Intermediate Swift

Instructors: John Clem, Brad Johnson
TA's: Cole Bratcher, Dan Fairbanks, Reed Sweeney

closures{}

before we can talk about closures, we need to talk about functions

functions are first class citizens in Swift (you can have functions be parameters to another function, and you can return a function from another function)

functions have a type, and the type is defined by the parameters + return type.

String has a method toInt() that takes a String and returns an int. So that function's type is `(String) -> Int`.

closures{}

functions are simply closures that don't do any value-capturing.

nested functions capture values from their enclosing function

closure expressions:

- like inline blocks are to Objective-C
- can capture values from their surrounding context

closure syntax

flexible, but challenging (illegible) syntax

closure expression syntax:

```
{ (parameters) -> return type in statements
```

the `in` keyword is the start of the closure body

please refer to fuckingclosuresyntax.com

refining the syntax

Implicit Returns from Single-Expression Closures

if the closure only has one expression, you don't need the `return`

shorthand argument names.

$0, $1, $2, refer to the parameters in the closures body, and drop the argument list entirely. Thanks Dre.

trailing closures{}

you can write a trailing closure if

- the closure is the final argument of a function

use trailing closures when you can't fit the closure on a single line

outside & after the the function parameters

```
networkController.fetchDataWithCompletion() {
  // completion stuff goes here
}
```

capturing values

closures capture variables from the context they are defined in

variables that are not modified are copied in (enclosed)

variables that are modified are referenced and kept alive.

extensions

add new functionality to an existing class, structure, or enum.

unlike categories in Objective-C, extensions do not have names

useful for computed properties, methods, initializers, subscripts, new nested types

use to make an existing type conform to a protocol

extensions

Computed properties don't actually store a value, just compute and return a value from existing data.

Initializers: adds new convenience initializers, but not designated initializers.

Methods: regular instance and type methods, or a mutating method that modifies self on structs or enums. Must be marked with mutating keyword.

extensions (cont)

Subscripts: add new subscripts to an existing class.

Nested Types. types, structs, and enums nested inside an extended type.

optionals?

forced unwrapping (!) accesses the value inside an optional

optional binding checks if it has a value before proceeding

optional chaining validates the optional nested properties and aborts after first nil?

Implicitly Unwrapped Optionals are used when you can be sure an optional! will have a value after initial setup

downcasting with ‘as?’