Mobile Application Development

Lecture 15
UIKit views and controls

Lecture Summary

- View Controller lifecycle
- UIColor, UIFont, NSAttributedString
- UIKit views and controls: UILabel, UIButton, UISlider, UISwitch, UITextField, UITextView
- NSNotificationCenter, keyboard notifications
- DEMO

View Controller Lifecycle

- View controllers receive messages whenever certain events related to the view controller's lifecycle occur

- Every view controller is a subclass of the `UIViewController` class which defines a set of methods that will be executed as the view controller receives lifecycle messages

- Subclasses of `UIViewController` might override such methods to provide specific behavior (the implementation of the superclass must also be called through `super`)

View Controller Lifecycle

1. The first step in the lifecycle is the creation; view controllers can be created
   - through *storyboards*
   - programmatically
2. Next, the outlets of the view controller are set
3. View controllers (their managed views) may appear and disappear from the screen
4. Low-memory notification

- Anytime one of these events occurs, the system sends a message to the view controller

viewDidLoad

- After the view controller has been created and the outlets set, `viewDidLoad` gets invoked
- `viewDidLoad` is where most of the initialization of the view controller can be performed, after the outlets have been set (safe)
- `viewDidLoad` is called only once in the life of a view controller, after it has been created
- At this point all outlets have been set, but bounds property of the view is not set, so it is not safe to perform geometry-based settings in `viewDidLoad`

```
- (void)viewDidLoad{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    // ...
}
```

viewWillAppear

- When the view is about to appear on the screen, `viewWillAppear` gets invoked
- The argument tells whether the view is appearing through an animation or instantly
- This method gets invoked as many times as the view appears on the screen, so it could be invoked multiple times: DO NOT PUT CODE THAT SHOULD BE EXECUTED JUST ONCE, USE `viewDidLoad` TO DO THAT!
- Typically, `viewWillAppear` is used to perform:
  - operations that are related to changes that occurred while the view was not on screen
  - long-running operations that could be unnecessary if the view is never displayed or could block the rendering of the view if executed in `viewDidLoad` (possibly in a separate thread)

```
- (void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];
    // ...
}
```

viewWillDisappear

- When the view is about to go off the screen, `viewWillDisappear` gets invoked
- The argument tells whether the view is disappearing through an animation or instantly
- This method gets invoked as many times as the view disappears from the screen, so it could be invoked multiple times
- Typically, `viewWillDisappear` is used to:
  - save view state for later retrieval (e.g. scroll position in a scrollable view)
  - cleanup resources (memory and processing) that are not necessary and could be brought back up the next time the view appears

```
- (void)viewWillDisappear:(BOOL)animated{
    [super viewWillDisappear:animated];
    // ...
}
```

viewDidAppear and viewDidDisappear

- When the view has appeared on the screen, `viewDidAppear` gets invoked
- The argument tells whether the view has appeared through an animation or instantly
- `viewDidAppear` invoked as many times as the view has come up on the screen, so it could be invoked multiple times
- Conversely, `viewDidDisappear` gets invoked when the view has disappeared from the screen

```
- (void)viewDidAppear:(BOOL)animated{
    [super viewDidAppear:animated];
    // ...
}
```

```
- (void)viewDidDisappear:(BOOL)animated{
    [super viewDidDisappear:animated];
    // ...
}
```

viewWillLayoutSubviews and viewDidLayoutSubviews

- These methods are invoked when the subviews of the view are about to be or have just been laid out
- Between the execution of `viewWillLayoutSubviews` and `viewDidLayoutSubviews`, **autolayout** is performed
- Geometry-related code can be performed here

```
- (void)viewWillLayoutSubviews{
    [super viewWillLayoutSubviews];
    // ...
}
```

```
- (void)viewDidLayoutSubviews{
    [super viewDidLayoutSubviews];
    // ...
}
```

Autorotation

- The view controller is responsible to handle device rotation (must be set in the project's settings file `Info.plist`)
- If the view controller's `shouldAutorotate` method returns YES, then the view contents should rotate

```
- (BOOL)shouldAutorotate{
    return YES;
}
```

- If so, the supported orientations are defined as return value of the `supportedInterfaceOrientations` method
- `supportedInterfaceOrientations` returns a `UIInterfaceOrientationMask`

```
- (NSUInteger)supportedInterfaceOrientations{
    return UIInterfaceOrientationMaskPortrait;
}
```

Autorotation

- Some methods are invoked to notify the view controller about rotation events

```
- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
                                duration:(NSTimeInterval)duration;


- (void)willAnimateRotationToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
                                         duration:(NSTimeInterval)duration;


- (void)didRotateFromInterfaceOrientation:(UIInterfaceOrientation)fromInterfaceOrientation;
```

didReceiveMemoryWarning

- When memory is low, the view controller is notified and the `didReceiveMemoryWarning` method gets invoked
- Should this happen, all unnecessary (and big) resources (in the heap) should be released; to do this, `strong` pointers should be set to `nil`
- Resources allocated in the memory released at this point should be re-creatable
- Good code should avoid releasing big resources at this point, but should do this well in advance

```
- (void)didReceiveMemoryWarning{
    [super didReceiveMemoryWarning];
    // ...
}
```

awakeFromNib and initialization of view controllers

- `init` is not invoked on objects instantiated by the storyboard
- `awakeFromNib` is invoked by any object instantiated by the storyboard (views, subviews, view controllers, ...) before outlets are set (before `viewDidLoad`)
- `awakeFromNib` should have initialization code that could not be executed anywhere else
- UIViewController's designated initializer and awakeFromNib should have the same initialization code:

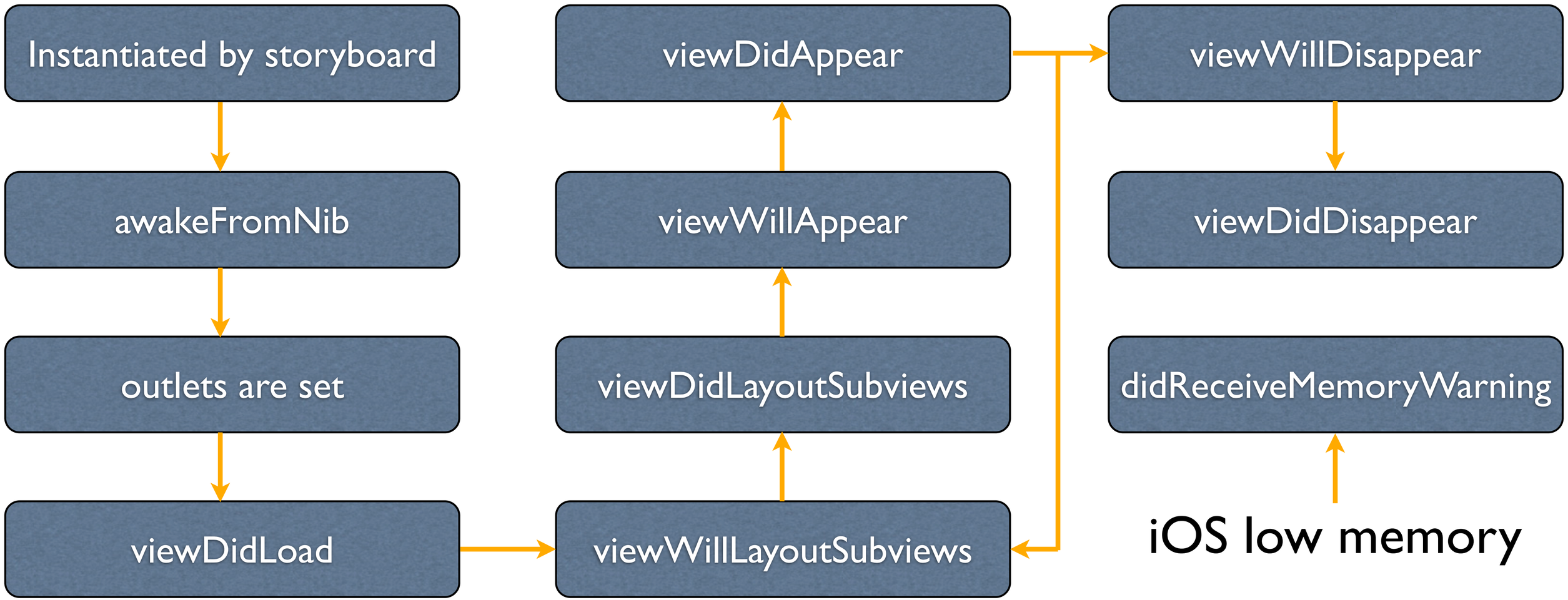
```
- (void)setup{...}

- (void)awakeFromNib{
    [self setup];
}

- (instancetype)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    [self setup];
    return self;
}
```

- `viewDidLoad` is preferrable

View Controller Lifecycle



UIColor

- `UIColor` class represents a color which can be initialized from:
  - RGB (red/gree/blue)
  - HSB (hue/saturation/brightness)
  - from images (patterns)
- Colors handle transparency through an `alpha` property, ranging from 0 to 1
- System colors are provided (`blackColor`, `redColor`, `greenColor`, ...)

Working with fonts

- Fonts can make applications great to see and might hugely improve user experience
- Choosing the right fonts and font sizes is extremely important to make applications good-looking and enjoyable for users
- `UIFont` class represents fonts; the best way to get a font is to ask the OS for the preferred font for a certain style of text (e.g. `UIFontTextStyleHeadline`, `UIFontTextStyleBody`, ...)

```
UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleHeadline];
```

- System fonts are used for buttons, not for content

NSAttributedString

- `NSAttributedString` object manages character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string
- Attributes (such as color, stroke width, ...) apply to portions (ranges) of the string
- Key/value pairs (dictionaries of attributes) are assigned to a certain range of a string
- Attributed strings allow developers to render styled text
- The UIKit framework adds methods to `NSAttributedString` to support the drawing of styled strings and to compute the size and metrics of a string prior to drawing
- `NSAttributedString` is used to render fonts on the screen
- `NSAttributedString` is not a subclass of `NSString`; it is possible however to get a `NSString` from a `NSAttributedString` with the `string` method

NSAttributedString

- `NSAttributedString` are typically created from a simple (unattributed) `NSString`, an existing `NSAttributedString`, or a `NSString` and a `NSDictionary` of attributes

```
- (id)initWithString:(NSString *)aString

- (id)initWithAttributedString:(NSAttributedString *)attributedString

- (id)initWithString:(NSString *)aString attributes:(NSDictionary *)attributes
```

NSAttributedString

- **NSAttributedString** are typically created from a simple (unattributed) **NSString**, an existing **NSAttributedString**, or a **NSString** and a **NSDictionary** of attributes

```
- (id)initWithString:(NSString *)aString

- (id)initWithAttributedString:(NSAttributedString *)attributedString

- (id)initWithString:(NSString *)aString attributes:(NSDictionary *)attributes
```

- A mutable version of **NSAttributedString**, named **NSMutableAttributedString**, allows you to change dynamically the characters and attributes of the string at runtime through the methods:

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)aRange

- (void)removeAttribute:(NSString *)name range:(NSRange)aRange

- (void)setAttributes:(NSDictionary *)attributes range:(NSRange)aRange
```

NSAttributedString

- Attributes that can be set are:

Attribute	Value type
NSFontAttributeName	UIFont*
NSForegroundColorAttributeName	UIColor*
NSBackgroundColorAttributeName	UIColor*
NSStrokeWidthAttributeName	NSNumber*
NSStrokeColorAttributeName	UIColor*

UIKit Views

- The `UIView` class defines a rectangular area on the screen and the interfaces for managing the content in that area
- Views can embed other views and create sophisticated visual hierarchies, creating a parent-child relationship between the view and its subviews
- The geometry of a view is defined by some properties:
  - `frame`: origin and dimensions of the view in the coordinate system of its superview
  - `bounds`: the internal dimensions of the view as it sees them
  - `center`: the coordinates of the center point of the rectangular area covered by the view
- All UI elements inherit from `UIView`

https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#//apple_ref/doc/uid/TP40012857

UIKit Views

- The following are the most commonly used `UIView` properties:

Property	Value type	Description
`frame`	`CGRect`	Frame rectangle describing the view's location and size in the superview's coordinate system
`bounds`	`CGRect`	Bounds rectangle describing the view's location and size in its own coordinate system
`center`	`CGPoint`	Center of the frame
`backgroundColor`	`UIColor*`	Background color; defaults to `nil` (transparent)
`alpha`	`CGFloat`	0.0 means transparent and 1.0 means opaque
`hidden`	`BOOL`	YES means the view is invisible, NO means visible
`userInteractionEnabled`	`BOOL`	NO means user events are ignored

UILabel

- `UILabel` objects are used to display static text on a fixed (by you) number of lines
- `UILabel` has the following properties to work with the text to display:

Property	Value type	Description
`text`	`NSString*`	Text being displayed
`font`	`UIFont*`	Font of the text
`textColor`	`UIColor*`	Color of the text
`textAlignment`	`NSTextAlignment`	Alignment of the text (NSTextAlignmentLeft, NSTextAlignmentRight, NSTextAlignmentCenter...)
`attributedText`	`NSAttributedString*`	Styled text being displayed
`numberOfLines`	`NSInteger`	Maximum number of lines to use

UIKit Controls

- A control is a communication tool between a user and an app
- Controls convey a particular action or intention to the app through user interaction
- Controls can be used to manipulate content, provide user input, navigate within an app, or execute other pre-defined actions
- The `UIControl` class (subclass of `UIView`) is the base class for all controls
- `UIControl` is never used, but its subclasses, such as `UIButton` and `UISlider`, are used instead
- Typical controls that can be used in iOS:
  - **Buttons**
  - **Date Pickers**
  - **Page Controls**
  - **Segmented Controls**
  - **Text Fields**
  - **Sliders**
  - **Steppers**
  - **Switches**

Control states

- A **control state** describes the current interactive state of a control:
  - normal (enabled but not selected or highlighted)
  - selected (e.g. in UISegmentedControl) ⟶ 

  - disabled
  - highlighted (when a touch enters and exits during tracking and when there is a touch up event)
- The control state changes as the user interacts with the control
- Specific behavior and appearance can be specified for each control state

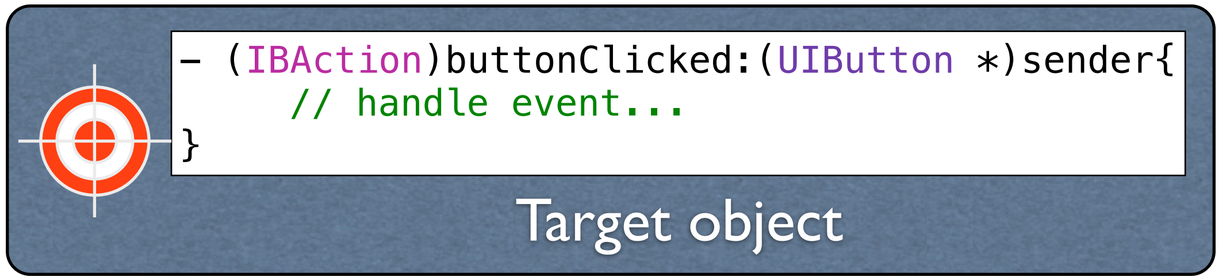
Control events

- **Control events** represent the ways (physical gestures) that users can make on controls
- Typical control events:
  - `UIControlEventTouchDown`: touch down inside a control
  - `UIControlEventTouchDownRepeat`: repeated touch down
  - `UIControlEventTouchDragInside`: a finger is dragged inside the bounds of the control
  - `UIControlEventTouchDragOutside`: a finger is dragged outside the bounds of the control
  - `UIControlEventTouchDragEnter`: a finger is dragged into the bounds of the control
  - `UIControlEventTouchDragExit`: a finger is dragged from within a control to outside its bounds
  - `UIControlEventTouchUpInside`: a finger is lifted when inside the bounds of the control (typical for `UIButton`)
  - `UIControlEventTouchUpOutside`: a finger is lifted when outside the bounds of the control
  - `UIControlEventTouchCancel`: system event canceling the current touches for the control
  - `UIControlEventValueChanged`: touch dragging or otherwise manipulating a control, causing a series of different values

Target-action

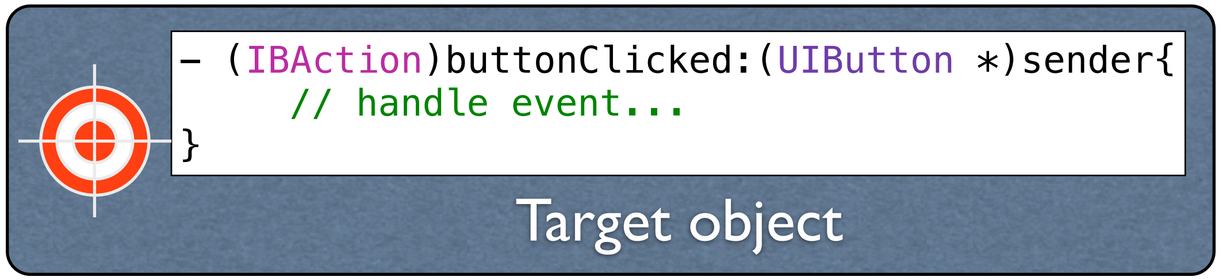
- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event

Button



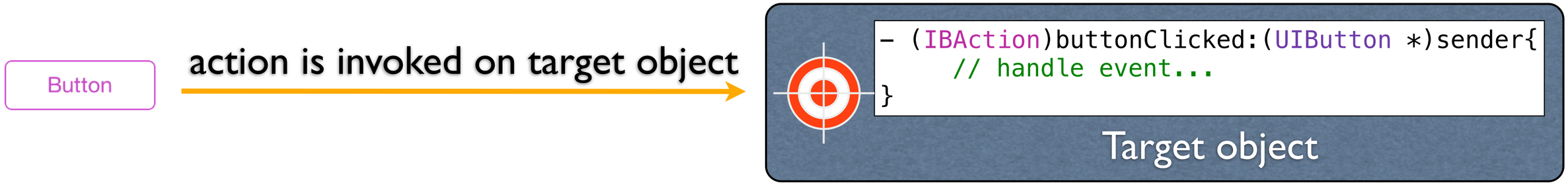
Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event



Target-action

- The **target-action** mechanism is a model for configuring a control to send an action message to a target object after a specific control event



Target-action binding

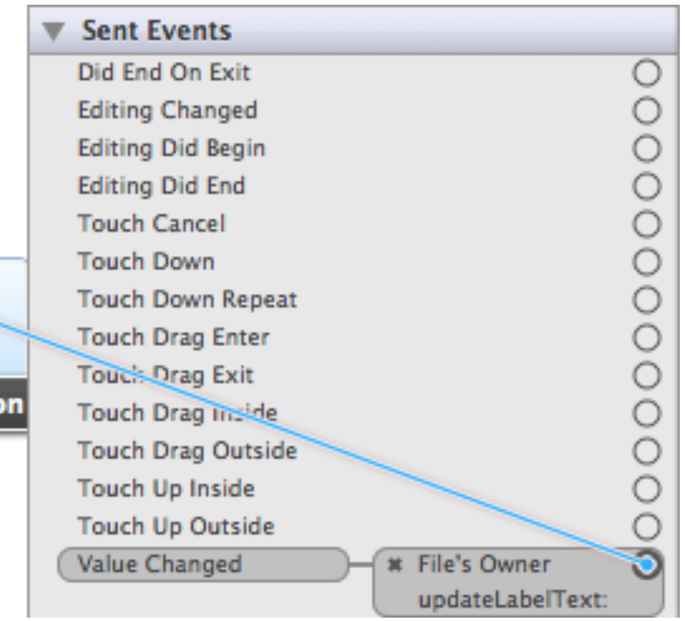
- Binding a target-action to a control event:

1. programmatically

```
[self.mySlider addTarget:self action:@selector(myAction:) forControlEvents:UIControlEventValueChanged];
```

2. with the Connection Inspector in Interface Builder to Control-drag the slider's Value Changed event to the action method in the target file

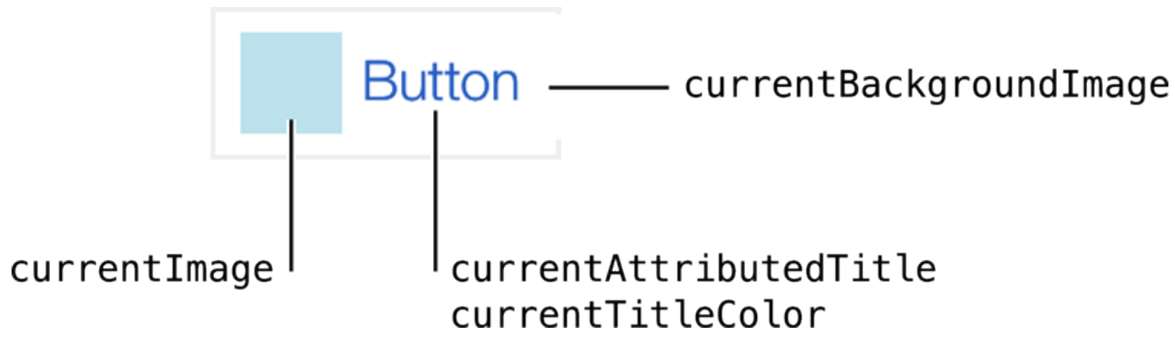
```
- (IBAction)updateLabelText:(UISlider *)
    sender {
}
```



3. Control-click the slider in Interface Builder, and drag its Value Changed event to the target object in your Storyboard and select the appropriate action from the list of actions available for the target

UIButton

- Buttons let a user initiate behavior with a tap
- Buttons display textual or image content
- When the users taps on a button, it changes its state to highlighted and its appearance accordingly
- If an action has been bound for a certain event, a button will trigger the execution of that action each time the event occurs
- The current appearance of button can be retrieved with some read-only properties



UIButton

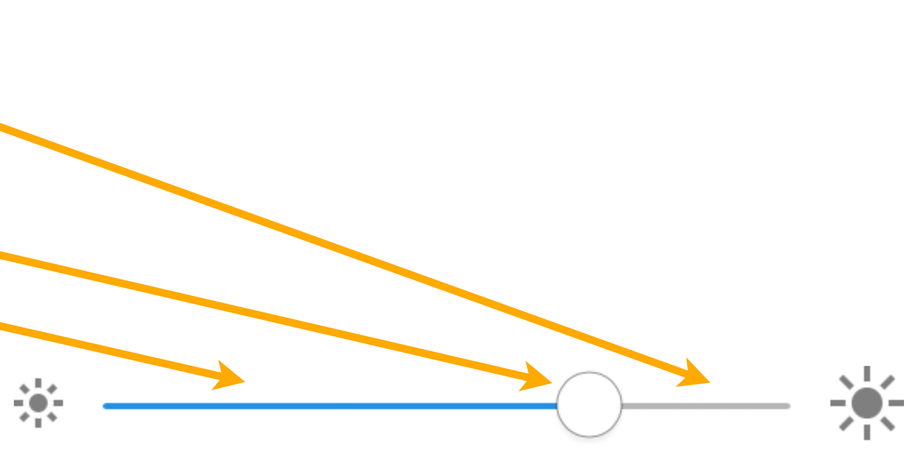
- It is possible to set the title, image, and background image of a button for each state

1. in the attributes inspector
2. programmatically

```
- (void)setTitle:(NSString *)title forState:(UIControlState)state
- (void)setAttributedTitle:(NSAttributedString *)title forState:(UIControlState)state
- (void)setImage:(UIImage *)image forState:(UIControlState)state
- (void)setBackgroundImage:(UIImage *)image forState:(UIControlState)state
```

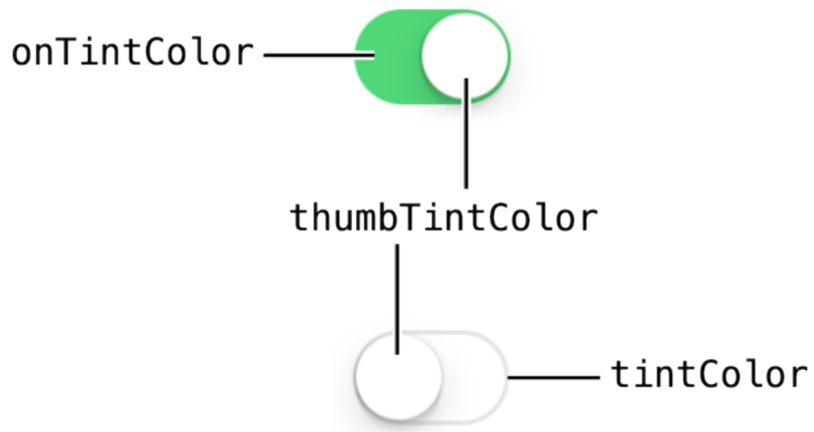
UISlider

- Sliders enable users to interactively modify some adjustable value in an app
- It is possible to configure a minimum, maximum, and current value for your slider with the properties `minimumValue`, `maximumValue`, and `value`, respectively
- Default values are minimum = 0, maximum = 1, and current value = 0.5
- It is possible to change the tint of the slider with the properties
  - `maximumTrackTintColor`
  - `thumbTintColor`
  - `minimumTrackTintColor`



UISwitch

- A switch lets the user turn an option on and off
- The appearance of a switch can be customized by setting the appropriate properties



- The boolean property **on** is used to set the off/on state of the switch

UITextField

- Text fields allows the user to input a single line of text into an app
- You can set and retrieve the value of the input text with the `text` and `attributedText` properties
- The `placeholder` and `attributedPlaceholder` properties are used to set a placeholder text in the field
- Text can also be styled using `font`, `textAlignment`, `textColor` properties
- The clear button on the right is displayed by default; it is possible to configure whether it should be present or not by using the property `clearButtonMode` with a `UITextFieldViewMode`
- The keyboard style and layout can also be set (`UITextInputTraits`)

Keyboard management

- Tapping a `UITextField` causes the keyboard to appear
- The keyboard slides in from the bottom of the screen and covers a certain area of the view
- The keyboard becomes the first responder
- It is necessary to handle the appearance of the keyboard properly:
  - it might be necessary to slide the view content up so that it does not get covered by the keyboard (more on this later)
  - it is necessary to make the keyboard disappear when done using it (tapping “done” won;t make it go away)
- The trick is to set the background view’s class to be UIControl instead of UIView so that it can receive tap events; when the background view is tapped, a target-action method is invoked and that’s where the keyboard can be dismissed by using:

```
[textField resignFirstResponder];
```

UITextView

- More powerful way to display text:
  - multiple lines, editable and selectable, scrollable
- Set the text through two properties:
  - `text`: `NSString` for normal text; properties that can affect the text style are `font` (`UIFont`), `textColor` (`UIColor`), and `textAlignment` (`NSTextAlignment`)
  - `attributedText`: `NSAttributedString` for styled text
- Efficient way to manipulate text:
  - `textStorage`: `NSTextStorage` allows you to change the styled text and it will automatically update the view (since iOS7)

UITextView

- `UITextView` has many following notable properties:

Property	Value type	Description
`text`	`NSString*`	Text being displayed
`font`	`UIFont*`	Font of the text
`textColor`	`UIColor*`	Color of the text
`textAlignment`	`NSTextAlignment`	Alignment of the text (NSTextAlignmentLeft, NSTextAlignmentRight, NSTextAlignmentCenter...)
`attributedText`	`NSAttributedString*`	Styled text being displayed
`textStorage`	`NSTextStorage`	Efficient text manipulation
`editable`	`BOOL`	Whether the receiver is editable
`selectable`	`BOOL`	Whether the receiver is selectable
`selectedRange`	`NSRange`	Current selection range in the text view

UITextViewDelegate

- `UITextViewDelegate` is a protocol that defines a set of optional methods can be used to receive editing-related messages for a certain `UITextView`
- It is possible to set a delegate for a `UITextView` object with the `delegate` property
- Protocol methods:

```
- (BOOL)textViewShouldBeginEditing:(UITextView *)textView
- (BOOL)textViewShouldEndEditing:(UITextView *)textView
- (void)textViewDidBeginEditing:(UITextView *)textView
- (void)textViewDidChange:(UITextView *)textView
- (void)textViewDidChangeSelection:(UITextView *)textView
- (void)textViewDidEndEditing:(UITextView *)textView
```

Notifications

- iOS provides another way for object interaction, other then message passing
- Notifications are the standard way by which the model can notify the controller of certain event
- **`NSNotificationCenter`** is a class that provides a mechanism for broadcasting information within a program
- A reference to a **`NSNotificationCenter`** instance is retrieved in the following way:

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
```

- Each program has its own **`NSNotificationCenter`**, so it does not need to be created
- Objects can register with a notification center to receive notifications and upon receiving such notification can execute a particular method
- It is important to unregister for notifications when no longer needed to avoid possible crashes
- Typically, register when view appears and unregister when view disappears; if notifications should be received when the view is off screen, unregister in **`dealloc`** (rare)

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

object that is listening for notifications

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

selector to execute when receiving the notification

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

name of notification

Registering for Notifications

- Objects can register with the **`NSNotificationCenter`** for certain notifications with the method:

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

source of notifications (nil if any object)

Registering for Notifications

- For instance:

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(aMethod)
                                             name:UIKeyboardWillShowNotification
                                           object:nil];
```

Unregistering for Notifications

- When an objects is no longer interested in notifications, it can unregister with the `NSNotificationCenter` with the method:

```
- (void)removeObserver:(id)notificationObserver
                  name:(NSString *)notificationName
                object:(id)notificationSender
```

- For instance:

```
[[NSNotificationCenter defaultCenter] removeObserver:self
                                                name:UIKeyboardWillShowNotification
                                              object:nil];
```

Generating Notifications

- An object that needs to generate a notification can post the notification to the `NSNotificationCenter` with the method:

```
- (void)postNotificationName:(NSString *)notificationName
                      object:(id)notificationSender
                    userInfo:(NSDictionary *)userInfo
```

- For instance:

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"MyNotification"
                                                    object:self
                                                  userInfo:nil];
```

Keyboard notifications

- When the system shows or hides the keyboard, it posts several keyboard notifications
- Notifications contain information about the keyboard, such as its size
- Registering for these notifications is the only way to get some types of information about the keyboard
- System notifications for keyboard-related events (names are similar to delegate methods, but these are notifications, which implies a different communication paradigm):
  - `UIKeyboardWillShowNotification`
  - `UIKeyboardDidShowNotification`
  - `UIKeyboardWillHideNotification`
  - `UIKeyboardDidHideNotification`
- The selectors related to these events should be responsible to move the view to make the content visible when the view keyboard appears, and to reposition it when it disappears

Mobile Application Development

Lecture 15
UIKit views and controls