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.
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.)
• 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.
Views

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

- **Create a UIView subclass**
- **Get the view’s current context:**
  ```swift
  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.

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
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.
Calendar and EventKit
Event Kit

*Event Kit* is a set of classes for accessing and manipulating a user’s calendar events and reminders, which live in the *Event Store* database on a device.

You can, among other things:

- Create a calendar
- Delete a calendar
- Get a list of calendars
- Get the attributes of a given calendar
- Create an event
- Modify an event
- Delete an event
At the heart of EventKit is the class `EKEventStore`.

An instance of `EKEventStore` provides access to an API for performing read and write operations on the user’s calendars and reminder lists.

```swift
let eventStore = EKEventStore()
```
Event Kit Authorization

Your app must ask for permission to access the calendars and/or reminders.

• Check to see if your app is authorized:

    authorizationStatus(
        for entityType:EKEntityType) ->
    EKAAuthorizationStatus

type EntityType: either .event or .reminder

returns EKAAuthorizationStatus:
    .authorized
    .denied
    .notDetermined
    .restricted
If your app isn’t authorized, you must request access.

```swift
requestAccess(
    to entityType:EKEntityType,
    completion: <completion handler>)
```

- entityType: either `EKEntityType`.
- completion: code to execute when the request completes.

Your app is not blocked while the user decides.

The completion handler executes regardless of what the user’s choice was.

Note that the user can change the calendar access state at any time. Consequently, include this code in `viewWillAppear` to make sure that the current state of authorization is used each time the user sees the application interface.
Event Kit

To use Event Kit:

• import EventKit

• Create an instance of EKEventStore

• Through the EKEventStore object:
  • Verify that your app has permission to access the event store
  • Include handling if you don’t have access

• Read and write calendars / events from and to the event store
To check to see if your app is authorized to access the event store:

```swift
if (EKEventStore.authorizationStatus(for: .event) != EKAuthorizationStatus.authorized) {
    // handle error
} else {
    // do stuff
}
```
If the status returned is **Authorized**, you can start reading and writing from or to the Event Store.

If the status returned is **NotDetermined** (as in the first execution), then ask the user for access to the calendars:

```swift
eventStore.requestAccess(to: .event,
    completion: { (accessGranted: Bool, error: NSError?) in
        if accessGranted == true {
            <we can access the event store>
        } else {
            <help the user give you access>
        }
    })
```
Once you’ve been given access to the calendars, you can get a list of them:

```swift
eventStore.calendarsForEntityType(EKEntityType.Event)
```

This returns an array of `EKCalendar` objects.
Managing Calendars

Creating calendars:
- Create an `EKCalendar` object.
- Set various attributes.
- After saving, store the key associated with that calendar.

Deleting a calendar:
- Get the calendar to delete using the stored key.
- Remove the calendar.

Creating events:
- Get the calendar you want to add an event to.
- Create an `EKEvent` object.
- Set various attributes.
- Save.
Events

To create an event:

- **create an instance of `EKEvent` for the appropriate `eventStore`:**

  ```swift
  let event = EKEvent(eventStore:eventStore)
  ```

- **set the properties of the event:**

  ```swift
  event.title = "UT vs. Oklahoma"
  event.startDate = Date("2019-10-12")
  event.calendar = calendarKey
  ```
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.
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 indicates 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
Storage
There are several ways to *persist* data – meaning to retain information across multiple invocations of your app. Essentially, this means storing data to the hard drive of your phone.

- User Defaults
- File Manager
- Keychain
- Core Data
User Defaults

- Typically used for simple user preferences, or to store simple values that improve the user's experience, such as the number of times the app has been launched, or when the user first launched the app.

- It's not encouraged to store large amounts of data, images, or other large data structures. Booleans, numbers, and strings are usually okay because they're small and very well-suited for key-value storage.

- Never store sensitive data in User Defaults, because it's not encrypted, and space is shared across apps.

- Reading from User Defaults is fast, because the data is automatically loaded into memory at app launch.
The file system typically has a large amount of storage available, but it can be relatively slow to read or write large amounts of data. If you need something quicker, you should consider SQLite or in-memory storage.

Disk storage is typically used for larger amounts of data than User Defaults, such as:

- images, videos, or large JSON files: binary data that has no properties that you might want to query
- text files or PDFs
- data structures that you might make conform to Codable so they can be easily converted to Data, which can then be written to a file on the file system.
Disk storage is not as secure as you might think. Apple encrypts devices when they are locked, but as soon as the device is unlocked and your app becomes active, the encryption no longer applies. Consequently, you should avoid storing sensitive data on disk.
The keychain is the place where iOS stores all kinds of sensitive user data, such as usernames, passwords, certificates, OAuth tokens, etc. It is probably the most secure place for storing data.

The keychain APIs are very non-trivial to navigate and use. Fortunately, there are frameworks available on CocoaPods, Carthage, and SwiftPM that make it much easier.
Databases

- Databases are very good at efficiently storing, retrieving, and filtering large amounts of structured data.

- Databases are stored on the device, often in the Documents directory. You should consider any data written to databases as compromised unless you provide your own encryption.

- There are several third-party databases available, but the framework supported by Apple is Core Data.
Combinations

- If you have large objects of structured data, such as a user's to-do list, metadata for documents created by the user, weekly blog entries, etc., a database can be very useful.

- If any of these objects have large collections of binary data attached to them (such as images), a combination of disk storage and database storage could be used by writing the images to disk, and storing the path to the images in the database.
User Defaults
User Defaults

- User defaults are a way to store simple key/value pairs
  - Similar to a dictionary, but with persistence
  - Useful for **small** amounts of data (<100 KB)

- Very easy mechanism, but limited capabilities

- Typically used for application configuration data

- Data types you can store in User Defaults:
  - NSData
  - NSString; String
  - NSNumber; UInt/Int/Float/Double/Bool
  - NSDate
  - NSArray; Array
  - NSDictionary; Dictionary
You use the **singleton object** `UserDefaults.standard` to access the user defaults store.

1. **Associate the singleton object with a variable:**

   ```swift
   let defaults = UserDefaults.standard
   ```

2. **Call a method to save the data you want to store, providing key and value**

   ```swift
   defaults.set(<value>, forKey:<keyName>)
   
   <value> is the data you want to store (most any type)
   <keyName> is a String that you want to use to identify the data
   ```
// Define keys for the values to store
let kUserIdKey = "userId"
let kTotalKey = "total"
let kNameKey = "name"

// Define the values to store
let userId = 900
let total = 1275.55
let name = "University of Texas"

// Get a reference to the global user defaults object
let defaults = UserDefaults.standard

// Store various values
defaults.set(userId, forKey: kUserIdKey)
defaults.set(total, forKey: kTotalKey)
defaults.set(name, forKey: kNameKey)
Reading From User Defaults

There are several convenience methods for retrieving values from User Defaults:

```swift
func defaults.integer(forKey: <keyName>)
func defaults.double(forKey: <keyName>)
func defaults.string(forKey: <keyName>)

etc.
```
// Retrieve the previously stored values
let retrievedUserId = 
defaults.integer(forKey: kUserIdKey)

let retrievedTotal = 
defaults.double(forKey: kTotalKey)

let retrievedName = 
defaults.string(forKey: kNameKey)
Removing From User Defaults

// Remove the objects from User Defaults
defaults.removeObject(forKey: kUserIdKey)

defaults.removeObject(forKey: kTotalKey)

defaults.removeObject(forKey: kNameKey)
Some caveats:

- The methods that return Bool, Int, Float, and Double do not return Optionals. In those cases, they return sentinel values appropriate to their types which are false (or NO), 0, 0.0, and 0.0 respectively.

- Methods that return AnyObject indicates you’ll need to cast to the correct type before using
Core Data
Core Data

- One of the ways to persist data in an iOS app
- Some consider it the best for non-trivial storage
- Makes it simple and fast to work with large amounts of data (> 100KB)
- It's technically not a true "database", but you can do database-like stuff
- Multiple choices for the "backing store":
  - Sqlite (default)
  - XML
  - Binary
Core Data is an iOS framework that provides powerful functionality to query and persist non-trivial data.

- It lets developers store (or retrieve) data in a database in an object-oriented way.

- With Core Data, you can easily map the objects in your apps to the table records in the database without even knowing any SQL.

- That said, it is much more than a storage mechanism:
  - Manages data object life cycles
  - Tracks changes to the data
  - Effortless undo support
  - Saves data to disk
Core Data Architectural Overview

Managed Object Model

- Defines the structure of the data – the data schema (data type, relationships)
- Use the Data Model Design Tool in Xcode to define the models: it’s a way of creating an object graph backed by a database.

Managed Object Context

- Acts as a temporary scratch space
- Objects fetched from the persistent store are stored in the context, where we can manipulate them
- Changes are monitored here
Core Data Architectural Overview: Entities

*Entities* are instances of the data models.

- Example: an Employee or a Company.

- In a relational database, an entity corresponds to a *table*.
  (Think of a table like an Excel spreadsheet, with columns and rows.)

In Core Data, an entity is a *class definition*. 
Core Data Architectural Overview: Attributes

Attributes are values stored in the Entities.

- Example: an Employee entity could have attributes for the employee’s name, position and salary.
- In a relational database, an attribute corresponds to a particular field in a table.

In Core Data, an attribute is a *property* representing a piece of information attached to a particular entity.
Relationships are connections between the Entities.

- Relationships between two entities are called *to-one* relationships.
- Relationships between one and many entities are called *to-many* relationships.
- Example: A Manager can have a to-many relationship with a set of employees, whereas an individual Employee will have a to-one relationship with his manager.

In Core Data, a relationship is a link between multiple entities. It’s a *property* that points to other entities in the database.
Haptics
**Haptics**

*Haptics* is a term derived from the Greek word "haptikos" which means "able to come into contact with".

In computer science, it is defined as the use of technology that stimulates the senses of touch and motion, especially to reproduce the sensations that would be felt by a user interacting directly with physical objects.

Haptics are used to engage people's sense of touch to enhance the experience of interacting with an on-screen interface, such as scrolling through a picker or toggling a switch.
Guidelines for using haptics

• Build a clear, causal relationship between each haptic and its trigger.
• Use haptics in ways that complement other feedback in your app.
• Use haptics judiciously.
• Use haptics consistently.
• Avoid overusing a haptic.
• Be sure to test the haptics in your app.
• Make haptics optional.
• In a game, consider ways to use custom haptics to enhance the player’s experience.
• Be conscious that playing haptics might impact other user experiences.
Adding Haptics to an app

The Taptic Engine is Apple's implementation of haptic UI feedback. Using a linear actuator, iOS devices can generate a number of stored or user-defined tactile experiences.

There are three main ways to add haptics to an app:

• Use standard UI elements, such as switches, sliders, and pickers, that automatically play Apple-designed system haptics by default.

• Use a feedback generator to play one of several predefined haptic patterns.

• Compose and play your own haptic patterns.
Feedback Generators

A *feedback generator* is an interface to the Taptic Engine. Feedback generators belong to the class `UIFeedbackGenerator`, but you never create instances of this class yourself. Instead, you instantiate one of the three provided subclasses:

- `UIImpactFeedbackGenerator`: used to indicate that an impact has occurred. For example, you might trigger impact feedback when a user interface object collides with something or snaps into place.

There are five standard impact patterns ("styles"): light, medium, heavy, rigid, and soft.
Feedback Generators (cont.)

- **UISelectionFeedbackGenerator**: used to indicate a change in selection.

  There is one standard selection pattern: *selection*.

- **UINotificationFeedbackGenerator**: used to indicate successes, failures, and warnings.

  There are three standard notification patterns: *success*, *warning*, and *failure*.
Using Feedback Generators

To use a feedback generator:

1. Instantiate one of the three subclasses.

   // Instantiate a new selection feedback generator
   feedbackGenerator = UISelectionFeedbackGenerator()

2. OPTIONAL: prepare the generator. Calling the generator's prepare() method puts the Taptic Engine in a "prepared" state to reduce latency for a short while. Don't do this too early or too late.

   // Prepare the generator in anticipation of being triggered
   feedbackGenerator?.prepare()
3. Trigger the feedback using `impactOccurred()`, `selectionChanged()`, or `notificationOccurred()`. 

```swift
// Trigger selection feedback.
feedbackGenerator?.selectionChanged()
```

```swift
// OPTIONAL: stay in a prepared state for the next change
feedbackGenerator?.prepare()
```

4. Release the generator to let the Taptic Engine return to an idle state.

```swift
// Release the current generator.
feedbackGenerator = nil
```
Core Haptics lets you add customized haptic and audio feedback to your app. You create custom patterns out of basic building blocks called *haptic events*.

- *Transient* events are short and sweet, like the feedback you get from toggling a switch.
- *Continuous* events are longer, like the vibration or sounds from a ringtone.

Core Haptics is only supported on iPhone 8 and newer. It is not currently supported on any type of iPad or Apple Watch.
Using Core Haptics

**Import CoreHaptics, then create a Haptic Engine:**

```swift
var engine: CHHapticEngine? // Optional since it
                          // might not be supported

Use a guard statement to halt progress if the device is not supported on your device:

```swift
  guard CHHapticEngine.capabilitiesForHardware()
   .supportsHaptics else { return } 
```

```swift
do {
    engine = try CHHapticEngine()
    try engine?.start()
} catch {
    <handle error>
}
```
Create a series of **CHHapticEvent** objects:

```swift
let e1 = CHHapticEvent(eventType: .hapticTransient,
                        parameters: [], relativeTime: 0)
let e2 = CHHapticEvent(eventType: .hapticContinuous,
                        parameters: [], relativeTime: 0.2)
```

...

Create a pattern with **CHHapticPattern**:

```swift
let pattern = try CHHapticPattern(events: [ev1, ev2...],
                                   parameters: [])
```

Create a player and play the pattern:

```swift
let player = try engine?.makePlayer(with: pattern)
try player?.start(atTime: 0)
```