognized` (Tap ist keine normale diskrete Geste).

Für kontinuierliche Gesten (z.B. ein Pan), durchläuft es von `.Began` über mehrfach `.Changed` bis hin zu `.End` jeden Status.

Es kann ebenfalls zu `.Failed` und `.Cancelled` wechseln.

```
func velocityInView(view: UIView) -> CGPoint
```

Wie schnell der Finger sich bewegt (in Points/s)

```
func setTranslation(translation: CGPoint, inView: UIView)
```

Dies erlaubt die Translation zu resetten die bisher stattgefunden hat. Durch das resetten zu Null können wir die inkrememtelle statt der kumulativen Distanz erhalten.

Gestures

Wie kann ein solcher Pan Handler aussehen?

```
func pan(gesture: UIPanGestureRecognizer) {
   switch gesture.state {
      case .Changed: fallthrough
      case .Ended:
         let translation = gesture.translationInView(pannableView)
         // update von allem was von der Geste abhängt mittels translation.x
         // und .y
         gesture.setTranslation(CGPointZero, inView: pannableView)
      default: break
   }
}
```

Unser action: #selector.

Nur etwas machen, wenn der Finger sich bewegt oder vom Display verschwindet.

Gestures

Wie kann ein solcher Pan Handler aussehen?

```
func pan(gesture: UIPanGestureRecognizer) {
   switch gesture.state {
      case .Changed: fallthrough
      case .Ended:
         let translation = gesture.translationInView(pannableView)
         // update von allem was von der Geste abhängt mittels translation.x
         // und .y
         gesture.setTranslation(CGPointZero, inView: pannableView)
      default: break
   }
}
```

Unser action: #selector.

Nur etwas machen, wenn der Finger sich bewegt oder vom Display verschwindet.

Gestures

Wie kann ein solcher Pan Handler aussehen?

```
func pan(gesture: UIPanGestureRecognizer) {
   switch gesture.state {
      case .Changed: fallthrough
      case .Ended:
         let translation = gesture.translationInView(pannableView)
         // update von allem was von der Geste abhängt mittels translation.x
         // und .y
         gesture.setTranslation(CGPointZero, inView: pannableView)
      default: break
   }
}
```

Unser action: #selector.

Nur etwas machen, wenn der Finger sich bewegt oder vom Display verschwindet.

`fallthrough` führt den Code des darunter liegenden Cases aus.

Gestures

Wie kann ein solcher Pan Handler aussehen?

```
func pan(gesture: UIPanGestureRecognizer) {
   switch gesture.state {
      case .Changed: fallthrough
      case .Ended:
         let translation = gesture.translationInView(pannableView)
         // update von allem was von der Geste abhängt mittels translation.x
         // und .y
         gesture.setTranslation(CGPointZero, inView: pannableView)
      default: break
   }
}
```

Unser action: #selector.

Nur etwas machen, wenn der Finger sich bewegt oder vom Display verschwindet.

`fallthrough` führt den Code des darunter liegenden Cases aus.

Zugriff auf Location des Pans im Koordinatensystem von pannabaleView.

Gestures

Wie kann ein solcher Pan Handler aussehen?

```
func pan(gesture: UIPanGestureRecognizer) {
   switch gesture.state {
      case .Changed: fallthrough
      case .Ended:
         let translation = gesture.translationInView(pannableView)
         // update von allem was von der Geste abhängt mittels translation.x
         // und .y
         gesture.setTranslation(CGPointZero, inView: pannableView)
      default: break
   }
}
```

Unser action: #selector.

Nur etwas machen, wenn der Finger sich bewegt oder vom Display verschwindet.

`fallthrough` führt den Code des darunter liegenden Cases aus.

Zugriff auf Location des Pans im Koordinatensystem von pannabaleView.

Information verarbeiten.

Gestures

Wie kann ein solcher Pan Handler aussehen?

```
func pan(gesture: UIPanGestureRecognizer) {
   switch gesture.state {
      case .Changed: fallthrough
      case .Ended:
         let translation = gesture.translationInView(pannableView)
         // update von allem was von der Geste abhängt mittels translation.x
         // und .y
         gesture.setTranslation(CGPointZero, inView: pannableView)
      default: break
   }
}
```

Unser action: #selector.

Nur etwas machen, wenn der Finger sich bewegt oder vom Display verschwindet.

`fallthrough` führt den Code des darunter liegenden Cases aus.

Zugriff auf Location des Pans im Koordinatensystem von pannabaleView.

Information verarbeiten.

Durch reset der Translation erhalten wir beim nächsten mal wie weit der Pan sich bewegt hat seit dem aktuellen Durchlauf.

Gestures

```
UIPinchGestureRecognizer
   var scale: CGFloat                  // nicht read-only (kann resetten)
   var velocity: CGFloat { get } // Skalierungsfaktor pro Sekunde


UIRotationGestureRecognizer
   var rotation: CGFloat               // nicht read-only (kann resetten)
                                       // in Radian
   var velocity: CGFloat { get } // Skalierungsfaktor pro Sekunde


UISwipeGestureRecognizer
   Richtung und Anzahl Finger setzten, dann .Recognized prüfen
   var direction: UISwipeGestureRecognizerDirection // welche
                                                    // Swipes
   var numberOfTouchesRequired: Int           // Anzahl Finger


UISwipeGestureRecognizer
   Anzahl der Taps und Finger setzten, dann .Ended prüfen
   var numberOfTapsRequired: Int    // Single Tap, Double Tap, etc.
   var numberOfTouchesRequired: Int // Anzahl Finger
```