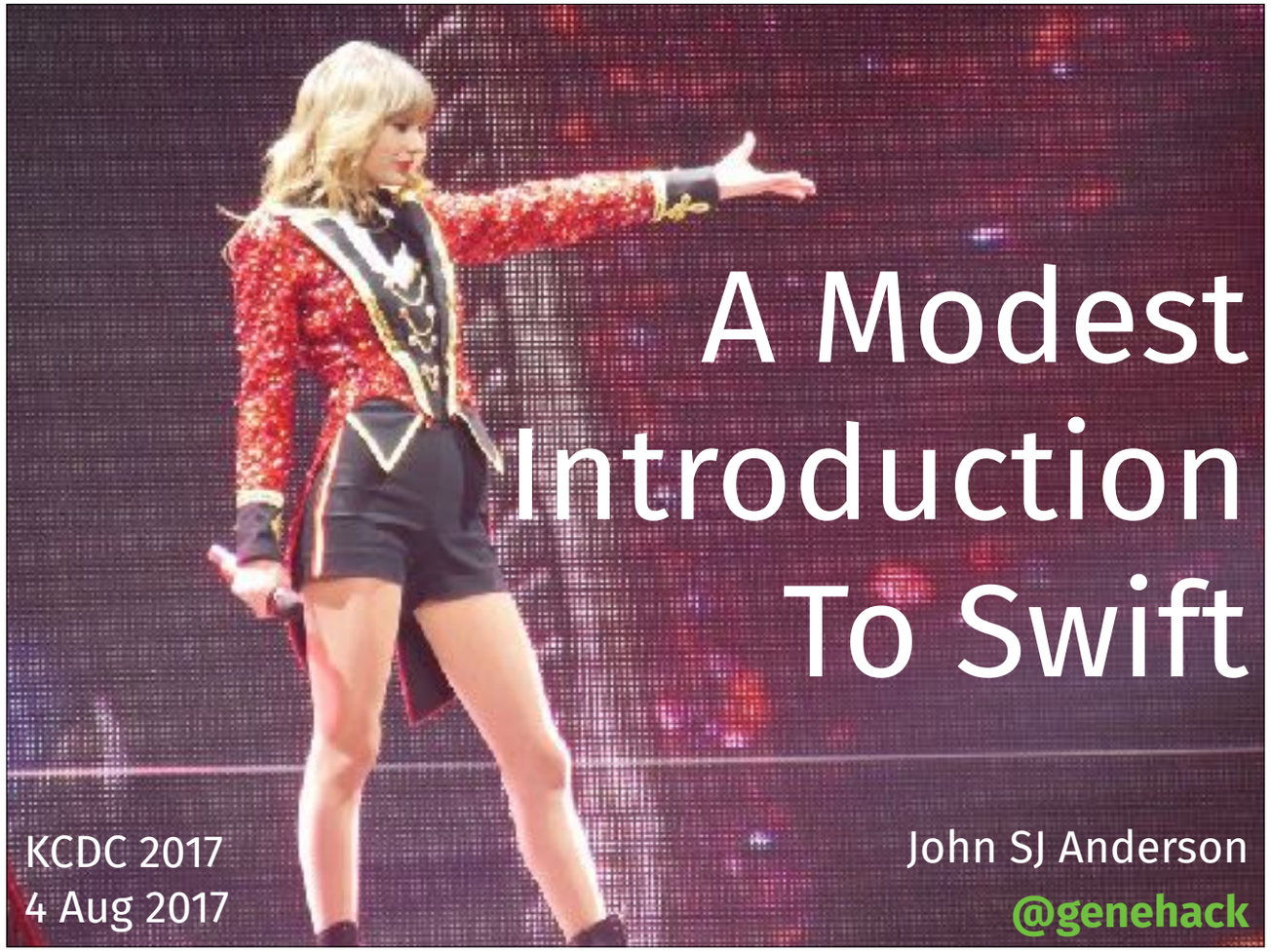


A Modest Introduction To Swift

KCDC 2017
4 Aug 2017

John SJ Anderson
@genehack

Sorry!

These are my obligatory two "Swift" jokes, wanted to get those out of the way...

TITANIUM SPONSORS

epiq
PAIGE TECHNOLOGIES
INTELLIGENT PAIRING. PERPETUAL SUCCESS.
ADAPTIVE™ SOLUTIONS GROUP
okta

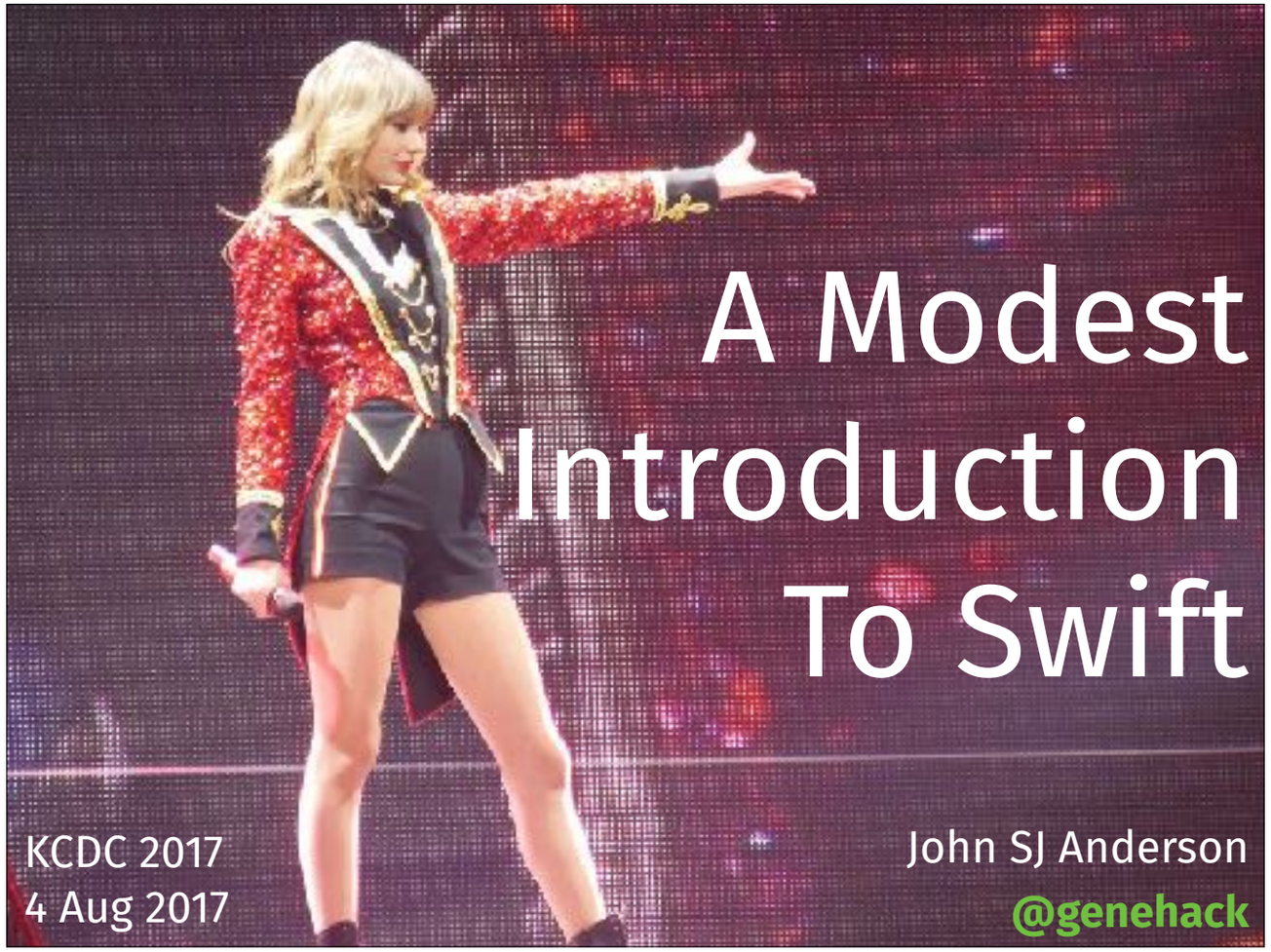
Platinum Sponsors

VinSolutions
Make every connection count.
jack henry & ASSOCIATES INC.
VALOREM
NAIC
National Association of Insurance Commissioners
NIPR
NATIONAL INSURANCE PRODUCER REGISTRY
TEKsystems
Our people make it possible.
KEYHOLE SOFTWARE
smartsheet
Work Better™
Couchbase
EquipmentShare
KOCH™
PLURALSIGHT
Google Cloud
VML
mylo™
POWERED BY LOCKTON

Gold Sponsors

Advantage Tech
PeopleAdmin
CENTRIQ TRAINING
Cerner
KU EDWARDS CAMPUS
twilio
{Track:js}
OBJECT PARTNERS
smg service management group®
slackify
GARMIN.
Balance Innovations
Allied Global Services
dsi
LAS
Ayiture
TouchNet - Heartland
Medware
AUREUS GROUP
Children's Mercy KANSAS CITY
Commerce Bank
SECURITY PS
ProKarma


infinityinteractive
technical, not typical
partner, not vendor
We solve problems with technology
consulting
development
web
mobile
devops



A Modest Introduction To Swift

KCDC 2017
4 Aug 2017

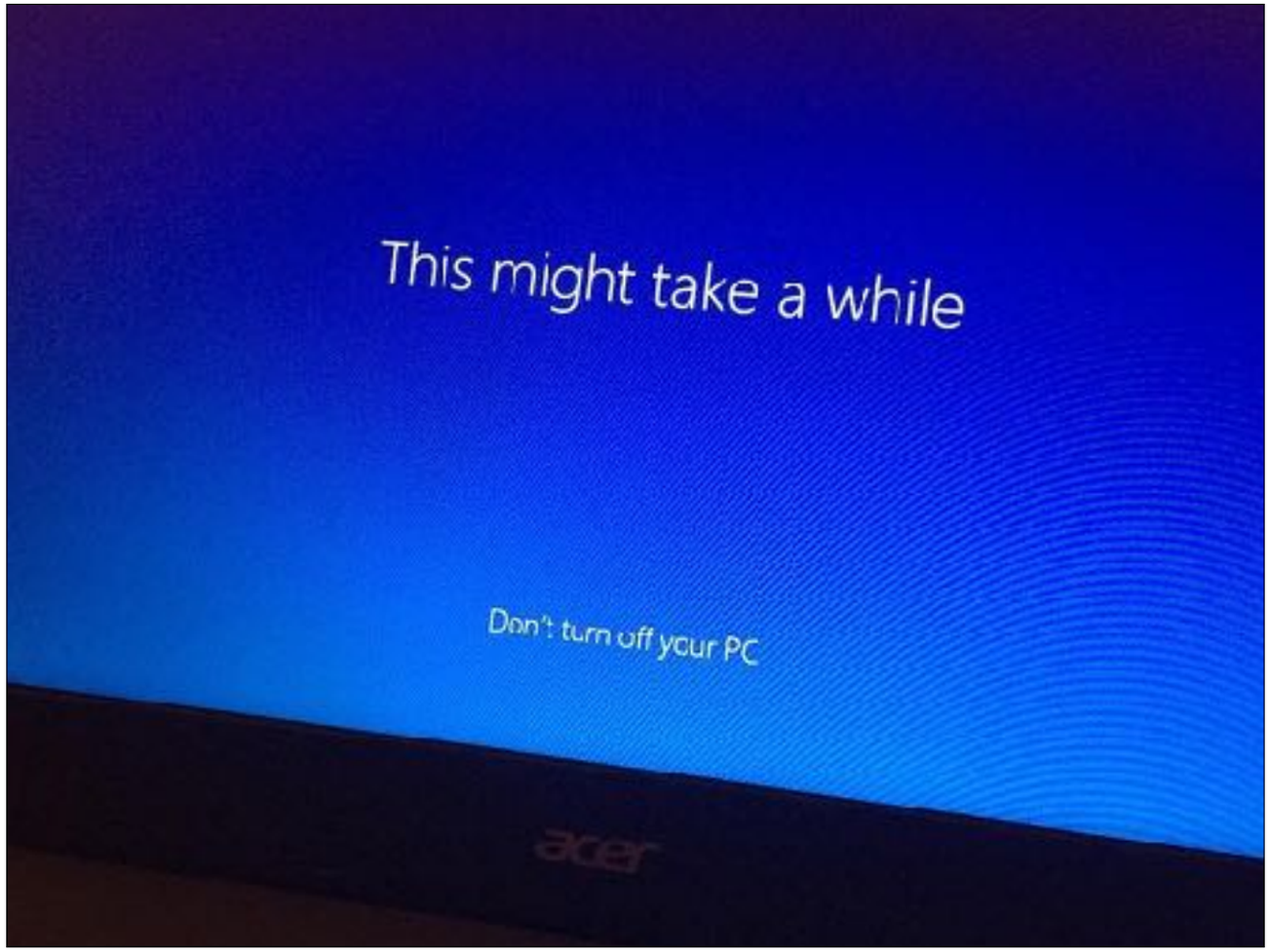
John SJ Anderson
@genehack

Hi I'm John

aka **@genehack**

- VP Tech, Infinity Interactive
- Perl tribe
- Polyglot coder

- Just this guy, you know?



Disclaimer

Swift?

So, what is Swift?

Introduced in 2014

Went Open Source at version 2.2

Version 3.1 just released (in March)

Version 4 coming
Real Soon Now

Originally macOS only

Now on Linux too.

Releases

Swift 3.1.1

Download	Date
Xcode 8.3.2[7] (Toolchain) (Debugging Symbols)	April 21, 2017
Ubuntu 16.10 (Signature)	April 21, 2017
Ubuntu 16.04 (Signature)	April 21, 2017
Ubuntu 14.04 (Signature)	April 21, 2017

Android in the works!

[RFC] Port to Android #1442

Open **modocache** wants to merge 1 commit into `apple:master` from `SwiftAndroid:master`

Conversation 54 | Commits 1 | Files changed 53

modocache commented 14 days ago

What's in this pull request?

This adds an Android target for the stdlib. It is also the first example of cross-compiling outside of Darwin: a Linux host machine builds for an Android target.

Port to Android #1442

Merged **swift-ci** merged 1 commit into `apple:master` from `SwiftAndroid:master` on Apr 12

Conversation 198 | Commits 1 | Files changed 34

modocache commented on Feb 25

What's in this pull request?

This adds an Android target for the stdlib. It is also the first example of cross-compiling outside of Darwin: a Linux host machine builds for an Android target.

Android ~~in newer~~ks!

DONE!!

Windows too?


https://swiftforwindows.github.io/
Swift for Windows
Write in Swift, Run on Windows

Originally targeted macOS, iOS, watchOS, & tvOS

With expanded platform support, offers potential “single-language stack” advantages a la Node


So what's it like, man?


First, a brief digression...

How many macOS / iOS users do we have here?

How many macOS / iOS ***developers*** do we have?

The dirty little secret of developing for Apple

From *In The Beginning Was The Command Line* by Neal Stephenson

During the late 1980's and early 1990's I spent a lot of time programming Macintoshes, and eventually decided for fork over several hundred dollars for an Apple product called the Macintosh Programmer's Workshop, or MPW. MPW had competitors, but it was unquestionably the premier software development system for the Mac. It was what Apple's own engineers used to write Macintosh code. Given that MacOS was far more technologically advanced, at the time, than its competition, and that Linux did not even exist yet, and given that this was the actual program used by Apple's world-class team of creative engineers, I had high expectations. It arrived on a stack of floppy disks about a foot high, and so there was plenty of time for my excitement to build during the endless installation process. The first time I launched MPW, I was probably expecting some kind of touch-feely multimedia showcase. Instead it was austere, almost to the point of being intimidating.

Weird Pascal-based naming & calling conventions

HANDLES?!?

ObjectiveC’s “syntax”


Swift is the Mac of Apple
programming languages

Swift Syntax

Comments

```
// this is a comment
```

Comments

```
/* this is a
multi-line
comment */
```

Comments

```
/* this is a
/* _nested_
multi-line comment, */
which is cool! */
```

Variables

```
var foo = 1
var bar: Int
var baz = "whee!"
```

Variables

```
var foo = 1
var bar: Int
var baz = "whee!"
```

Variables

```
var foo = 1
var bar: Int
var baz = "whee!"
```

Variables

```
var foo = 1
var bar: Int
var baz = "whee!"
```

Variables

```
let bar = 1
bar += 1
// ^^ compile time error!
```

Variables

```
let bar = 1
bar += 1
// ^^ compile time error!
```

Variables

```
let bar
```

Variables

```
let bar

// also a compile time error

/*
You canNOT have an uninitialized and
untyped variable. You also can't use
an uninitialized variable _at all_
*/
```


How friggin’ awesome is that?

Operators

Flow Control

```
let n = 1

if n > 1 {
    print("we got a big N here")
}
```

Flow Control

```
let n = 1

if n > 1 {
    print("we got a big N here")
}
```

Flow Control

```
let n = 1

if n > 1 {
    print("we got a big N here")
}
```

Flow Control

```
let arr = [ 1, 2, 3, 4]
var sum = 0

for elem in arr {
    sum += elem
}

// sum now is 10
```

Flow Control

```
let arr = [ 1, 2, 3, 4]
var sum = 0

for elem in arr {
    sum += elem
}

// sum now is 10
```

Flow Control

```
let arr = [ 1, 2, 3, 4]
var sum = 0

for elem in arr {
    sum += elem
}

// sum now is 10
```

Flow Control

```
let arr = [ 1, 2, 3, 4]
var sum = 0

for elem in arr {
    sum += elem
}

// sum now is 10
```

Flow Control

```
for index in 1 ... 10 {
    # do something 10 times
}
```

Flow Control

```
for index in 1 ... 10 {
    # do something 10 times
}
```

Flow Control

```
for index in 1 ..< 10 {
    # do something 9 times
}
```

Flow Control

```
for index in 1 ..< 10 {
    # do something 9 times
}
```

Flow Control

```
var countDown = 5
while countDown > 0 {
    countDown--
}
```

Flow Control

```
var countDown = 5
while countDown > 0 {
    countDown--
}
```

Flow Control

```
var countDown = 5
while countDown > 0 {
    countDown--
}
```

Flow Control

```
var countDown = 5
while countDown > 0 {
    countDown -= 1
}
```

Flow Control

```
var countUp = 0
repeat {
    countUp++
} while countUp < 5
```

Flow Control

```
var countUp = 0
repeat {
    countUp++
} while countUp < 5
```

Flow Control

```
var countUp = 0
repeat {
    countUp++
} while countUp < 5
```

Flow Control

```
var countUp = 0
repeat {
    countUp += 1
} while countUp < 5
```

Flow Control

```
let sample = 2

switch sample {
case 0:
    print("Is 0")

case 2:
    print("Is 2")

default: // mandatory when cases not exclusive
    print("Not 0 or 2, is it.")
}
```

Flow Control

```
let sample = 2

switch sample {
case 0:
    print("Is 0")

case 2:
    print("Is 2")

default: // mandatory when cases not exclusive
    print("Not 0 or 2, is it.")
}
```

Flow Control

```
let sample = 2

switch sample {
case 0:
    print("Is 0")

case 2:
    print("Is 2")

default: // mandatory when cases not exclusive
    print("Not 0 or 2, is it.")
}
```

Flow Control

```
let sample = 2

switch sample {
case 0:
    print("Is 0")

case 2:
    print("Is 2")

default: // mandatory when cases not exclusive
    print("Not 0 or 2, is it.")
}
```

No fallthru by default!

Flow Control

```
let sample = 2

switch sample {
case 0:
    print("Is 0")

case 2:
    print("Is 2")

default: // mandatory when cases not exclusive
    print("Not 0 or 2, is it.")
}
```

Flow Control

```
let sample = "foo"

switch sample {
case "foo":
    print("Is foo")

case "bar":
    print("Is bar")

default: // mandatory when cases not exclusive
    print("Not foo or bar, is it.")
}
```

Flow Control

```
let sample = "foo"

switch sample {
case "foo":
    print("Is foo")

case "bar":
    print("Is bar")

default: // mandatory when cases not exclusive
    print("Not foo or bar, is it.")
}
```

Flow Control

```
let sample = ("foo", 2)

switch sample {
case ("foo", 2):
    print("Is foo, 2")

case ("bar", _):
    print("Is bar")

default: // mandatory when cases not exclusive
    print(" ¯\_(ツ)_/¯ ")
}
```

Flow Control

```
let sample = ("foo", 2)

switch sample {
case ("foo", 2):
    print("Is foo, 2")

case ("bar", _):
    print("Is bar")

default: // mandatory when cases not exclusive
    print(" ¯\_(ツ)_/¯ ")
}
```

Flow Control

```
let sample = ("foo", 2)

switch sample {
case ("foo", 2):
    print("Is foo, 2")

case ("bar", _):
    print("Is bar")

default: // mandatory when cases not exclusive
    print(" ¯\_(ツ)_/¯ ")
}
```

Flow Control

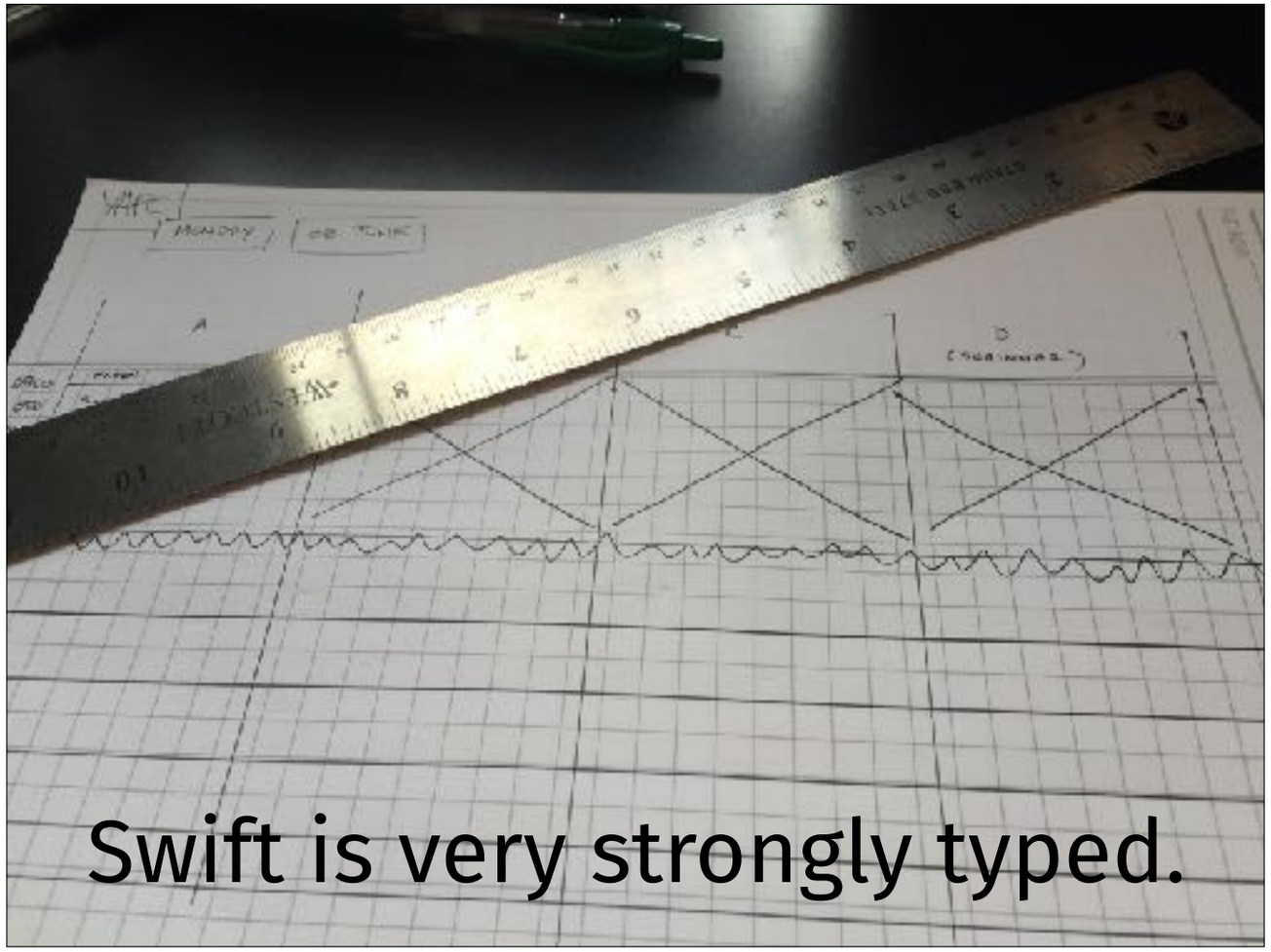
```
let sample = ("foo", 2)

switch sample {
case ("foo", 2):
    print("Is foo, 2")

case ("bar", _):
    print("Is bar")

default: // mandatory when cases not exclusive
    print(" ¯\_(ツ)_/¯ ")
}
```

Flow Control

```
let sample = ("foo", 2)

switch sample {
case ("foo", 3):
    print("Is foo, 3")

case (let one, let two):
    print("Is \(one) and \(two)")
}
```

Flow Control

```
let sample = ("foo", 2)

switch sample {
case ("foo", 3):
    print("Is foo, 3")

case (let one, let two):
    print("Is \(one) and \(two)")
}
```


Swift is very strongly typed.

Typing

```
var foo = 1  // foo is an Int

var bar: Int // bar is an uninit'd Int

var baz = Int()

if baz is Int {
    print("Nice Int you got there")
}
```

Typing

```
var foo = 1  // foo is an Int

var bar: Int // bar is an uninit'd Int

var baz = Int()

if baz is Int {
    print("Nice Int you got there")
}
```

Typing

```
var foo = 1  // foo is an Int

var bar: Int // bar is an uninit'd Int

var baz = Int()

if baz is Int {
    print("Nice Int you got there")
}
```

Typing

```
var foo = 1  // foo is an Int

var bar: Int // bar is an uninit'd Int

var baz = Int()

if baz is Int {
    print("Nice Int you got there")
}
```

Typing

```
var foo = 1  // foo is an Int

var bar: Int // bar is an uninit'd Int

var baz = Int()

if baz is Int {
    print("Nice Int you got there")
}
```

Casts

```
var foo = 1 // foo is an Int

var bar = String(foo) // "1"

var maybeBaz = stringishThing as? String
// maybeBaz is an optionally typed String

var forceBaz = stringishThing as! String
```

Casts

```
var foo = 1 // foo is an Int

var bar = String(foo) // "1"

var maybeBaz = stringishThing as? String
// maybeBaz is an optionally typed String

var forceBaz = stringishThing as! String
```

Casts

```
var foo = 1 // foo is an Int

var bar = String(foo) // "1"

var maybeBaz = stringishThing as? String
// maybeBaz is an optionally typed String

var forceBaz = stringishThing as! String
```

Casts

```
var foo = 1 // foo is an Int

var bar = String(foo) // "1"

var maybeBaz = stringishThing as? String
// maybeBaz is an optionally typed String

var forceBaz = stringishThing as! String
```

Optional Types

```
// When a variable may not have a value
var bar: Int?

// test
if bar != nil {
    // has a value
}
```

Optional Types

```
// When a variable may not have a value
var bar: Int?

// test
if bar != nil {
    // has a value
}
```

Optional Types

```
// When a variable may not have a value
var bar: Int?

// test
if bar != nil {
    // has a value
}
```

Optional Types

```
// force unwrap the value to use
if bar != nil {
  bar! += 2
}

// unwrapping nil --> runtime exception!
```

Optional Types

```
// force unwrap the value to use
if bar != nil {
  bar! += 2
}

// unwrapping nil --> runtime exception!
```

Optional Types

```
// force unwrap the value to use
if bar != nil {
  bar! += 2
}

// unwrapping nil --> runtime exception!
```

if-let

```
var bar: Int?

if let foo = bar {
    // bar had a value &
    // foo now has that unwrapped value
}
else {
    // bar was nil
}
```

if-let

```
var bar: Int?

if let foo = bar {
    // bar had a value &
    // foo now has that unwrapped value
}
else {
    // bar was nil
}
```

if-let

```
var bar: Int?

if let foo = bar {
    // bar had a value &
    // foo now has that unwrapped value
}
else {
    // bar was nil
}
```

if-var

```
var bar: Int?

if var foo = bar {
    // bar had a value &
    // foo now has that unwrapped value &
    // foo is mutable
    foo += 1
}
else {
    // bar was nil
}
```

if-var

```
var bar: Int?

if var foo = bar {
    // bar had a value &
    // foo now has that unwrapped value &
    // foo is mutable
    foo += 1
}
else {
    // bar was nil
}
```

if-var

```
var bar: Int?

if var foo = bar {
    // bar had a value &
    // foo now has that unwrapped value &
    // foo is mutable
    foo += 1
}
else {
    // bar was nil
}
```

Types of Variables

Tuples

```
let tuple = ("foo", 42)

let first = tuple.0 // "foo"

let labeledTuple = (one: "foo", two: 42)

let second = labeledTuple.two // 42
```

Tuples

```
let tuple = ("foo", 42)

let first = tuple.0 // "foo"

let labeledTuple = (one: "foo", two: 42)

let second = labeledTuple.two // 42
```

Tuples

```
let tuple = ("foo", 42)

let first = tuple.0 // "foo"

let labeledTuple = (one: "foo", two: 42)

let second = labeledTuple.two // 42
```

Tuples

```
let tuple = ("foo", 42)

let first = tuple.0 // "foo"

let labeledTuple = (one: "foo", two: 42)

let second = labeledTuple.two // 42
```

Tuples

```
let tuple = ("foo", 42)

let first = tuple.0 // "foo"

let labeledTuple = (one: "foo", two: 42)

let second = labeledTuple.two // 42
```

Arrays

```
let nums = [1, 2, 3]

var strs : [String]

// _can_ mix & match
let mixed = [1, "foo"]

// but you probably shouldn't
// in Swift 3+ this is a compile
// error unless specifically
// type-annotated
let mixed: [Any] = [1, "foo"]
```

Arrays

```
let nums = [1, 2, 3]

var strs : [String]

// _can_ mix & match
let mixed = [1, "foo"]

// but you probably shouldn't
// in Swift 3+ this is a compile
// error unless specifically
// type-annotated
let mixed: [Any] = [1, "foo"]
```

Arrays

```
let nums = [1, 2, 3]

var strs : [String]

// _can_ mix & match
let mixed = [1, "foo"]

// but you probably shouldn't
// in Swift 3+ this is a compile
// error unless specifically
// type-annotated
let mixed: [Any] = [1, "foo"]
```

Arrays

```
let nums = [1, 2, 3]

var strs : [String]

// _can_ mix & match
let mixed = [1, "foo"]

// but you probably shouldn't
// in Swift 3+ this is a compile
// error unless specifically
// type-annotated
let mixed: [Any] = [1, "foo"]
```

Arrays

```
let nums = [1, 2, 3]

var strs : [String]

// _can_ mix & match
let mixed = [1, "foo"]

// but you probably shouldn't
// in Swift 3+ this is a compile
// error unless specifically
// type-annotated
let mixed: [Any] = [1, "foo"]
```

Dictionary

```
let capitalCityStates = [
  "Salem": "Oregon",
  "Jeff City": "Missouri",
  "Topeka": "Kansas"
]

// capitalCityStates has type
// [String:String]
```

Dictionary

```
let capitalCityStates = [
  "Salem": "Oregon",
  "Jeff City": "Missouri",
  "Topeka": "Kansas"
]

// capitalCityStates has type
// [String:String]
```

Dictionary

```
let capitalCityStates = [
  "Salem": "Oregon",
  "Jeff City": "Missouri",
  "Topeka": "Kansas"
]

// capitalCityStates has type
// [String:String]
```

Dictionary

```
let capitalCityStates = [
  "Salem": "Oregon",
  "Jeff City": 2,
  "Topeka": "Kansas"
]

// capitalCityStates must be
// annotated with type
// [String:Any]
```

Dictionary

```
let capitalCityStates = [
  "Salem": "Oregon",
  "Jeff City": 2,
  "Topeka": "Kansas"
]

// capitalCityStates must be
// annotated with type
// [String:Any]
```

Dictionary

```
let capitalCityStates = [
  "Salem": "Oregon",
  "Jeff City": 2,
  "Topeka": "Kansas"
]

// capitalCityStates must be
// annotated with type
// [String:Any]
```

Dictionary

```
let capitalCityStates: [String:Any] = [
  "Salem": "Oregon",
  "Jeff City": 2,
  "Topeka": "Kansas"

]

// capitalCityStates must be
// annotated with type [String:Any]
```

Sets

```
var petSet :Set = [ "cat", "dog",
                    "fish", "dog"]

petSet.count // returns 3
```

Sets

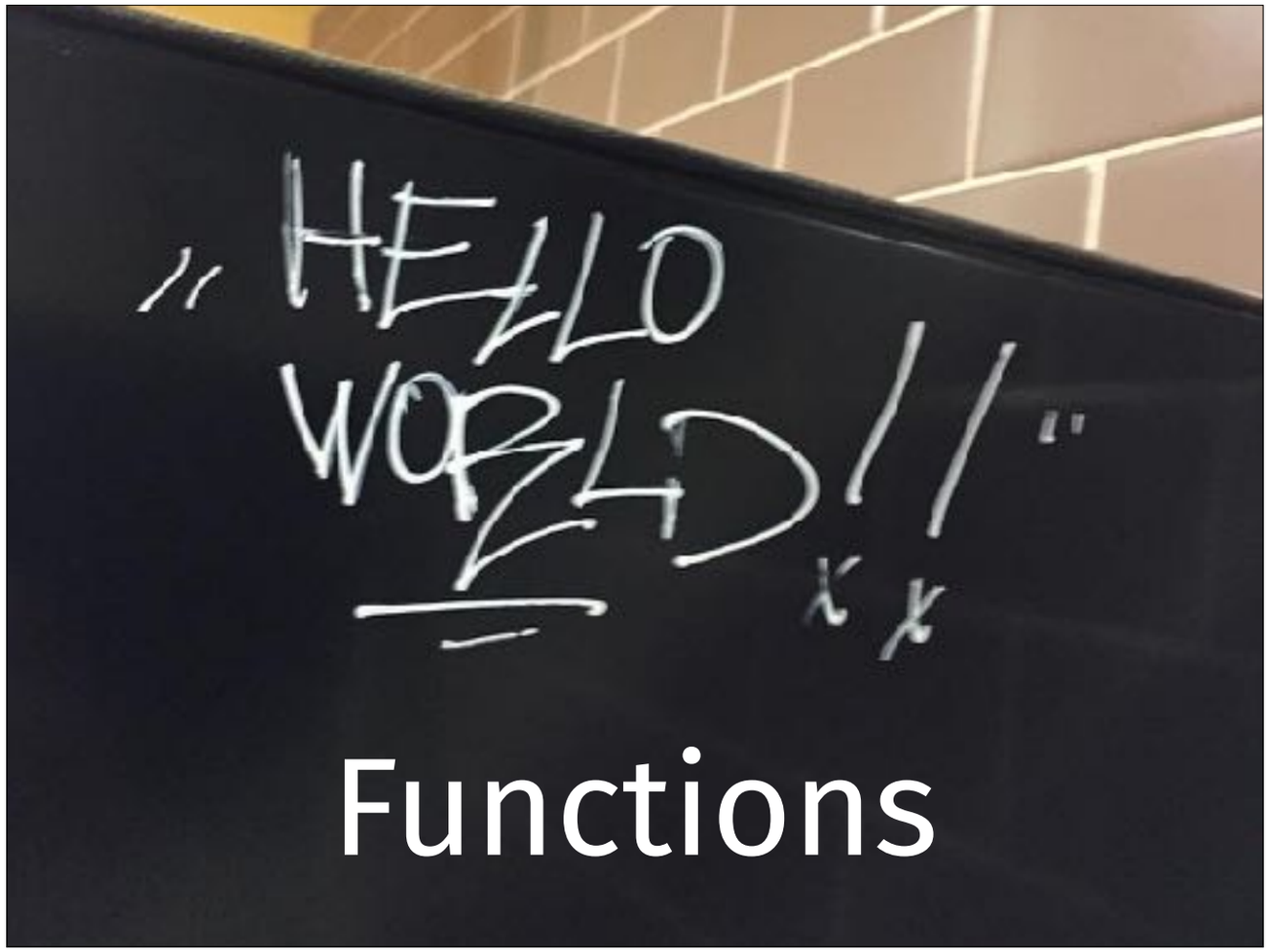
```
var petSet :Set = [ "cat", "dog",
                    "fish", "dog"]

petSet.count // returns 3
```

Sets

```
var petSet :Set = [ "cat", "dog",
                    "fish", "dog"]

petSet.count // returns 3
```

Sets

```
var petSet :Set = [ "cat", "dog",
                    "fish", "dog"]

petSet.count // returns 3
```



Functions

Functions

```
func obExample () {
  print("Hello, KCDC!")
}

// call like:
obExample()
```

Functions

```
func obExample (who :String) {
  print("Hello, \(who)!")
}

// call like
obExample(who: "KCDC 2017")
```

Functions

```
func obExample (who :String) {
  print("Hello, \(who)!")
}

// call like
obExample(who: "KCDC 2017")
```

Functions

```
func obExample (who :String) {
  print("Hello, \(who)!")
}

// call like
obExample(who: "KCDC 2017")
```

Functions

```
func obExample (who :String) -> String {
  return "Hello, \(who)!"
}

// call like
let greets = obExample(who: "KCDC")

// greets == "Hello, KCDC"
```

Functions

```
func obExample (who :String) -> String {
  return "Hello, \(who)!"
}

// call like
let greets = obExample(who: "KCDC")

// greets == "Hello, KCDC"
```

Functions

```
func obExample (who :String = "Swift") -> String {
  return "Hello, \(who)!"
}

// call like
let greets = obExample(who:"KCDC")
// "Hello, KCDC!"

let defGreets = obExample()
// "Hello, Swift!"
```

Functions

```
func obExample (who :String = "Swift") -> String {
  return "Hello, \(who)!"
}

// call like
let greets = obExample(who:"KCDC")
// "Hello, KCDC!"

let defGreets = obExample()
// "Hello, Swift!"
```

Functions

```
func obExample (who :String = "Swift") -> String {
  return "Hello, \(who)!"
}

// call like
let greets = obExample(who:"KCDC")
// "Hello, KCDC!"

let defGreets = obExample()
// "Hello, Swift!"
```

Functions

```
func obExample (who :String = "Swift") -> String {
  return "Hello, \(who)!"
}

// call like
let greets = obExample(who:"KCDC")
// "Hello, KCDC!"

let defGreets = obExample()
// "Hello, Swift!"
```

Functions

```
func obExample (what :String = "Hello",
                who them :String = "Swift") {
  print("\(what), \(them)!")
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who: "Felicia") // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                who them :String = "Swift") {
  print("\(what), \(them)!")
}

// call like
obExample(what: "bye")                 // "bye, Swift!"
obExample(what: "bye", who: "Felicia") // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                who them :String = "Swift") {
  print("\(what), \(them)!")
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who: "Felicia") // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                who them :String = "Swift") {
  print("\(what), \(them)!")
}

// call like
obExample(what: "bye")                  // "bye, Swift!"
obExample(what: "bye", who: "Felicia") // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                who them :String = "Swift") {
  print("\(what), \(them)!")
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who: "Felicia") // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                _ who :String = "Swift") {
  print "\(what), \(who)!"
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who:"Felicia") // COMPILE ERROR
obExample(what: "bye", "Felicia")     // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                _ who :String = "Swift") {
  print "\(what), \(who)!"
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who:"Felicia") // COMPILE ERROR
obExample(what: "bye", "Felicia")     // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                _ who :String = "Swift") {
  print "\(what), \(who)!"
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who:"Felicia") // COMPILE ERROR
obExample(what: "bye", "Felicia")     // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                _ who :String = "Swift") {
  print "\(what), \(who)!"
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who:"Felicia") // COMPILE ERROR
obExample(what: "bye", "Felicia")     // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                _ who :String = "Swift") {
  print "\(what), \(who)!"
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who:"Felicia") // COMPILE ERROR
obExample(what: "bye", "Felicia")     // "bye, Felicia!"
```

Functions

```
func obExample (what :String = "Hello",
                _ who :String = "Swift") {
  print "\(what), \(who)!"
}

// call like
obExample(what: "bye")                // "bye, Swift!"
obExample(what: "bye", who:"Felicia") // COMPILE ERROR
obExample(what: "bye", "Felicia")     // "bye, Felicia!"
```

Functions

```
func variadiacExample (nums: Int...)  {
    // do something with nums
    // nums is Array[Int]
}
```

Functions

```
func variadiacExample (nums: Int...)  {
    // do something with nums
    // nums is Array[Int]
}
```


Functions are first-class citizens

Closures

```
let numbers = [2,1,56,32,120,13]

var sorted = numbers.sorted(by:{
  (n1: Int, n2: Int) -> Bool in return n2 > n1
})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

var sorted = numbers.sorted(by:{
  (n1: Int, n2: Int) -> Bool in return n2 > n1
})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

var sorted = numbers.sorted(by:{
  (n1: Int, n2: Int) -> Bool in return n2 > n1
})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

var sorted = numbers.sorted(by:{
  (n1: Int, n2: Int) -> Bool in return n2 > n1
})

// sorted = [1, 2, 13, 32, 56, 120]
```

This is already the func sig for `sorted(by:)`!

Closures

```
let numbers = [2,1,56,32,120,13]

// inferred param & return types
var sorted = numbers.sorted(by: {n1, n2 in return n2 > n1})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

// inferred param & return types
var sorted = numbers.sorted(by: {n1, n2 in return n2 > n1})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

// positionally named parameters
var sorted = numbers.sorted(by: {return $0 > $1})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

// positionally named parameters
var sorted = numbers.sorted(by: {return $0 > $1})

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

// when closure is last param,
// param name & parens optional
var sorted = numbers.sorted { $0 > $1 }

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
let numbers = [2,1,56,32,120,13]

// when closure is last param,
// param name & parens optional
var sorted = numbers.sorted { $0 > $1 }

// sorted = [1, 2, 13, 32, 56, 120]
```

Closures

```
var sorted = numbers.sorted(by:{
  (n1: Int, n2: Int) -> Bool in return n2 > n1
})
```

```
var sorted = numbers.sorted { $0 > $1 }
```

OOP is also well supported

Classes

```
class Dog {

}
```

Properties

```
class Dog {
    var name: String
    let noise = "WOOF!"
}
```


just variable declarations inside the class

Properties

```
class Dog {
    var name: String
    let noise = "WOOF!"
}
```

Properties

```
class Dog {
    var name: String
    let noise = "WOOF!"
}
```

Class ‘Dog’ has no initializers

...because you're not allowed to have uninitialized properties.

Properties

```
class Dog {
    var name: String?
    let noise = "WOOF!"
}
```


you can fix that by making the property an optional type

Properties

```
class Dog {
    var name: String?
    let noise = "WOOF!"
}
```


you can fix that by making the property an optional type

Initializers

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }
}
```


or by providing an initializer that assigns a value to the property

Initializers

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }
}
```


or by providing an initializer that assigns a value to the property

Initializers

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }
}
```


or by providing an initializer that assigns a value to the property

Deinitializers

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }

    deinit () {
        // do any cleanup here
    }
}
```


or by providing an initializer that assigns a value to the property

Deinitializers

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }

    deinit () {
        // do any cleanup here
    }
}
```


or by providing an initializer that assigns a value to the property

Methods

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }

    func speak () -> String {
        return self.noise
    }
}
```

Methods

```
class Dog {
    var name: String
    let noise = "WOOF"

    init (name: String) {
        self.name = name
    }

    func speak () -> String {
        return self.noise
    }
}
```

Using Objects

```
let sammy = Dog(name: "Sammy");

// sammy is Dog

print(sammy.name)     // prints "Sammy\n"

print(sammy.speak())  // prints "WOOF!\n"

sammy.name = "Buster" // works b/c prop is var
```

Using Objects

```
let sammy = Dog(name: "Sammy");

// sammy is Dog

print(sammy.name)     // prints "Sammy\n"

print(sammy.speak())  // prints "WOOF!\n"

sammy.name = "Buster" // works b/c prop is var
```

Using Objects

```
let sammy = Dog(name: "Sammy");

// sammy is Dog

print(sammy.name)      // prints "Sammy\n"

print(sammy.speak())   // prints "WOOF!\n"

sammy.name = "Buster" // works b/c prop is var
```

Using Objects

```
let sammy = Dog(name: "Sammy");

// sammy is Dog

print(sammy.name)     // prints "Sammy\n"

print(sammy.speak())  // prints "WOOF!\n"

sammy.name = "Buster" // works b/c prop is var
```

Using Objects

```
let sammy = Dog(name: "Sammy");

// sammy is Dog

print(sammy.name)     // prints "Sammy\n"

print(sammy.speak())  // prints "WOOF!\n"

sammy.name = "Buster" // works b/c prop is var
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set {
      // this is horrible, don't do this
      self.birthYear = currentYear - newValue
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set {
      // this is horrible, don't do this
      self.birthYear = currentYear - newValue
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set {
      // this is horrible, don't do this
      self.birthYear = currentYear - newValue
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set {
      // this is horrible, don't do this
      self.birthYear = currentYear - newValue
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set (age) {
      // this is horrible, don't do this
      self.birthYear = currentYear - age
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set (age) {
      // this is horrible, don't do this
      self.birthYear = currentYear - age
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    get {
      return currentYear - self.birthYear
    }
    set (age) {
      // this is horrible, don't do this
      self.birthYear = currentYear - age
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    willSet {
      // runs before property value changes
    }
    didSet {
      // runs after property value changes
    }
  }
}
```

Computed Properties

```
class Dog {
  var age :Int {
    willSet {
      // runs before property value changes
    }
    didSet {
      // runs after property value changes
    }
  }
}
```

Inheritance

Inheritance

```
class Animal {

}

class Dog : Animal {

}
```

Inheritance

```
class Animal {

}

class Dog : Animal {

}
```

Inheritance

```
class Animal {
  let name: String

  init (name: name) {
    self.name = name
  }
}

class Dog : Animal {
  override init (name: name) {
    super.init(name: name)
  }
}
```

Inheritance

```
class Animal {
  let name: String

  init (name: name) {
    self.name = name
  }
}

class Dog : Animal {
  override init (name: name) {
    super.init(name: name)
  }
}
```

Inheritance

```
class Animal {
  let name: String

  init (name: name) {
    self.name = name
  }
}

class Dog : Animal {
  override init (name: name) {
    super.init(name: name)
  }
}
```

Overrides

```
class Animal {
  func speak() { ... }
}

class Dog : Animal {
  override func speak () { ... }
}
```

Overrides

```
class Animal {
  func speak() { ... }
}

class Dog : Animal {
  override func speak () { ... }
}
```

Structs & Enumerations

Structs

```
struct Animal {
  var name: String
  var noise: String

  init (name: String, makes: String) {
    self.name = name
    noise = makes
  }

  func speak () -> String {
    return noise
  }
}
```


very much like classes -- support properties, initializers, methods, etc.

big difference: structs are pass-by-value; classes/objects are pass-by-reference

Structs

```
struct Animal {
  var name: String
  var noise: String

  init (name: String, makes: String) {
    self.name = name
    noise = makes
  }

  func speak () -> String {
    return noise
  }
}
```


very much like classes -- support properties, initializers, methods, etc.

big difference: structs are pass-by-value; classes/objects are pass-by-reference

Structs

```
struct Animal {
  var name: String
  var noise: String

  init (name: String, makes: String) {
    self.name = name
    noise = makes
  }

  func speak () -> String {
    return noise
  }
}
```


very much like classes -- support properties, initializers, methods, etc.

big difference: structs are pass-by-value; classes/objects are pass-by-reference

Structs

```
struct Animal {
  var name: String
  var noise: String

  init (name: String, makes: String) {
    self.name = name
    noise = makes
  }

  func speak () -> String {
    return noise
  }
}
```


very much like classes -- support properties, initializers, methods, etc.

big difference: structs are pass-by-value; classes/objects are pass-by-reference

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest
}

var conf = OpenSourceConfs.KCDC
// conf is type OpenSourceConfs
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest
}

var conf = OpenSourceConfs.KCDC
// conf is type OpenSourceConfs
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest
}

var conf = OpenSourceConfs.KCDC
// conf is type OpenSourceConfs
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest
}

var conf = OpenSourceConfs.KCDC
// conf is type OpenSourceConfs
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs :Int {
  case KCDC = 1
  case OpenWest
}

var conf = OpenSourceConfs.OpenWest
// conf is type OpenSourceConfs

conf.rawValue // 2
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs :Int {
  case KCDC = 1
  case OpenWest
}

var conf = OpenSourceConfs.OpenWest
// conf is type OpenSourceConfs

conf.rawValue // 2
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs :Int {
  case KCDC = 1
  case OpenWest
}

var conf = OpenSourceConfs.OpenWest
// conf is type OpenSourceConfs

conf.rawValue // 2
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs :Int {
  case KCDC = 1
  case OpenWest
}

var conf = OpenSourceConfs.OpenWest
// conf is type OpenSourceConfs

conf.rawValue // 2
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC:
        return "Hello Kansas City!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC:
        return "Hello Kansas City!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC:
        return "Hello Kansas City!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC:
        return "Hello Kansas City!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC(String)
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC(let location):
        return "Hello \(location)!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC("Olathe")
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC(String)
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC(let location):
        return "Hello \(location)!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC("Olathe")
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC(String)
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC(let location):
        return "Hello \(location)!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC("Olathe")
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC(String)
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC(let location):
        return "Hello \(location)!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }
}

var conf = OpenSourceConfs.KCDC("Olathe")
conf.describe()
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC(String)
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC (let location):
        return "Hello \(location)!"
      case .OpenWest:
        return "Hello Salt Lake City!"
    }
  }

}

var kcdc2017 = OpenSourceConfs.KCDC("KC")
var kcdc2018 = OpenSourceConfs.KCDC("Olathe")
```


enums provide a constrained set of enumerated values

Enumerations

```
enum OpenSourceConfs {
  case KCDC(String)
  case OpenWest

  func describe() -> String {
    switch self {
      case .KCDC (let location):
        return "Hello \(location)!"
      case .OpenWest
        return "Hello Salt Lake City!"
    }
  }
}

var kcdc2017 = OpenSourceConfs.KCDC("KC")
var kcdc2018 = OpenSourceConfs.KCDC("Olathe")
```


enums provide a constrained set of enumerated values

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker {
  var noise: String { get }
  func talk () -> String
  mutating func mute()
}

class Dog: Talker {
  var noise: String
  init (noise: String) {
    self.noise = noise
  }
  func talk () -> String {
    return noise
  }
  func mute () {
    noise = ""
  }
}
```

Protocols

```
protocol Talker { ... }

class Dog: Talker { ... }

class Fish { ... }

var sammy: Talker

sammy = Dog(noise: "WOOF!")

sammy = Fish()  // compile error
```


protocols can be used as types in variable declarations

Protocols

```
protocol Talker { ... }

class Dog: Talker { ... }

class Fish { ... }

var sammy: Talker

sammy = Dog(noise: "WOOF!")

sammy = Fish()  // compile error
```


protocols can be used as types in variable declarations

Protocols

```
protocol Talker { ... }

class Dog: Talker { ... }

class Fish { ... }

var sammy: Talker

sammy = Dog(noise: "WOOF!")

sammy = Fish()  // compile error
```


protocols can be used as types in variable declarations

Protocols

```
protocol Talker { ... }

class Dog: Talker { ... }

class Fish { ... }

var sammy: Talker

sammy = Dog(noise: "WOOF!")

sammy = Fish()  // compile error
```


protocols can be used as types in variable declarations

Extensions

```
extension Int {
  func squared () -> Int {
    return self * self
  }
}

let foo = 2

print(foo.squared())           //  4
print(foo.squared().squared()) // 16
```


extensions can be used to extend built-in types to allow them to conform to protocols

Extensions

```
extension Int {
  func squared () -> Int {
    return self * self
  }
}

let foo = 2

print(foo.squared())           //  4
print(foo.squared().squared()) // 16
```


extensions can be used to extend built-in types to allow them to conform to protocols

Extensions

```
extension Int {
  func squared () -> Int {
    return self * self
  }
}

let foo = 2

print(foo.squared())            //  4
print(foo.squared().squared()) // 16
```


extensions can be used to extend built-in types to allow them to conform to protocols

Extensions

```
extension Int {
  func squared () -> Int {
    return self * self
  }
}

let foo = 2

print(foo.squared())          //  4
print(foo.squared().squared()) // 16
```


extensions can be used to extend built-in types to allow them to conform to protocols

Exceptions & Error Handling

Exceptions

```
enum TalkErrors: Error {
  case TooShort
  case TooLong
  case TooBoring
}

func giveATalk (talk: String) throws -> String {
  if talk == "boring" {
    throw TalkErrors.TooBoring
  }
  return "talk!"
}
```

Exceptions

```
enum TalkErrors: Error {
  case TooShort
  case TooLong
  case TooBoring
}

func giveATalk (talk: String) throws -> String {
  if talk == "boring" {
    throw TalkErrors.TooBoring
  }
  return "talk!"
}
```

Exceptions

```
enum TalkErrors: Error {
  case TooShort
  case TooLong
  case TooBoring
}

func giveATalk (talk: String) throws -> String {
  if talk == "boring" {
    throw TalkErrors.TooBoring
  }
  return "talk!"
}
```

Exceptions

```
enum TalkErrors: Error {
  case TooShort
  case TooLong
  case TooBoring
}

func giveATalk (talk: String) throws -> String {
  if talk == "boring" {
    throw TalkErrors.TooBoring
  }
  return "talk!"
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "boring")
  print(thisTalk)
}
catch {
  print(error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "boring")
  print(thisTalk)
}
catch {
  print(error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "boring")
  print(thisTalk)
}
catch {
  print(error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "boring")
  print(thisTalk)
}
catch {
  print(error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "fine")
  print(thisTalk)
}
catch TalkErrors.TooLong {
  print("shut up already")
}
catch let talkError as TalkErrors {
  print("Talk error: \(talkError).")
}
catch {
  print (error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "fine")
  print(thisTalk)
}
catch TalkErrors.TooLong {
  print("shut up already")
}
catch let talkError as TalkErrors {
  print("Talk error: \(talkError).")
}
catch {
  print (error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "fine")
  print(thisTalk)
}
catch TalkErrors.TooLong {
  print("shut up already")
}
catch let talkError as TalkErrors {
  print("Talk error: \(talkError).")
}
catch {
  print (error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "fine")
  print(thisTalk)
}
catch TalkErrors.TooLong {
  print("shut up already")
}
catch let talkError as TalkErrors {
  print("Talk error: \(talkError).")
}
catch {
  print (error)
}
```

Exceptions

```
do {
  let thisTalk = try giveATalk(talk: "fine")
  print(thisTalk)
}
catch TalkErrors.TooLong {
  print("shut up already")
}
catch let talkError as TalkErrors {
  print("Talk error: \(talkError).")
}
catch {
  print (error)
}
```

Exceptions

```
// silently discards error
let thisTalk = try? giveATalk(talk:"fine")

// thisTalk isa String?
```

Exceptions

```
// silently discards error
let thisTalk = try? giveATalk(talk:"fine")

// thisTalk isa String?
```

Exceptions

```
// silently discards error
let thisTalk = try? giveATalk(talk:"fine")

// thisTalk isa String?
```

Defer

```
func needsMuchSetupAndTearDown () {
  // do the setup here, open files, etc.
  defer {
    // and do the cleanup here,
    // right next to set up
  }

  // other code here.
}
```

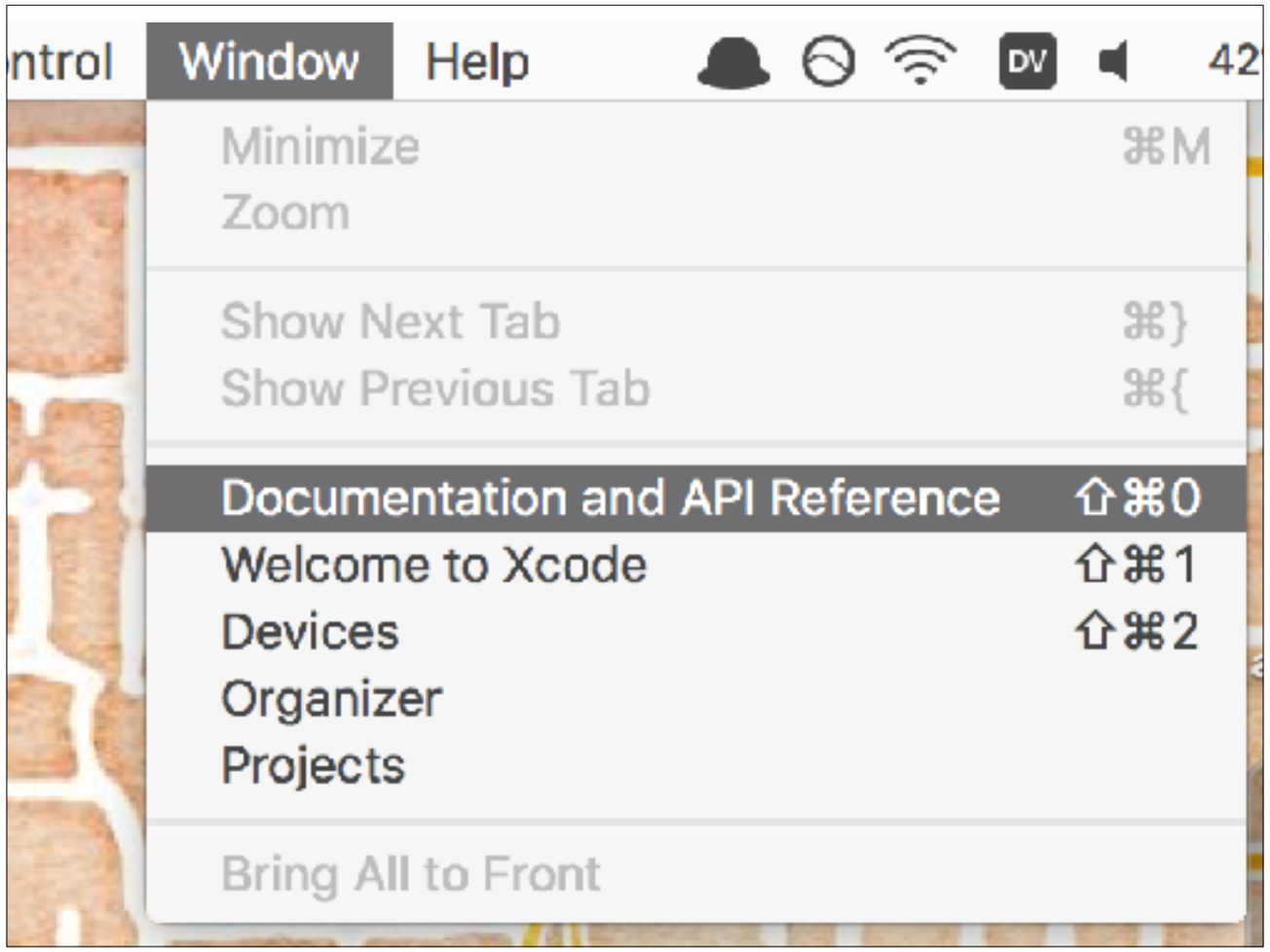
Defer

```
func needsMuchSetupAndTearDown () {
  // do the setup here, open files, etc.
  defer {
    // and do the cleanup here,
    // right next to set up
  }

  // other code here.
}
```

Workspaces


Ill-advised
live
demo
time!


dvised
live
mo
e!


ntrol
Window
Help
42
Minimize ⌘M
Zoom
Show Next Tab ⌘}
Show Previous Tab ⌘{
Documentation and API Reference ⇧⌘0
Welcome to Xcode ⇧⌘1
Devices ⇧⌘2
Organizer
Projects
Bring All to Front

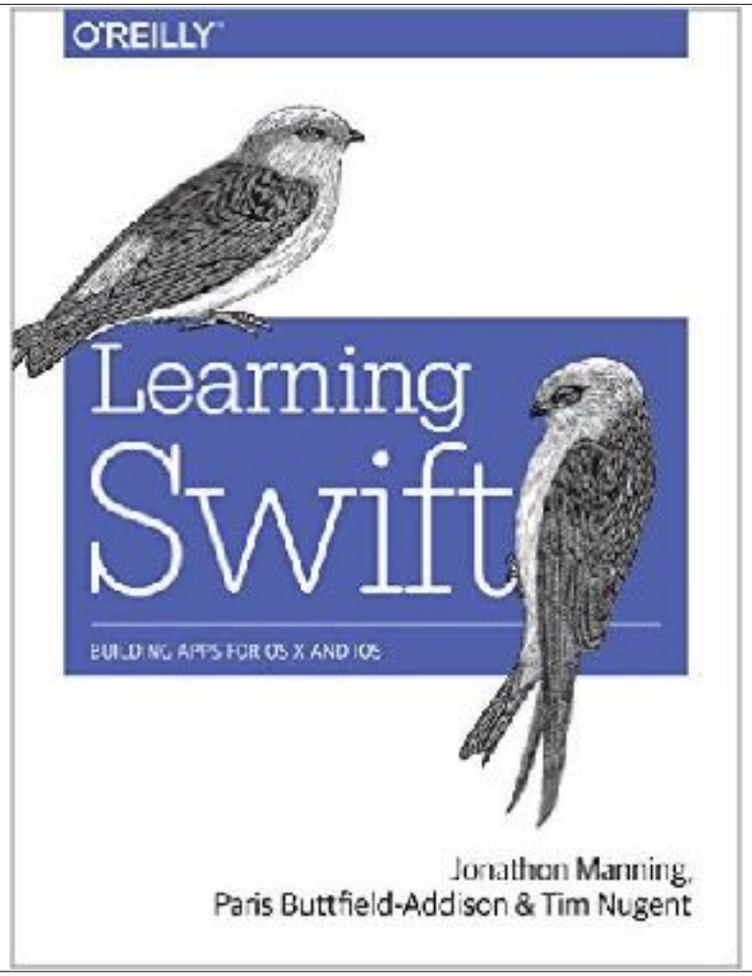
A Swift Tour

Tradition suggests that the first program in a new language should print the words "Hello, world!" on the screen. In Swift, this can be done in a single line:

```
print("Hello, world!")
```

"Hello, world!\n"

https://developer.apple.com/swift


O'REILLY
Learning
Swift
BUILDING APPS FOR OS X AND iOS
Jonathon Manning,
Paris Buttfield-Addison & Tim Nugent


Christian Gloddy
@gloddy
Follow
How to end your technical presentation:
1. Yell "FREE PIXELS FOR EVERYONE!"
2. Throw fistfuls of glitter into the crowd.
3. Run.
RETWEETS
424
FAVORITES
464
9:22 AM - 1 Mar 2015

thanks!


infinityinteractive
technical, not typical
partner, not vendor
We solve problems with technology
consulting
development
web
mobile
devops

SeaGL

GIMME YR TALKS **CFP OPEN NOW!**

Oct 6-7

seagl.org


Christian Gloddy
@gloddy
Follow
How to end your technical presentation:
1. Yell "FREE PIXELS FOR EVERYONE!"
2. Throw fistfuls of glitter into the crowd.
3. Run.
RETWEETS
424
FAVORITES
464
9:22 AM - 1 Mar 2015

questions?