CS-444

Swift Programming

Functions and Closures

Use `func` to declare a function.

A `func` with no arguments and doesn't return anything:

```
func populateImages()
```

Functions and Closures

A `func` with 1 argument that doesn't return anything:

```
func add(caption: Caption)
```

Functions and Closures

A `func` with 1 argument that doesn't return anything:
By default, functions use their parameter names as labels for their arguments.

```
func add(caption: Caption)
```

Functions and Closures

A `func` with 1 argument and returns a string:
Use `->` to separate the parameter names and types from the function's return type

```
func author(caption: Caption) -> String
```

Functions and Closures

A `func` with 1 argument and returns a string:
Use `->` to separate the parameter names and types from the function's return type

```
func author(caption: Caption) -> String
```

Functions and Closures

```
func author(caption: Caption) -> String
```

Call a function by following its name with a list of arguments in parenthesis

Functions and Closures

```
func author(caption: Caption) -> String
```

Call a function by following its name with a list of arguments in parenthesis

```
let caption = Caption(from: data)
let author = author(caption: caption)
```

Functions and Closures

Write a custom argument label before the parameter name, or write _ to use no argument label

```
func author(_ caption: Caption) -> String

let author = author(caption)
```

Functions and Closures

Please don't do this

```
func author(_ caption: Caption) -> String
```

Optionals

Optionals

Handle the absence of a value

Optionals ensure that `nil` values are handled explicitly

Optionals

You use optionals in situations where a value may be absent

Optionals

An optional represents 2 possibilities:

1. **Either there is a value, and you can unwrap the optional to access that value**
2. **There IS not a value**

Optionals

In our CapThat app, we have an array named Pictures.
We use the array's built-in `randomElement()` function to return a random Picture

```
func randomImage() -> UIImage {
   let picture = images.randomElement()
```

Optionals

Array's randomElement() func:

```
///- Complexity: O(1) if the collection conforms to
///  `RandomAccessCollection`; otherwise, O(*n*), where *n* is the length
///  of the collection.
public func randomElement() -> Element?
```

Optionals

Array's randomElement() func:

```
///- Complexity: O(1) if the collection conforms to
///  `RandomAccessCollection`; otherwise, O(*n*), where *n* is the length
///  of the collection.
public func randomElement() -> Element?
```

The question mark in `Element?` indicates that the value it returns is optional, meaning that it might contain *some* Element, or it might contain *no value at all*

Optionals

```
var pictures: [Photo] = []
let picture = pictures.randomElement()
```

Array func `randomElement()` **can't guarantee that it will be able to return an element, so it returns an optional** `Element`

If Statements and Forced Unwrapping

You can use an `if` statement to find out whether an optional contains a value by comparing the optional value against `nil`

Perform this comparison with the `equal to` operator (==) or the "not equal to" operator (!=)

If Statements and Forced Unwrapping

If an optional has a value, it's considered to be "not equal to" nil:

```
let picture = pictures.randomElement()
if picture != nil { ... }
```

If Statements and Forced Unwrapping

Once you're sure that the optional does contain a value, you can access its underlying value by adding an exclamation mark (!) at the end of the optional's name.

If Statements and Forced Unwrapping

Once you're sure that the optional does contain a value, you can access its underlying value by adding an exclamation mark (!) at the end of the optional's name.

```
let picture = pictures.randomElement()
if picture != nil {
 print("random picture with name found: \(picture.name!))"
}
```

If Statements and Forced Unwrapping

```
let picture = pictures.randomElement()
if picture != nil {
 print("random picture with name found: \(picture.name!))"
}
```

`picture.name!` **effectively says, "I know this optional definitely has a value, please use it."**

If Statements and Forced Unwrapping

`picture.name!` **This is known as *forced unwrapping* of an optional's value**

Optional Binding

Optional Binding

You use optional binding to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable.

Optional binding can be used with `if` and `while` statements to check for a value inside an optional, and to extract that value into a constant or a variable, as part of a single action.

Optional Binding

If Statements

With *Forced Unwrapping* (previous):

```
let picture = pictures.randomElement()
if picture != nil {
 print("random picture with name found: \(picture.name!))"
}
```

Optional Binding

If Statements

With *Optional Binding*

```
let picture = pictures.randomElement()
if let p = picture {
 print("random picture with name found: \(p.name))"
}
```

Optional Binding

```
let picture = pictures.randomElement()
if let p = picture {
 print("random picture with name found: \(p.name))"
}
```

This can be read as: "If the optional Picture returned by `pictures.randomElement()` contains a value, set a new constant called `p` to the value contained in the optional."

Optional Binding

If the conversion is successful (if let p = picture), the `p` constant becomes available for use within the first branch of the `if statement`

```
let picture = pictures.randomElement()
if let p = picture {
 print("random picture with name found: \(p.name))"
}
```

Optional Binding

If Statement formula

```
if let [constantName] = [someOptional] { ... }
```

Optional Binding

You can use as many optional bindings and Boolean conditions in a single if statement as you need to, separated by commas. If any of the values in the optional bindings are `nil` or any Boolean conditions evaluates to `false`, the *entire* if statement's condition is considered to be false.

Optional Binding

The following statement would hit ELSE

```
var captions: [Caption] = []
let picture = pictures.randomElement()
let caption = caption.randomElement()

if let p = picture, c = caption {
 print("random picture with name found: \(p.name), and caption: \(c.text)"
} else {
 print("A caption and/or picture was not found"
}
```

Error Handling

Error Handling

You use error handling to error conditions your program may encounter during execution

Error Handling

In contrast to optionals, which can use the presence or absence of a value to communicate success or failure of a function...error handling allows you to:

1. determine the underlying causes of failure
2. Propagate the error to another part of your program

Error Handling

When a function encounters an error condition, it throws an error. That function caller can then catch the error and response appropriately.

```
static func readJSONFromFile(fileName: String) throws -> Any? { ... }
```

Error Handling

```
static func readJSON(from fileName: String) throws -> Any? { ... }
```

A function indicates that it can throw an error by including the `throws` keyboard in the declaration.

Error Handling

When you call a function that can throw an error, you prepend the `try` keyword to the expression

```
let fileName = "theoffice.json"
let jsonData = try readJSONFromFile(fileName: fileName)
```

Error Handling

Swift automatically propagates errors out of their current scope until they're handled by a `catch` clause

```
do {
  try can ThrowAnError()
  // no error was thrown
} catch {
  // a error was thrown
}
```

Error Handling

A `do` statement creates a new containing scope, which allows errors to be propagated to one of more `catch` clauses

```
do {
   try can ThrowAnError()
   // no error was thrown
} catch {
   // an error was thrown
}
```

Error Handling

```
    // Import quotes from local JSON file
    let fileName = "theoffice"
    do {
    let captions = try LocalJsonService.readJSON(from: fileName) as! NSDictionary
            print("captions count: \(captions.count)")
        } catch let error as NSError {
            print("Failed to \(error.localizedDescription)")
        }

static func readJSON(from fileName: String) throws -> Any? {
    // does some json stuff. If it can't find the file,
    // or json doesn't parse correctly, throw an error
}
```

Error Handling

```
// Import quotes from local JSON file
let fileName = "theoffice"
do {
let captions = try LocalJsonService.readJSON(from: fileName) as! NSDictionary
        print("captions count: \(captions.count)")
    } catch let error as NSError {
        print("Failed to \(error.localizedDescription)")
    }

static func readJSON(from fileName: String) throws -> Any? {
 // does some json stuff. If it can't find the file,
 // or json doesn't parse correctly, throw an error
}
```

If no error handling is thrown, the `print(captions count:` log is called. If an error is thrown and it matches the generic `NSError`, then the `print("Failed to \...)` is called.

Assertions and Preconditions

Assertions and preconditions are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code.

Assertions and Preconditions

If the boolean condition in the assertion or precondition evaluates to be `true`, code execution continues as usual.

Assertions and Preconditions

If the condition evaluates to `false`, the current state of the program is invalid; code execution ends, and your app is terminated

Assertions and Preconditions

Assertions and preconditions are great

You use assertions and preconditions to express the assumptions you make and the expectations you have while coding, as you can include them as part of your code.

Assertions and Preconditions

Unlike Error Handling (try/do/catch) assertions and preconditions aren't used for recoverable or expected errors. Because a failed assertion or precondition indicates an invalid program state, there's no way to catch a failed assertion..

Debugging with Assertions

You can write an assertion calling the `assert(_:_:file:line:)`

TO BE CONTINUED

Add what we learned to CapThat