Core Motion
Core Motion

*Core Motion* is a framework that allows your application to receive motion data from device hardware.

For an iOS developer, this means you can create applications that can observe and respond to the motion and orientation of an iOS device.

**Important note:**

You can only test or use the functionality of Core Motion on an actual device. The simulator does not have any facilities for reproducing physical motion for your app.
Hardware Elements of Core Motion

Accelerometer
• Measures acceleration in all three dimensions

Gyroscope
• Calculates orientation and rotation in all three dimensions

Magnetometer
• Measures magnetic forces
Coordinate System

+x is towards the right of the screen

+y is towards the top of the screen

+z is towards the user when the screen is faceup
Rotations Within the Coordinate System

*Pitch*: rotation around the x axis

*Roll*: rotation around the y axis

*Yaw*: rotation around the z axis
A `CMDeviceMotion` object contains the following objects as properties:

- **attitude**: `CMAttitude`  
  - Returns the orientation of the device  
  - Can access the data in any of 3 representations

- **rotationRate**: `CMRotationRate`  
  - Returns the rotation rate of the device for devices with a gyro  
  - x, y, z values in radians per second

- **gravity**: `CMAcceleration`  
  - Returns the gravity vector expressed in the device's reference frame  
  - x, y, z values in g’s (gravitational force)

- **userAcceleration**: `CMAcceleration`  
  - Returns the acceleration that the user is giving to the device  
  - x, y, z values in g’s (gravitational force)

- **magneticField**: `CMCalibratedMagneticField`  
  - Returns the magnetic field vector with respect to the device for devices with a magnetometer
CMAAttitude contains three different representations of the device’s orientation:

- Euler angles (pitch, roll, yaw)
- Rotation matrices
- Quaternions

Each of these is in relation to a given reference frame.
Euler Angles are the most readily understood of the 3 representations, as they simply describe rotation around each of the axes.

- **Pitch** - is rotation around the x-axis, increasing as the device tilts toward you, decreasing as it tilts away

- **Roll** - is rotation around the y-axis, decreasing as the device rotates to the left, increasing to the right

- **Yaw** - is rotation around the z-axis, decreasing clockwise, increasing counter-clockwise
CMMotionManager provides a consistent interface for each of the four motion data types:

- Attitude (rotation)
- Acceleration
- Gravity
- Magnetic Field

Although you can access data for each of these motion types individually, it’s simplest to create a CMMotionManager instance to access all of the above.
If `deviceMotion` is a `CMDeviceMotion` object:

```javascript
deviceMotion.gravity.x
deviceMotion.gravity.y
deviceMotion.gravity.z

deviceMotion.userAcceleration.x
deviceMotion.userAcceleration.y
deviceMotion.userAcceleration.z

deviceMotion.attitude.pitch
deviceMotion.attitude.roll
deviceMotion.attitude.yaw

deviceMotion.magneticField.field.x
deviceMotion.magneticField.field.y
deviceMotion.magneticField.field.z
```
Using Core Motion in Your App

1. Create a motion manager:

   In your `ViewController` class:

   ```swift
   let motionManager = CMMotionManager()
   ```
Using Core Motion in Your App

2. Start receiving updates at the desired frequency:

```swift
override func viewDidLoad() {
    super.viewDidLoad()

    motionManager.deviceMotionUpdateInterval = 0.1

    motionManager.startDeviceMotionUpdates(to: OperationQueue.current!) {
        (deviceMotion, error) -> Void in

        if(error == nil) {
            // you write these methods
            self.handleUpdate(deviceMotion: deviceMotion!)
        } else {
            self.handleError()
        }
    }
}
```
Using Core Motion in Your App

3. Write code specifying what you want to happen at each update

```swift
func handleUpdate(deviceMotion: CMDeviceMotion) {
    let acceleration = deviceMotion.userAcceleration
    let xAcc = acceleration.x
    let yAcc = acceleration.y
    let zAcc = acceleration.z

    print("Acceleration in the x direction: \(xAcc)")
    print("Acceleration in the y direction: \(yAcc)")
    print("Acceleration in the z direction: \(zAcc)")
}

func handleError() {
    print("An error occurred")
```
Camera
Camera

Starting with the iOS 8 SDK, you can get access to the camera device, camera roll and photo library through the `UIImagePickerController` class.

This allows photos and videos to be taken from within an application and for existing photos and videos to be presented to the user for selection.

The `UIImagePickerController` is a view controller *that gets presented modally* (meaning as a popover). When we select or cancel the picker, it runs the delegate, where we handle the case and dismiss the modal.
The ultimate purpose of the `UIImagePickerController` class is to provide applications with either a photo or video. It achieves this by providing the user with access to the camera, camera roll or photo library on the device.

In the case of the camera, the user is able to either take a photo or record a video depending on the capabilities of the device and the application’s configuration of the `UIImagePickerController` object.
Attributes of an `UIImagePickerController`

- `sourceType` : `UIImagePickerControllerSourceType`
  One of
  - `.camera`
  - `.photoLibrary`
  - `.savedPhotosAlbum`

- `mediaTypes` : array of strings
  - `kUTTypeImage` (image)
  - `kUTTypeMovie` (video)

- `allowsEditing` : Boolean
  allow changes before the image is passed back to the application
Creating and configuring a `UIImagePickerController`

- Optionally, check to make sure you have access to the camera / camera roll / photo library using the `isSourceTypeAvailable(_:)` class method

- Optionally, check to make sure the media type you want to use is available by using the `availableMediaTypes(for:)` class method

- Create an instance of `UIImagePickerController` and set up its parameters.

- Identify a `UIImagePickerControllerDelegate`

- Present the image picker using `present()`.
Example code for UIImagePickerController

// create instance
let imagePicker = UIImagePickerController()

// identify delegate
imagePicker.delegate = self

// set up properties
imagePicker.sourceType = UIImagePickerControllerSourceType.photoLibrary
imagePicker.allowsEditing = false

// present the instance
present(imagePicker, animated:true, completion: nil)
As part of the `UIImagePickerController` delegate, you need to implement these protocol methods:

```swift
// Indicate that the user selected a photo/video
func imagePickerController(
    (UIImagePickerController, 
    didFinishPickingMediaWithInfo: 
    [String: Any] )

// Indicate that the user cancelled the pick
func imagePickerControllerDidCancel(
    (UIImagePickerController)
)```
Gesture Recognizers
There are 4 general types of UI events in iOS:

- **Touch** events: the most common
- **Motion** events
- **Remote-control** events: allow a responder object to receive commands from an external accessory or headset (usually to manage audio and video)
- **Press** events: represent interactions with a game controller, AppleTV remote, or other device that has physical buttons
Gestures refer to touches and touch events.

- Central to the modern smart phone experience
- A core built-in capability in iOS

A *touch* is an instance of the user putting a finger on the screen.

The OS and the hardware work together to know when a finger touches the screen, where it is, when it moves, and when it is no longer touching the screen.

Its location at any point in time is reduced to a single appropriate point.
Why are they important?

• They allow us to interact more naturally and intuitively with the application

• It is a significant paradigm shift to how humans interact with computers: analogous to what happened when people were first provided GUIs to interact with computers
Gesture Recognizers

*Gesture recognizers* are high-level mechanisms provided by iOS that takes care of the nitty-gritty of touch events, and makes it very easy to respond to a set of common touch events/sequences.

- They handle touches and movements of one or more fingers that happen on a specific area of the screen
- They are objects derived from the abstract `UIGestureRecognizer` class that are related to a view, and monitor for a predefined gesture made on that view
- There are some predefined subclasses which deal with specific (common) kinds of gestures
- They all perform an action once a valid gesture is detected.

*Without* gesture recognizers, you would be writing pages of code to handle what takes only a few lines of code *with* gesture recognizers.
You can set up gesture recognizers in IB or in code.

- A view can contain more than one gesture recognizer
- They are contained in a `UIView` property (an array) named `gestureRecognizers`

However, just one gesture can occur at any given point in time.

There are two types of gesture recognizers:

- Discrete: manage a single event; for example, touch to select an object
- Continuous: manage a series of events; for example, dragging an object on the screen
Gesture Recognizers

Predefined gesture recognizer classes:

- `UITapGestureRecognizer` *(discrete)*
- `UISwipeGestureRecognizer` *(discrete)*
- `UIPanGestureRecognizer` *(continuous)*
- `UIPinchGestureRecognizer` *(continuous)*
- `UIRotationGestureRecognizer` *(continuous)*
- `UILongPressGestureRecognizer` *(continuous)*
- `UIScreenEdgePanGestureRecognizer` *(continuous)*
Setting Up a Gesture Recognizer Using IB

- In IB, identify the object that you want to manipulate on the storyboard. Drag a Gesture Recognizer object on top of the target object.
- In the Swift file, write a function to handle the gesture.
- In IB, ctrl-drag the Gesture Recognizer object to the View Controller. Choose the name of the function you wrote.
- Click on the target object and go to the Attribute Inspector. Make sure "User Interface Enabled" is clicked on.
Setting Up a Gesture Recognizer Programmatically

- Create a Gesture Recognizer using one of the functions listed on the previous chart.
  
  ```swift
  let tapRecognizer = 
  UITapGestureRecognizer(target: self, action: 
  #selector(handleTap(recognizer:)))
  ```

- Set up any properties for the Gesture Recognizer that you may want.

- Associate the Gesture Recognizer with the target object.
  
  ```swift
  targetObject.addTapRecognizer(tapRecognizer)
  ```

- In the Swift file, write a function to handle the gesture.
  
  ```swift
  @IBAction func handleTap(recognizer: 
  UITapGestureRecognizer) {
  <code>
  }
  ```