Alert Views

Alert Views

- Alert views are an easy way to display concise and informative information to the user.
- The kind of UI that is displayed in a UI Alert Controller is specified by the controller's `preferred` style when creating the controller
- You customize the UI by identifying what buttons or text fields you want to include

Key classes

The primary classes used in an Alert are:

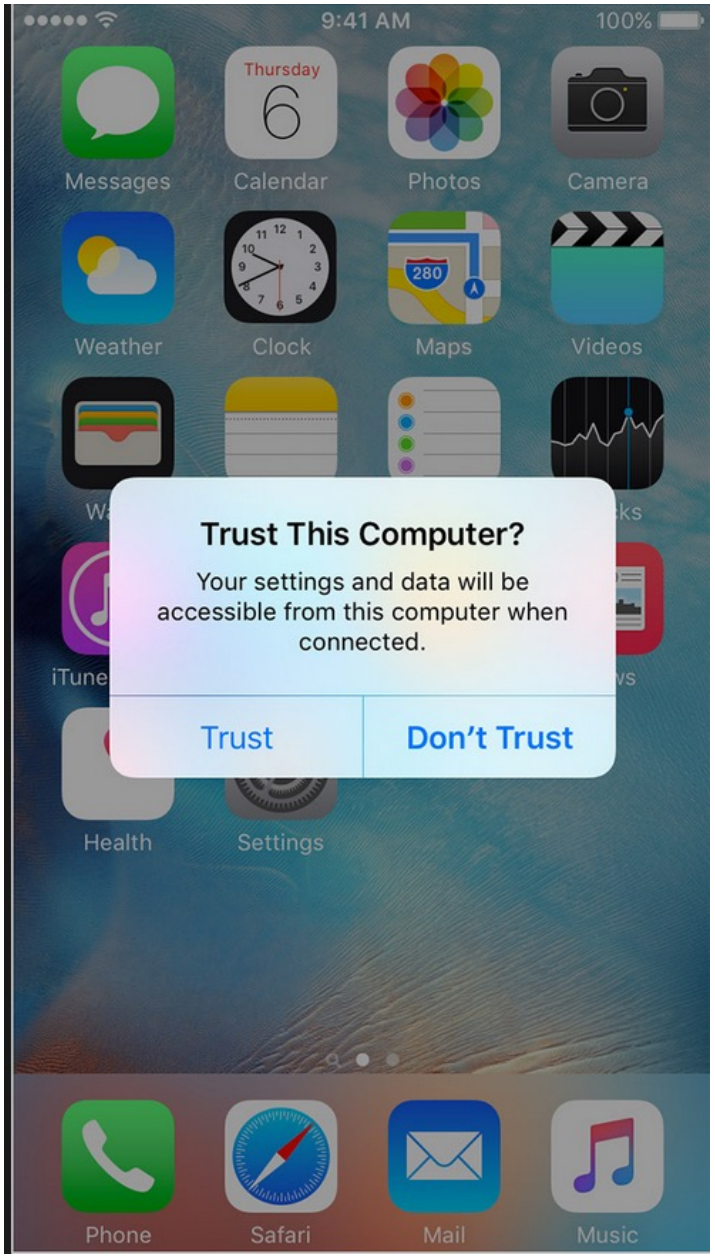
- `UIAlertController`
  is a VC that displays an alert message to the user
- `UIAlertAction`
  represents an action that can be taken when tapping a button in an alert

You create a `UIAlertController` object first, and then add as many `UIAlertAction` objects as needed, typically based on the number of buttons defined.

`UIAlertController` Style Settings

Alert: a UI that displays over and grays out the current UI.

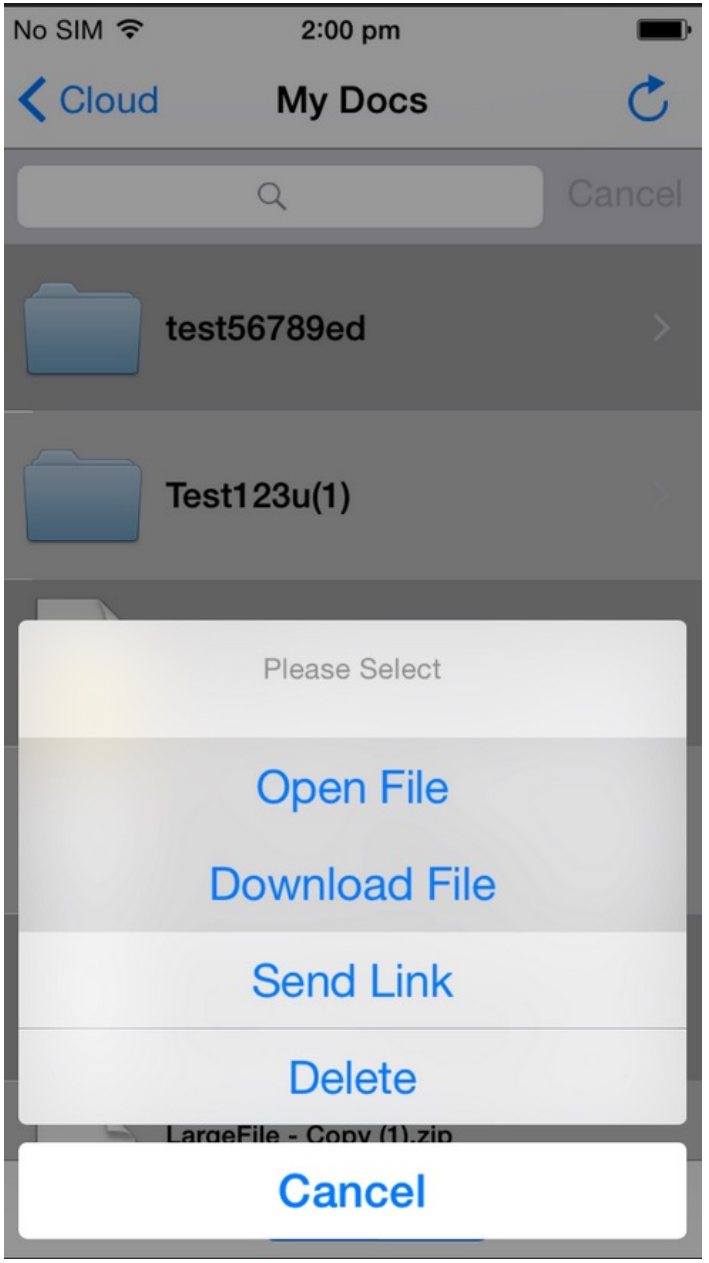
- "Trust" and "Don't Trust" are the two `UIAlertAction` objects.



`UIAlertController` Style Settings

Action Sheet: a UI that slides up from the bottom of the screen and grays out the current UI.

- In this example, there are five `UIAlertAction` objects.



`UIAlertAction`

- A `UIAlertAction` represents an action that can be taken when tapping a button in an alert
- You use this class to configure information about a single action, including
  - The *title* to display in the button
  - Any *style* information
  - A *handler* to execute when the user taps the button

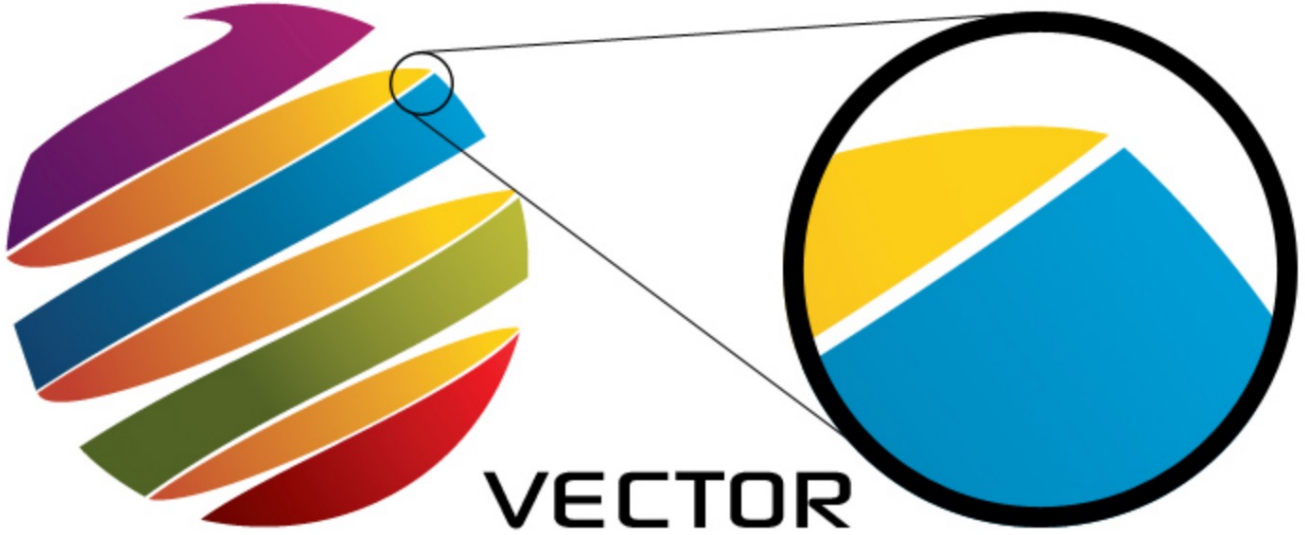
`UIAlertAction` Style Settings

- *Default*:
  - Apply the default style to the action's button
  - Normal text
- *Cancel*:
  - Apply a style that indicate the action cancels the operation and leaves things unchanged
  - Can only have one of these. (App crashes if you define more than one for a given button)
  - Bold text
- *Destructive*:
  - Apply a style that indicates the action might change or delete data
  - Red text color

2D Graphics

Vector Graphics

- In *vector graphics*, a graphical object is defined using geometric primitives such as points, lines, curves, and shapes or polygons based on mathematical expressions.
- This means when you want to draw a line, for example, you define the starting and ending point of that line in a coordinate space and let the rendering engine draw it.



Core Graphics

Core Graphics is Apple's drawing framework. It covers:

- declaration of basic geometric shapes, such as points, sizes, vectors, and rectangles
- functions that render the pixels onto the screen
- everything in between

Example: `CGRect`: we saw this structure before when we talked about a view's frame and bounds. CG stands for "Core Graphics".

Core Graphics is a vector drawing framework.

- It was previously known as "Quartz" or "Quartz 2D".
- It was originally built on top of the open high-level API OpenGL.
- As of iOS 9, it's built on top of Apple's low-level API Metal.

Graphics Context

A *graphics context* serves as the “canvas” you’re drawing on.

- It identifies the current drawing destination (screen, printer, file, etc.), the coordinate system, and any graphics attributes associated with the destination.
- It maintains global information and settings about the current draw environment:
    - current fill and stroke colors
    - line width and pattern
    - line cap and join (miter) styles
    - alpha (transparency)
    - antialiasing and blend mode
    - shadows
    - text attributes (font, size, etc.)

Graphics Context (cont.)

- It acts like a buffer for accumulating subsequent drawing operations.

In iOS, each UIView has a graphics context, and all drawing for the view renders into this context before being transferred to the device's hardware.

UIView Methods

Two important methods associated with the UIView class:

`draw()`

- It contains your custom drawing code.

`setNeedsDisplay()`

- Call this whenever you change something that affects what's drawn in `draw()`, like a view's frame or background color.
- It causes `draw()` to be called.
- The request to draw gets queued in the main queue.

`draw()`

`draw()` is automatically called whenever:

- The view is new to the screen.
- Other views on top of it are moved.
- The view's "hidden" property is changed.
- Your app explicitly calls the `setNeedsDisplay()` method on the view.

When a view's `draw()` method is executed, it renders the view into the appropriate context. You can override `draw()` for custom rendering.

`setNeedsDisplay()`

Never call `draw()` directly.

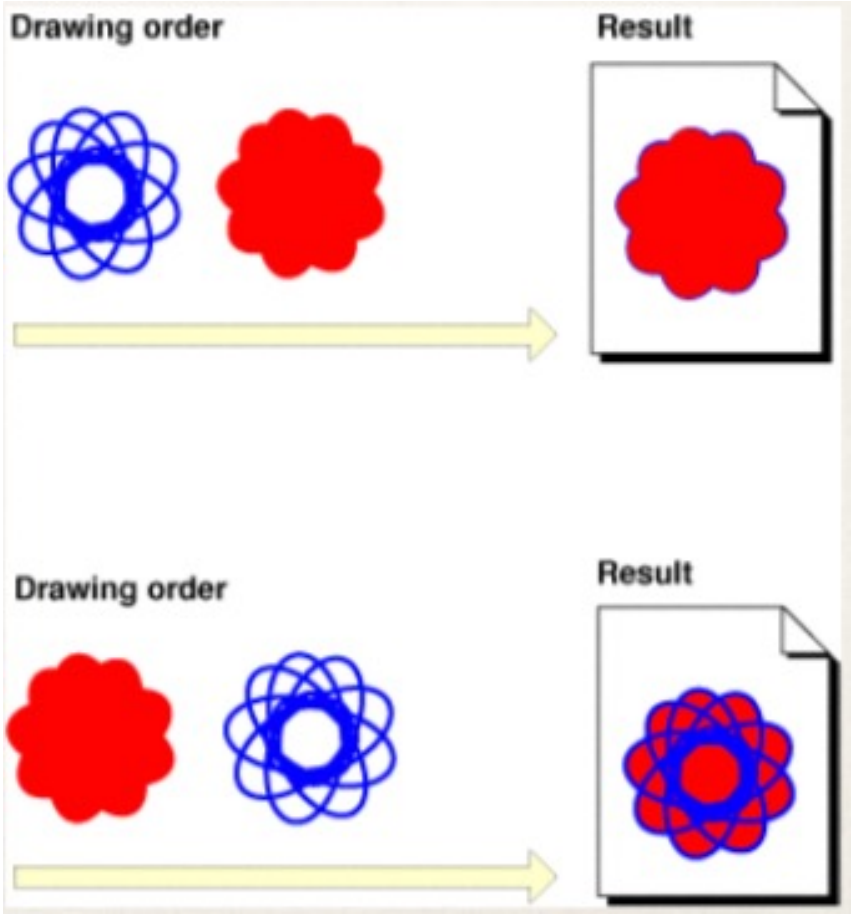
- If you need to update your view, call `setNeedsDisplay()` on the view.
- `setNeedsDisplay()` does not itself call `draw()`, but it flags the view as 'dirty', triggering a redraw using `draw()` on the next screen update cycle.
- Note that even if you call `setNeedsDisplay()` five times in the same method, you'll only ever actually call `draw()` once.

Order of `draw()` calls

Order matters when drawing in Core Graphics!

Pixels cannot be changed once they're "painted"

You must draw over existing pixels with new `draw()` commands



UIView

Whenever you want to do some custom drawing, all you have to do is:

- Create a `UIView` subclass
- Get the view's current context:

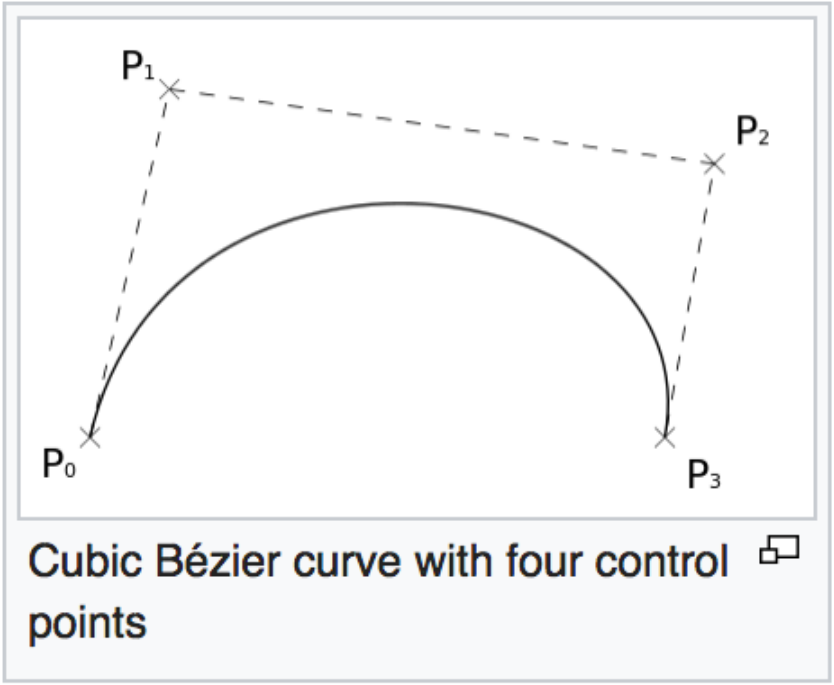
```
let context = UIGraphicsGetCurrentContext()
```

- Override the `draw` method and add your Core Graphics drawing code to paint pixels into `context`

Bézier Curves

A *Bézier curve* is a parametric curve based on Bernstein polynomials.

In vector graphics, Bézier curves are used to model smooth curves that can be scaled indefinitely.



Cubic Bézier curve with four control points

In iOS, a Bézier path is an object of class `UIBezierPath`.

- They allow for custom geometric paths and drawing properties within Core Graphics
- They can be defined as lines, ovals, rectangles, and arbitrary freeform paths
- They are also used for clipping and intersection testing

UIBezierPath

Line segment:

```
path = UIBezierPath()
path.move(to: CGPoint.myPoint1)
path.addLine(to: CGPoint.myPoint2)
```

Arc of a circle:

```
let path = UIBezierPath(arcCenter: center,
                        radius: myRadius,
                        startAngle: angle1,
                        endAngle: angle2,
                        clockwise: true)
```

Then draw it:

```
path.stroke()
```

Transformations

Translate coordinate system origin to (tx, ty):

```
CGContextTranslateCTM(c:CGContext?,
      tx:CGFloat, ty:CGFloat)
```

Scale coordinate system by sx and sy:

```
CGContextScaleCTM(c:CGContext?, sx:CGFloat,
      sy:CGFloat)
```

Rotate coordinate system by angle (in radians):

```
CGContextRotateCTM(c:CGContext?
      angle:CGFloat)
```

Changing Contexts

It's often beneficial to save the context before performing a series of graphics operations, and to restore the context afterwards.

- This isolates any changes in settings you may make while performing the operations to only those operations.
- In particular, when performing transformations, this preserves the context's coordinate system.

```
CGContextSaveGState(context)
CGContextRestoreGState(context)
```

These operations behave like a "push" and a "pop": they save and restore the context using a stack.

Dynamic changes to Interface Builder

There are two attributes that enable views to be dynamically updated in Interface Builder:

`@IBDesignable:` Specifies that objects of a class declaration should have their display refreshed whenever the object is changed by the user.

`@IBInspectable:` Specifies that there should be an interface that allows the user to change values of this object in Interface Builder.

The `@` character is used in Swift to indicate an *attribute*: additional information to be given to the compiler.