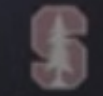
# Persistence

- Storing Data Permanently

  There are numerous ways to make data "persist" in iOS ...



# Persistence

- Storing Data Permanently

  There are numerous ways to make data "persist" in iOS ...

  In the filesystem (`FileManager`).



# Persistence

- Storing Data Permanently

  There are numerous ways to make data "persist" in iOS ...

  In the filesystem (`FileManager`).

  In a SQL database (`CoreData` for OOP access or even direct SQL calls).



# Persistence

- Storing Data Permanently

  There are numerous ways to make data "persist" in iOS ...

  In the filesystem (`FileManager`).

  In a SQL database (`CoreData` for OOP access or even direct SQL calls).

  iCloud (interoperates with both of the above).

  `CloudKit` (a database in the cloud).



# Persistence

- Storing Data Permanently

  There are numerous ways to make data "persist" in iOS ...

  In the filesystem (`FileManager`).

  In a SQL database (`CoreData` for OOP access or even direct SQL calls).

  iCloud (interoperates with both of the above).

  `CloudKit` (a database in the cloud).

  Many third-party options as well.



# Persistence

- Storing Data Permanently
  - There are numerous ways to make data "persist" in iOS ...
  - In the filesystem (`FileManager`).
  - In a SQL database (`CoreData` for OOP access or even direct SQL calls).
  - iCloud (interoperates with both of the above).
  - `CloudKit` (a database in the cloud).
  - Many third-party options as well.
  - `UserDefaults` (only for lightweight data like user preferences)



# File System

- Your application sees iOS file system like a normal Unix filesystem
  - It starts at /.
  - There are file protections, of course, like normal Unix, so you can't see everything.
  - In fact, you can only read and write in your application's "sandbox".



# File System

- Your application sees iOS file system like a normal Unix filesystem
  - It starts at /.
  - There are file protections, of course, like normal Unix, so you can't see everything.
  - In fact, you can only read and write in your application's "sandbox".
- Why sandbox?
  - Security (so no one else can damage your application)
  - Privacy (so no other applications can view your application's data)
  - Cleanup (when you delete an application, everything it has ever written goes with it)
  - Backup (certain parts of your sandbox are backed up when the device is backed up)



# File System

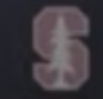
- Your application sees iOS file system like a normal Unix filesystem
  - It starts at /.
  - There are file protections, of course, like normal Unix, so you can't see everything.
  - In fact, you can only read and write in your application's "sandbox".
- Why sandbox?
  - Security (so no one else can damage your application)
  - Privacy (so no other applications can view your application's data)
  - Cleanup (when you delete an application, everything it has ever written goes with it)
  - Backup (certain parts of your sandbox are backed up when the device is backed up)
- So what's in this "sandbox"?
  - Application directory — Your executable, .jpgs, etc.; not writeable.
  - Documents directory — Permanent storage created by and always visible to the user.
  - Application Support directory — Permanent storage not seen directly by the user.
  - Caches directory — Store temporary files here (this is not backed up).
  - Other directories (see documentation) ...



# File System

- Getting a path to these special sandbox directories

  `FileManager` (along with `URL`) is what you use to find out about what's in the file system.



# File System

- Getting a path to these special sandbox directories

  `FileManager` (along with `URL`) is what you use to find out about what's in the file system.

  Usually we use the "default", shared `FileManager` via the `FileManager.default` static.



# File System

- Getting a path to these special sandbox directories

  `FileManager` (along with `URL`) is what you use to find out about what's in the file system.

  Usually we use the "default", shared `FileManager` via the `FileManager.default` static.

  You could use this to get a URL to one of the special directories mentioned above like this ...

```
let manager = FileManager.default
let url = manager.urls(for: .documentDirectory, in: .userDomainMask).first
```

  Similarly, you could get `.applicationSupportDirectory` or `.cachesDirectory`, etc.



# File System

- Getting a path to these special sandbox directories

  `FileManager` (along with `URL`) is what you use to find out about what's in the file system.

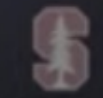
  Usually we use the "default", shared `FileManager` via the `FileManager.default` static.

  You could use this to get a URL to one of the special directories mentioned above like this ...

```
let manager = FileManager.default
let url = manager.urls(for: .documentDirectory, in: .userDomainMask).first
```

  Similarly, you could get `.applicationSupportDirectory` or `.cachesDirectory`, etc.

  When we are accessing the file system, we always start with a URL to a special directory.



# URL

- Building on top of these system paths

  URL methods:

```
func appendingPathComponent(String) -> URL
func appendingPathExtension(String) -> URL  // e.g. "jpg"
```



# URL

- Building on top of these system paths

  URL methods:

  ```
  func appendingPathComponent(String) -> URL
  func appendingPathExtension(String) -> URL  // e.g. "jpg"
  ```

- Finding out about what's at the other end of a URL

  ```
  var isFileURL: Bool  // is this a file URL (whether file exists or not) or something else?
  func resourceValues(for keys: [URLResourceKey]) throws -> [URLResourceKey:Any]?
  ```

  Example keys: `.creationDateKey`, `.isDirectoryKey`, `.fileSizeKey`



# File System

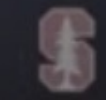
- Data

  Reading binary data from a URL …

  ```
  init(contentsOf: URL, options: Data.ReadingOptions) throws
  ```

  The options are almost always [].

  Notice that this function throws.



# File System

- Data

  Reading binary data from a URL …

  ```
  init(contentsOf: URL, options: Data.ReadingOptions) throws
  ```

  The options are almost always [].

  Notice that this function throws.

  Writing binary data to a URL …

  ```
  func write(to url: URL, options: Data.WritingOptions) throws -> Bool
  ```

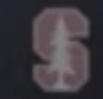
  The options can be things like .atomic (write to tmp file, then swap) or .withoutOverwriting.

  Notice that this function throws.



# File System

- `FileManager`
  - Provides utility operations.
  - e.g., `fileExists(atPath: String) -> Bool`
  - Can also create and enumerate directories; move, copy, delete files; etc.
  - Thread safe (as long as a given instance is only ever used in one thread).
  - Also has a `delegate` you can set which will have functions called on it when things happen.
  - And plenty more. Check out the documentation. And check out the demo ...



# Archiving

- `Codable` Mechanism

  Essentially a way to store all the vars of an object into a persistable blob.

  It's a great way to make an arbitrary `struct` be persistable (into the file system or wherever).



# Archiving

- `Codable` Mechanism

  Essentially a way to store all the vars of an object into a persistable blob.
  It's a great way to make an arbitrary `struct` be persistable (into the file system or wherever).

  For it to work, the `struct` you want to persist must implement the `Codable` protocol.
  For `structs` which contain only other `Codables`, Swift will implement the `Codable` for you.
  For enums, Swift will implement the `Codable` for you, unless you have associated data.



# Archiving

- Codable Mechanism

  Essentially a way to store all the vars of an object into a persistable blob.
  It's a great way to make an arbitrary struct be persistable (into the file system or wherever).

  For it to work, the struct you want to persist must implement the Codable protocol.
  For structs which contain only other Codables, Swift will implement the Codable for you.
  For enums, Swift will implement the Codable for you, unless you have associated data.

  Most standard types (that you'd recognize) already implement Codable ...
  String, Bool, Int, Double, Float
  Optional
  Array, Dictionary, Set, Data
  URL
  Date, DateComponents, DateInterval, Calendar
  CGFloat, AffineTransform, CGPoint, CGSize, CGRect, CGVector
  IndexPath, IndexSet



# Archiving

- Codable Mechanism

  Once your object graph is all Codable, you can convert it to JSON (a standard format).

```
let object: MyType = … // MyType must conform to Codable
let jsonData: Data? = try? JSONEncoder().encode(object)
```

  Note that this encode throws. You can catch and find errors easily (next slide).



# Archiving

- Codable Mechanism

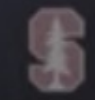
  Once your object graph is all Codable, you can convert it to JSON (a standard format).

```
let object: MyType = … // MyType must conform to Codable
let jsonData: Data? = try? JSONEncoder().encode(object)
```

  Note that this encode throws. You can catch and find errors easily (next slide).

  JSON is a text format and you can make a JSON String out of your jsonData like this …

```
let jsonString = String(data: jsonData!, encoding: .utf8) // JSON is always utf8
```



# Archiving

- Codable Mechanism

  Once your object graph is all Codable, you can convert it to JSON (a standard format).

```
let object: MyType = … // MyType must conform to Codable
let jsonData: Data? = try? JSONEncoder().encode(object)
```

  Note that this encode throws. You can catch and find errors easily (next slide).

  JSON is a text format and you can make a JSON String out of your jsonData like this …

```
let jsonString = String(data: jsonData!, encoding: .utf8) // JSON is always utf8
```

  You could also just write your jsonData out to a file with a URL you got from FileManager …

```
try jsonData.write(to: url) // note that this throws
```



# Archiving

- Codable Mechanism

  Once your object graph is all Codable, you can convert it to JSON (a standard format).

```
let object: MyType = … // MyType must conform to Codable
let jsonData: Data? = try? JSONEncoder().encode(object)
```

  Note that this encode throws. You can catch and find errors easily (next slide).

  JSON is a text format and you can make a JSON String out of your jsonData like this …

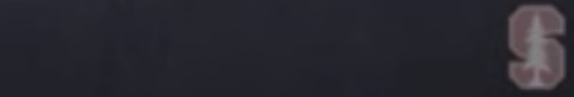
```
let jsonString = String(data: jsonData!, encoding: .utf8) // JSON is always utf8
```

  You could also just write your jsonData out to a file with a URL you got from FileManager …

```
try jsonData.write(to: url) // note that this throws
```

  To get your object graph back from the jsonData …

```
if let myObject: MyType = try? JSONDecoder().decode(MyType.self, from: jsonData!) {
    // do something with myObject, which we just decoded from jsonData
}
```



# Archiving

- Codable Mechanism

  Here's what it might look like to catch errors during a decoding.
  The thing decode throws is an enum of type DecodingError.
  Note how we can get the associated values of the enum similar to how we do with switch.

```
do {
    let object = try JSONDecoder().decode(MyType.self, from: jsonData!)
    // success, do something with object
} catch DecodingError.keyNotFound(let key, let context) {
    print("couldn't find key \(key) in JSON: \(context.debugDescription)")
} catch DecodingError.valueNotFound(let type, let context) {

} catch DecodingError.typeMismatch(let type, let context) {

} catch DecodingError.dataCorrupted(let context) {

}
```



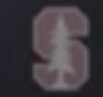
# Archiving

- `Codable` Example

  So how do you make your data types `Codable`? Usually you just say so …

```
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!
}
```

  If your vars are all also `Codable` (like the standard types all are), then you're done!



# Archiving

- Codable Example

  So how do you make your data types Codable? Usually you just say so …

```
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!
}
```

  If your vars are all also Codable (like the standard types all are), then you're done!

  The JSON for this might look something like …

```
{
    "someDate" : "2017-11-05T16:30:00Z",
    "someString" : "Hello",
    "other" : <whatever SomeOtherType looks like in JSON>
}
```



# Archiving

- Codable Example

  Sometimes JSON keys might have different names than your var names (or not be included).
  For example, someDate might be some_date.
  You can configure this by adding a private enum to your type called CodingKeys like this ...

```
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!

    private enum CodingKeys : String, CodingKey {
        case someDate = "some_date"
        // note that the someString var will now not be included in the JSON
        case other // this key is also called "other" in JSON
    }
}
```



# Archiving

- Codable Example

You can participate directly in the decoding by implementing the decoding initializer …

```
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        someDate = try container.decode(Date.self, forKey: .someDate)
        // process rest of vars, perhaps validating input, etc. …

    }
}
```

Note that this `init` throws, so we don't need `do { }` inside it (it will just rethrow).

Also note the "keys" are from the `CodingKeys` enum on the previous slide (e.g. `.someDate`).



# Archiving

- Codable Example

  You can participate directly in the encoding by implementing the encode(to:) function …

```
struct MyType : Codable {
    var someDate: Date
    var someString: String
    var other: SomeOtherType // SomeOtherType has to be Codable too!

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        // there are other containers too (e.g. an unkeyed (i.e. array) container) …
        try container.encode(someDate, forKey: .someDate)
        // encode the rest of vars, perhaps transforming them, etc. …
    }
}
```



# UserDefaults

- Data Types in UserDefaults
  - UserDefaults is an "ancient" API.
  - It far predates SwiftUI or even the Swift language.
  - As a result, its API is a little strange for those used to functional programming.
  - But if you "squint" at its API, you can integrate it into the Swift way of doing things.



# UserDefaults

- Data Types in UserDefaults

  UserDefaults is an "ancient" API.

  It far predates SwiftUI or even the Swift language.

  As a result, its API is a little strange for those used to functional programming.

  But if you "squint" at its API, you can integrate it into the Swift way of doing things.

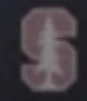
- Property Lists

  UserDefaults can only store what is called a Property List.



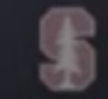
# UserDefaults

- Data Types in UserDefaults
  - UserDefaults is an "ancient" API.
  - It far predates SwiftUI or even the Swift language.
  - As a result, its API is a little strange for those used to functional programming.
  - But if you "squint" at its API, you can integrate it into the Swift way of doing things.
- Property Lists
  - UserDefaults can only store what is called a Property List.
  - This is not a protocol or a struct or anything tangible or Swift-like.
  - Property List simply a "concept".



# UserDefaults

- Data Types in UserDefaults
  - UserDefaults is an "ancient" API.
  - It far predates SwiftUI or even the Swift language.
  - As a result, its API is a little strange for those used to functional programming.
  - But if you "squint" at its API, you can integrate it into the Swift way of doing things.
- Property Lists
  - UserDefaults can only store what is called a Property List.
  - This is not a protocol or a struct or anything tangible or Swift-like.
  - Property List simply a "concept".
  - It is any combination of String, Int, Bool, floating point, Date, Data, Array or Dictionary.



# UserDefaults

- Data Types in UserDefaults

  UserDefaults is an "ancient" API.
  It far predates SwiftUI or even the Swift language.
  As a result, its API is a little strange for those used to functional programming.
  But if you "squint" at its API, you can integrate it into the Swift way of doing things.

- Property Lists

  UserDefaults can only store what is called a Property List.
  This is not a protocol or a struct or anything tangible or Swift-like.
  Property List simply a "concept".
  It is any combination of String, Int, Bool, floating point, Date, Data, Array or Dictionary.
  If you want to store anything else, you have to convert it into a Property List.

  A powerful way to do this for an arbitrary struct is using the Codable protocol in Swift.
  Codable converts structs into Data objects (and Data is a Property List).
  We'll let the demo cover how to do that.



# UserDefaults

- The Any type

  The API for UserDefaults is strange because it is pre-Swift.

  It has a lot of uses of the type Any (which basically means "untyped").



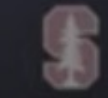
# UserDefaults

- The Any type

  The API for UserDefaults is strange because it is pre-Swift.
  It has a lot of uses of the type Any (which basically means "untyped").

  Swift is a strongly-typed language.
  It does not like Any but supports it for backwards compatibility.
  ... with things like UserDefaults.

  But we're going to try to ignore Any and still understand UserDefaults.



# UserDefaults

- Using UserDefaults

  First you need an instance of UserDefaults. Most often we do this ...

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



# UserDefaults

- Using UserDefaults

  First you need an instance of UserDefaults. Most often we do this …

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

- Storing Data

  To store something …

  ```
  defaults.set(object, forKey: "SomeKey") // object must be a Property List
  ```



# UserDefaults

- Using UserDefaults

  First you need an instance of UserDefaults. Most often we do this …

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

- Storing Data

  To store something …

  ```
  defaults.set(object, forKey: "SomeKey") // object must be a Property List
  ```

  For convenience, there are also a number of functions like this …

  ```
  defaults.setDouble(37.5, forKey: "MyDouble")
  ```

  See? Looks like a dictionary.



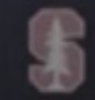
# UserDefaults

- Retrieving Data

  To retrieve something …

```
let i: Int = defaults.integer(forKey: “MyInteger”)
let b: Data = defaults.data(forKey: “MyData”)
let u: URL = defaults.url(forKey: “MyURL”)
let strings: [String] = defaults.stringArray(forKey: “MyStrings”)
```

  etc.



# UserDefaults

- Retrieving Data

  To retrieve something …

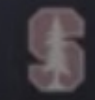
```
let i: Int = defaults.integer(forKey: “MyInteger”)
let b: Data = defaults.data(forKey: “MyData”)
let u: URL = defaults.url(forKey: “MyURL”)
let strings: [String] = defaults.stringArray(forKey: “MyStrings”)
```

  etc.

  Retrieving Arrays of anything but String is more complicated …

```
let a = array(forKey: “MyArray”)
```

  … will return an Array<Any>.



# UserDefaults

- Retrieving Data

  To retrieve something …

```
let i: Int = defaults.integer(forKey: "MyInteger")
let b: Data = defaults.data(forKey: "MyData")
let u: URL = defaults.url(forKey: "MyURL")
let strings: [String] = defaults.stringArray(forKey: "MyStrings")
```

  etc.

  Retrieving Arrays of anything but String is more complicated …

```
let a = array(forKey: "MyArray")
```

  … will return an Array<Any>.

  At this point you would have to use the as operator in Swift to "type cast" the Array elements.
  We'll stop there! But your reading assignment does describe how to use as.
  Hopefully you can use one of the above before needing Array<Any>.
  Codable might help you avoid this by using data(forKey:) instead.

