

Intro to Swift

Lecture 2

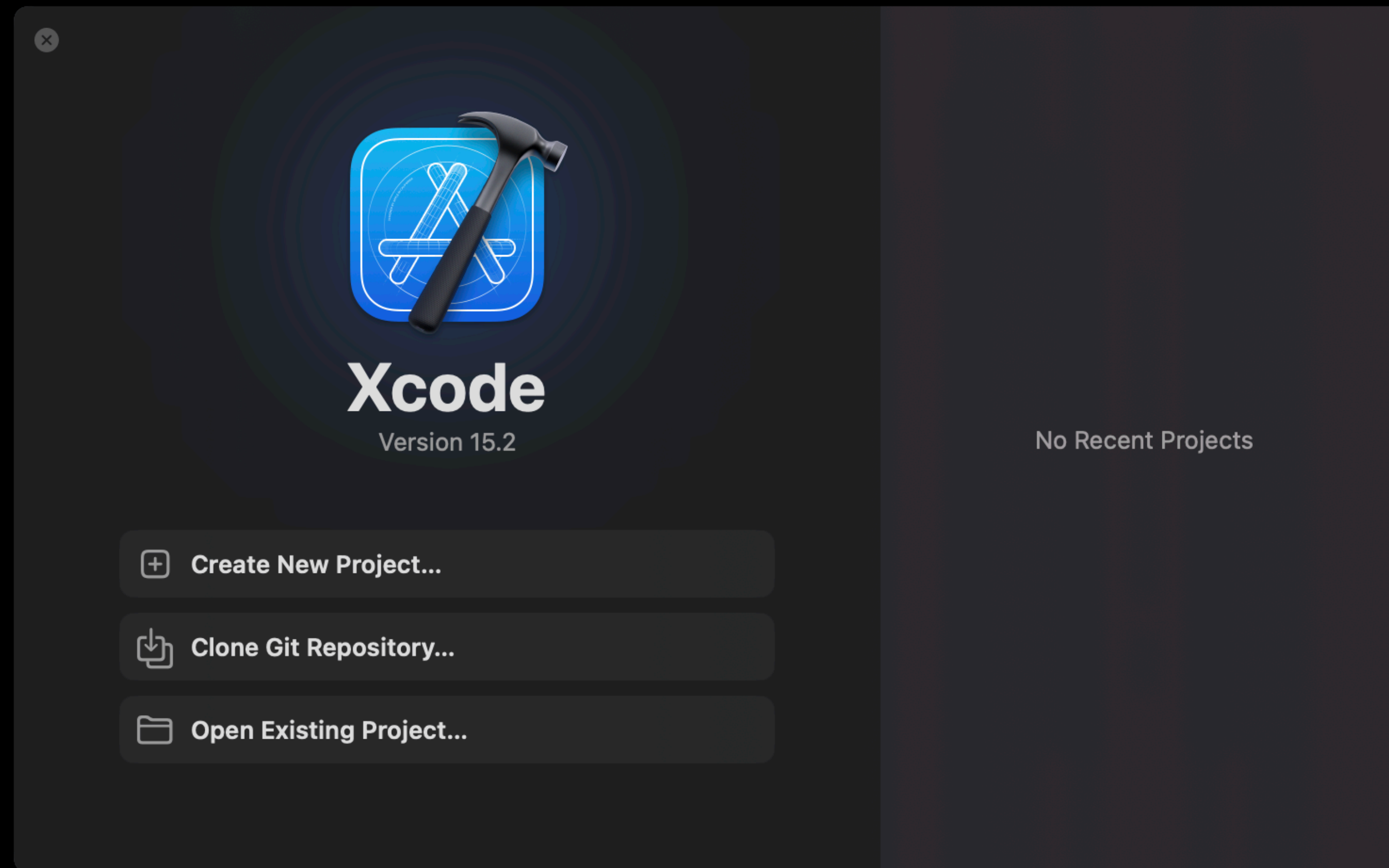
CIS 1951

Any questions, comments, or concerns from last week?

We value your feedback!

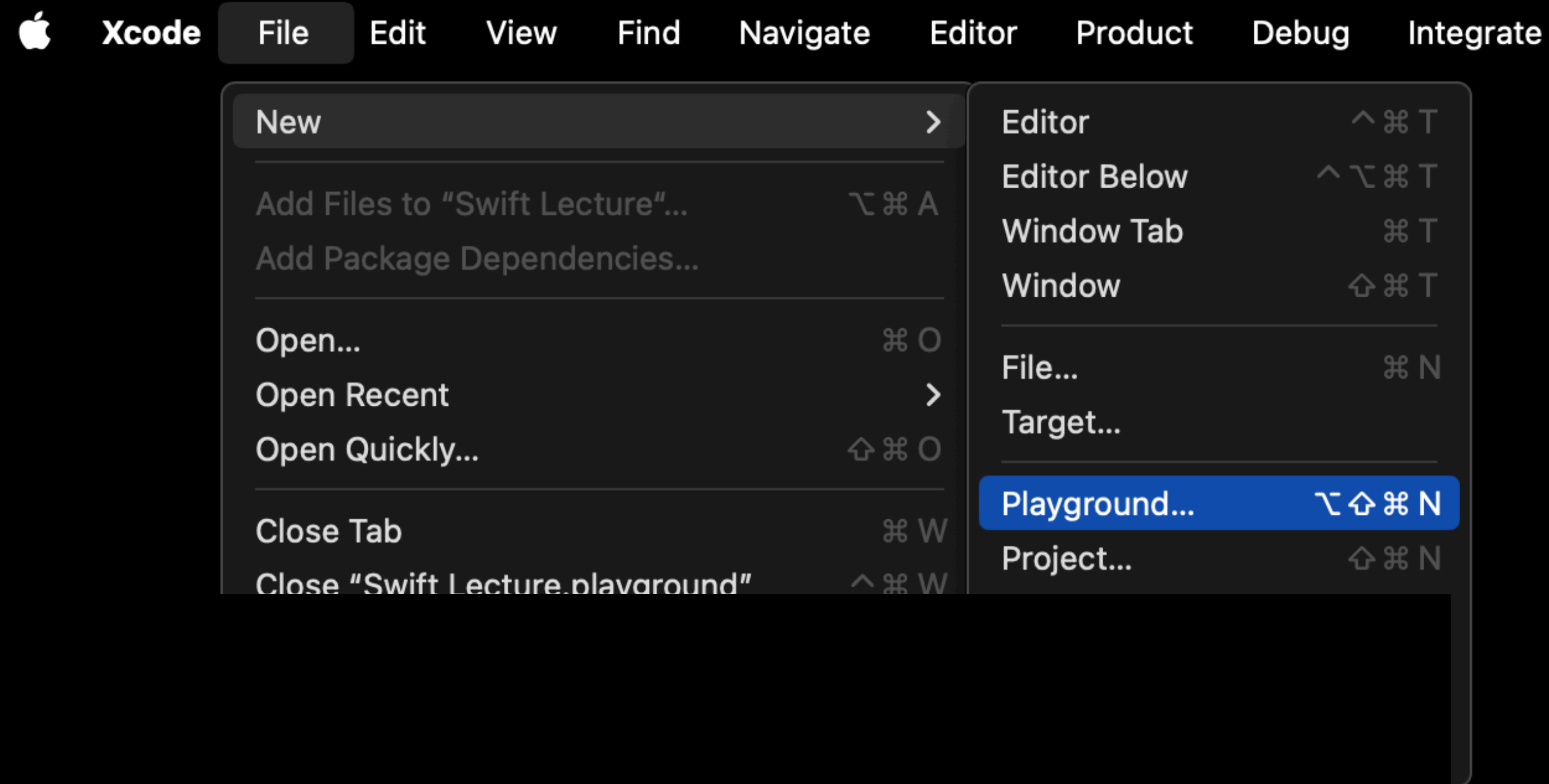
(please)

Let's get started



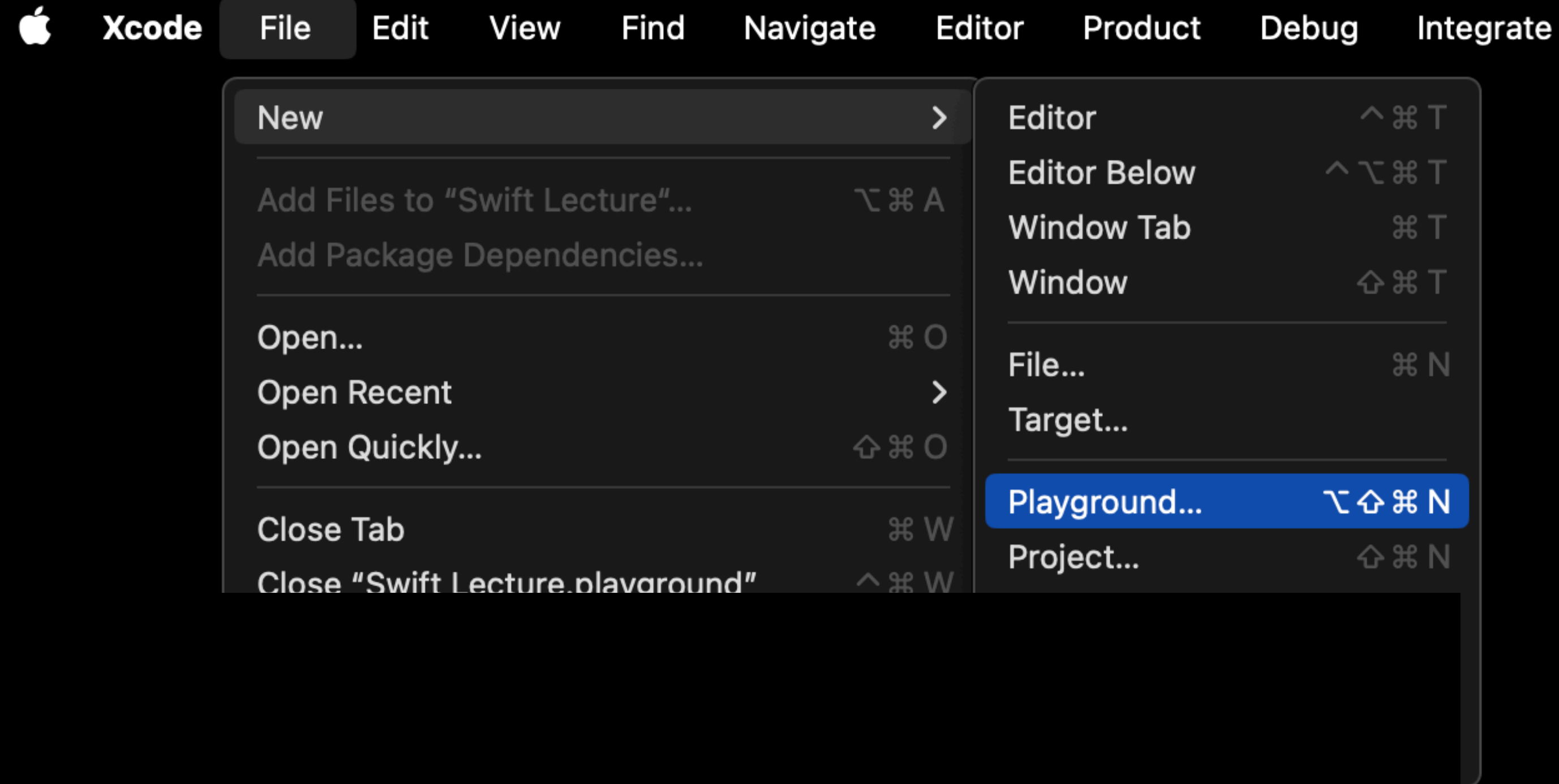
Fire up Xcode

Let's get started



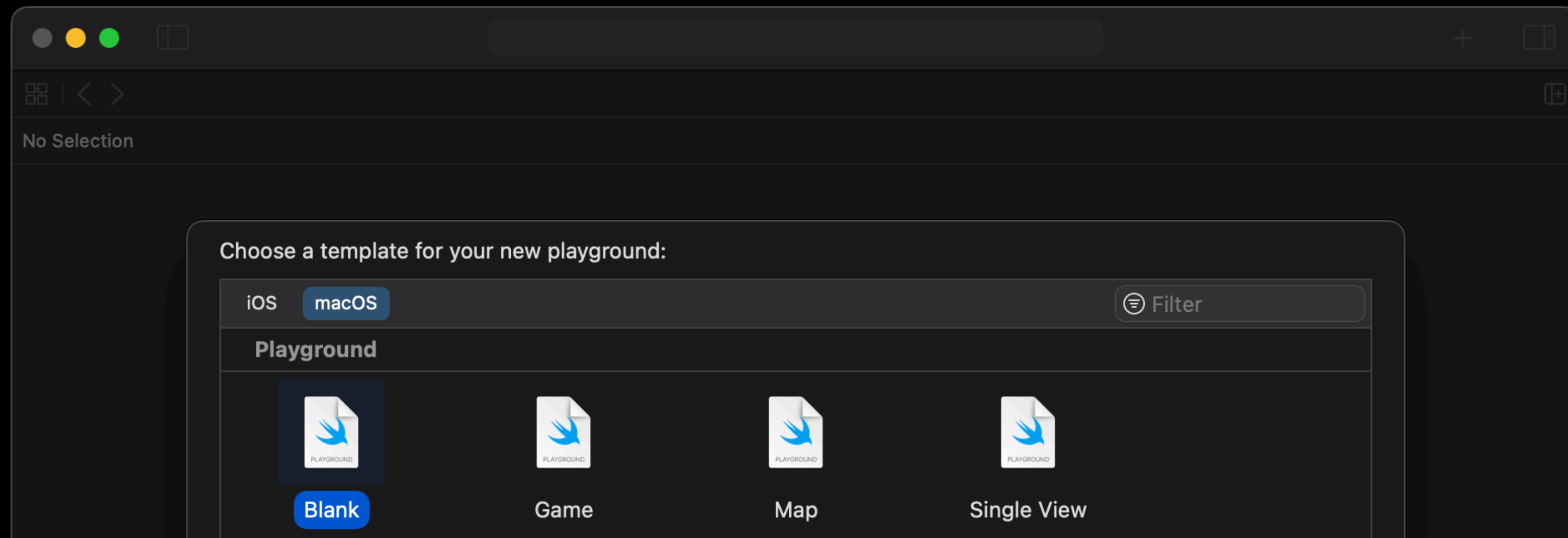
Create a new **playground**

Let's get started



Create a new **playground**

Let's get started



Choose **Blank**

Your new playground

Status



The screenshot shows the Swift Playground interface. The code editor contains the following Swift code:

```
1 import Cocoa
2
3 let url = URL(string:
4     "https://www.youtube.com/watch?v=xvFZjo5PgG0")!
5     NSLog("Enjoy!")
6
7
```

The code is running, and the console output on the right shows the following return values:

- ☐ https://www.youtube....
- ☐ true
- ☐ "Enjoy!\n"

Annotations with arrows point to specific UI elements:

- Run up to line**: Points to the play button icon on the left margin next to line 5.
- Stop & reset**: Points to the square button icon in the bottom left corner.
- Return values**: Points to the list of return values on the right side.
- Console output**: Points to the console output area at the bottom right.

First things first

```
// The obligatory first line  
print("Hello, world!")
```

When you're ready to run your
code, click [here](#)



The Basics

Variables

let and var

```
let name = "Anthony"  
var section = 201
```

Variables

let and var

```
let name = "Anthony"
```

```
name = "Yuying" ❌ Cannot assign to value: 'name' is a 'let' constant
```

```
var section = 201
```

```
section = 202 ✅
```

Type Inference

What's wrong with this code?

```
var x = 5  
x = 5.9
```

Type Inference

What's wrong with this code?

Compiler infers that `x` is an `Int`

`var x = 5`


`x = 5.9` ❌ Cannot assign value of type 'Double' to type 'Int'

Type Inference

Specifying Types Explicitly

Explicit type annotation

```
var x: Double = 5  
x = 5.9
```



⚠ This is not the same as *casting* an Int to a Double!

Some Basic Types

Int

$2 + 2 // 4$

$2 * 4 // 8$

$7 / 3 // 2$

$\text{Int.min} // -2^{(63)}$

$\text{Int.max} // 2^{(63)} - 1$

Some Basic Types

Double

`2.0 + 5.5 // 7.5`

`2.0 * 5.5 // 11.0`

`7.0 / 3.0 // 2.33333...`

`let int: Int = 3`

`7.0 / Double(int) // 2.33333...`

Some Basic Types

String

```
"This is a string!"
```

```
let name = "Jordan"
```

```
"Hello, " + name + "!" // "Hello, Jordan!"
```

```
"Hello, \(name)!" // "Hello, Jordan!"
```



Interpolation

Some Basic Types

Array

```
var foods: [String] = ["Penn Dining", "Allegro's"]
foods.count // 2
foods[0] // "Penn Dining"
foods[0] = "🤮"

foods.append("Terakawa")
foods.count // 3
foods[2] // "Terakawa"

foods // ["🤮", "Allegro's", "Terakawa"]
```

Some Basic Types

Dictionary

```
let violations = [  
  "1920 Commons": 38,  
  "Hill House": 21  
]
```

```
violations["1920 Commons"] // 38  
violations["Not Penn Dining"] // nil
```



More on this later...

Some Basic Types

Empty Arrays & Dictionaries

```
let array1 = [String]()
```

```
let array2: [String] = []
```

```
let dict1 = [String: Int]()
```

```
let dict2: [String: Int] = [:]
```



Why do we need this?

Some Basic Types

What's the type?

```
let a = [1, 2, 3, 4, 5]
```

```
let b = [1: "one", 2: "two", 3: "three"]
```

```
let c = [[1, 3], [2, 4]]
```

```
let d = [1: ["one", "1"], 2: ["two", "2"]]
```

Control Flow

Control Flow

If

```
let quantity = 1
```

```
if quantity > 0 {  
    print("Thanks for your purchase!")  
} else if quantity == 0 {  
    print("Look, at least buy *something*.")  
} else {  
    print("Nice try. Now please leave the store.")  
}
```

Control Flow

Switch

```
let course = "CIS 3200"
switch course {
case "CIS 1600":
    print("Oh no")
case "CIS 1210":
    print("Oh nooooo")
case "CIS 3200":
    print("Time to drop out")
default:
    print("Course not found")
}
```


Control Flow

While

```
var i = 0
while i < 5 {
    print(i)
    i += 1
}
```

0
1
2
3
4

Control Flow

For

```
for i in 0..<5 {  
    print(i)  
}
```

0
1
2
3
4

Control Flow

For

```
let names = ["Anthony", "Jordan", "Yuying"]  
for name in names {  
    print("Hello, \(name)!")  
}
```

```
Hello, Anthony!  
Hello, Jordan!  
Hello, Yuying!
```

Optionals

```
violations["Not Penn Dining"] // nil
```

What's *nil*?

Optionals

Remember CIS 1200?

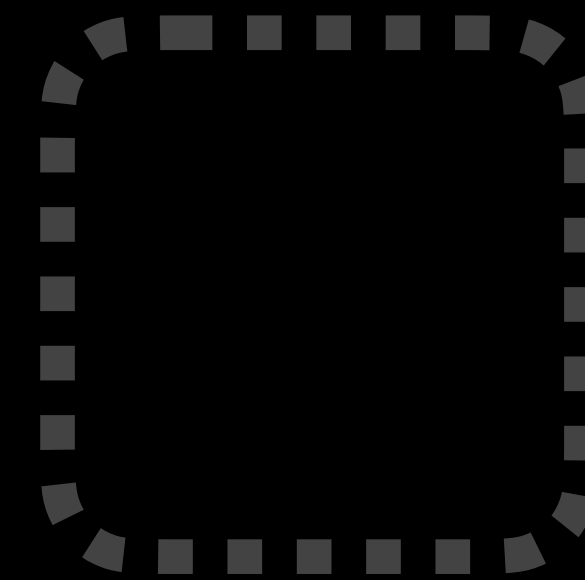
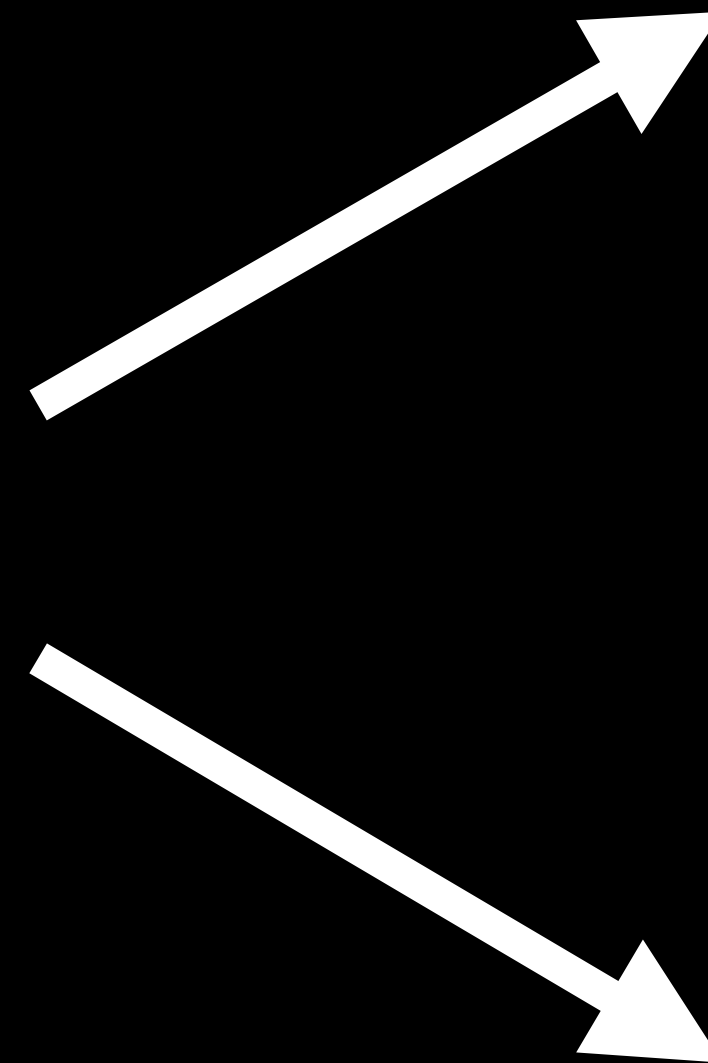
```
type 'a option =  
  | None  
  | Some of 'a
```

Optionals

A wrapper type



Int?



nil

OR



**A concrete
value**

Optionals

Motivation



I call it **my billion-dollar mistake**. It was the invention of the null reference in 1965... This has led to **innumerable errors, vulnerabilities, and system crashes**, which have probably caused a billion dollars of pain and damage in the last forty years.

- *Sir Charles Anthony (Tony) Hoare*

Optionals

Defining & Unwrapping

```
let optional: String? = "hi"
```

“Force unwrapping” `optional! // 2`

“Optional chaining” `optional?.count // Optional(2)`

“Optional binding”

```
if let str = optional {  
    str.count // 2  
}
```


Optionals

Defining & Unwrapping

```
let optional: String? = nil
```

“Force unwrapping” `optional! // FATAL ERROR`

“Optional chaining” `optional?.count // nil`

“Optional binding”

```
if let str = optional {  
    str.count // Doesn't run  
}
```

Optionals

Optional or not?

```
dictionary["generic key"]
```

```
Int.random(in: 1...3)
```

```
array.randomElement()
```

Optionals

A word of warning

`array[0]`

does not return an optional!

It will crash if `array` is empty.

Functions

Functions

The Basics

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

```
greet(name: "Anthony") // Hello, Anthony!
```



Arguments are *labelled*!

Functions

Omitting Labels

Add an underscore



```
func greet(_ name: String) -> String {  
    return "Hello, \(name)!"  
}
```

```
greet("Anthony") // Hello, Anthony!
```

Functions

First-Class Functions

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

```
let names = ["Anthony", "Jordan", "Yuying"]
```

```
// ["Hello, Anthony!", "Hello, Jordan!", "Hello, Yuying!"]  
names.map(greet)
```

Like transform
from CIS 1200

Treated like a
value

Functions

First-Class Functions

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

What type is greet?

`(String) -> String`

Functions

Closures

```
let names = ["Anthony", "Jordan", "Yuying"]
```

```
names.map({ (name: String) in  
    return "Hello, \(name)!"  
})
```

Functions

Closure Shorthand

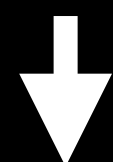
```
names.map({ (name: String) in  
    "Hello, \(name)!"  
})
```

For simple closures, we can **leave out the return**



```
names.map({ name in  
    "Hello, \(name)!"  
})
```

The compiler can **infer parameter types** for us



```
names.map({ "Hello, \($0)!" })
```

\$0 is shorthand for “argument 0”



```
names.map { "Hello, \($0)!" }
```

Trailing closure syntax lets us omit the parentheses

Functions

Closures

Type can be
inferred

```
let names = ["Anthony", "Jordan", "Yuying"]
```

```
names.map({ name in  
    return "Hello, \(name)!"  
})
```

Functions

Closures

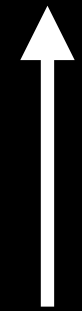
```
let names = ["Anthony", "Jordan", "Yuying"]  
names.map({ "Hello, \($0)!" })
```

↑
Shorthand for
"argument 0"

Functions

Closures

```
let names = ["Anthony", "Jordan", "Yuying"]  
names.map { "Hello, \($0)!" }
```



“Trailing closure
syntax”

Classes, Structs, Enums

Classes

A Basic Class

```
class Receipt {  
    // ...  
}
```

Classes

Properties

```
class Receipt {  
    var items: [String]  
    var amount: Int  
}
```


Classes

Initializers/Constructors

```
class Receipt {  
    var items: [String]  
    var price: Double  
  
    init(items: [String], price: Double) {  
        self.items = items  
        self.price = price  
    }  
}
```

```
Receipt(items: ["🍏 Polishing Cloth"], price: 19.00)
```

Classes

Methods

```
class Receipt {  
    // ...  
    func applyDiscount(percent: Int) {  
        price *= 1 - Double(percent) / 100  
    }  
}
```

```
let oops = Receipt(items: ["Polishing Cloth"], price: 19.00)  
oops.applyDiscount(percent: 20)  
oops.price // 15.2
```

Classes

Computed Properties

```
class Receipt {  
    // ...  
    var numberOfItems: Int {  
        items.count  
    }  
}
```

```
let oops = Receipt(items: ["Polishing Cloth"], price: 19.00)  
oops.numberOfItems // 1
```

Classes

Computed Properties \approx Methods

```
let oops = Receipt(items: ["Polishing Cloth"], price: 19.00)  
oops.numberOfItems // 1
```

```
oops.items.append("Polishing Cloth Travel Case")  
oops.numberOfItems // 2
```

Structs

Enter struct

```
class Receipt {  
    var items: [String]  
    var price: Double  
    // ...  
}
```

```
struct Receipt {  
    var items: [String]  
    var price: Double  
    // ...  
}
```

Pass by reference

Pass by value

Classes vs. Structs

Classes

```
class Receipt {  
    var items: [String]  
    var price: Double  
    // ...  
}  
  
let a = Receipt(items: ["Polishing Cloth"], price: 19.00)  
let b = a  
  
a.price = 1999.00  
b.price // 1999.00
```

Classes vs. Structs

Structs

```
struct Receipt {  
    var items: [String]  
    var price: Double  
    // ...  
}
```

```
var a = Receipt(items: ["Polishing Cloth"], price: 19.00)  
let b = a
```


```
a.price = 1999.00  
b.price // 19.00
```

Something to think about: Why did we have to change `let` to `var`?

Structs

Sidenote: mutating

Needed when modifying a
struct inside one of its
methods



```
struct Receipt {  
    // ...  
    mutating func applyDiscount(percent: Int) {  
        price *= 1 - Double(percent) / 100  
    }  
}
```


Structs

Sidenote: Auto-Generated Initializers

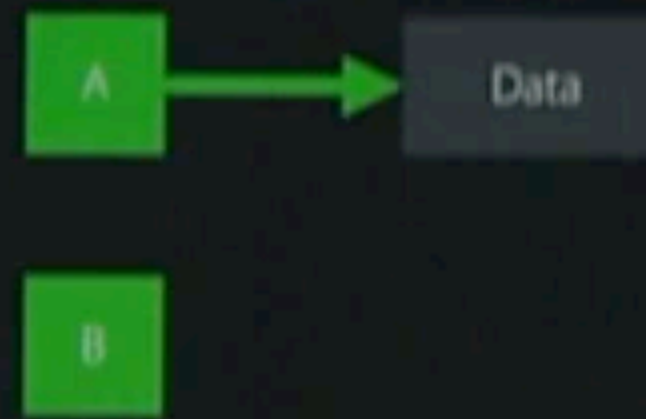
```
struct BankAccount {  
    var funds: Int  
}
```

```
var alice = BankAccount(funds: 0)  
var bob = BankAccount(funds: 100)
```

That's it!

Why do we need structs?

1. Implicit Sharing



“The Crusty Talk”

<https://www.youtube.com/watch?v=p3zo4ptMBiQ&t=359s>



Enumerations

Remember CIS 1200?

```
type cat_state =  
  | Sleeping  
  | Eating  
  | Playing  
  | Hunting
```

Enumerations

```
enum CatState {  
    case sleeping  
    case eating  
    case playing  
    case hunting  
    case slappingTheDog  
    case meowingAtYouToOpenTheDoor  
    case giftingYouALiveMouse  
}
```

```
var myCat = CatState.eating  
myCat = .slappingTheDog
```

Protocols and Extensions

Protocols

Declaration

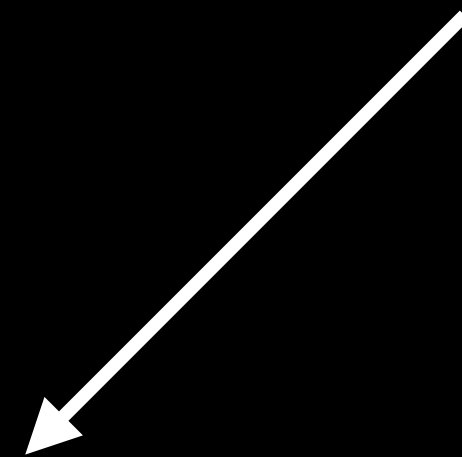
```
protocol ComputerStore {  
    var name: String { get }  
    func buyComputer() -> Receipt  
}
```

Like **interfaces** in Java

Protocols

Implementation

Declaration of
conformance



```
struct MicrosoftStore: ComputerStore {  
    let name = "Microsoft Experience Center"  
    func buyComputer() -> Receipt {  
        return Receipt(  
            items: ["Spyware"],  
            price: 1000.00  
        )  
    }  
}
```


Protocols

Implementation

```
struct LinuxStore: ComputerStore {  
    let name = "???"  
    func buyComputer() -> Receipt {  
        return Receipt(  
            items: ["IBM ThinkPad"],  
            price: 200.00  
        )  
    }  
}
```

Protocols

Implementation

```
struct AppleStore: ComputerStore {  
    let name = "Apple Store"  
    func buyComputer() -> Receipt {  
        fatalError("Bank account overdrawn")  
    }  
}
```

Protocols

Another Example

```
public protocol AdventureGame {  
    init()  
  
    var title: String { get }  
  
    mutating func start(context: AdventureGameContext)  
    mutating func handle(input: String, context: AdventureGameContext)  
}
```

Protocols

Built-in Protocol: Equatable

```
public protocol Equatable {  
    /// Returns a Boolean value indicating whether two values are equal.  
    static func == (lhs: Self, rhs: Self) -> Bool  
}
```

Extensions

Making Receipt Equatable

```
extension Receipt: Equatable {  
    static func ==(_ left: Receipt, _ right: Receipt) -> Bool {  
        return left.items == right.items &&  
            left.price == right.price  
    }  
}
```

```
let a = Receipt(items: ["Water"], price: 1)  
let b = Receipt(items: ["Water"], price: 1)  
a == b // Now works!
```

Error Handling

Error Handling

Defining Errors

```
enum BankError: Error {  
    case insufficientFunds  
}
```

Error Handling

Throwing Errors

```
struct BankAccount {  
    var funds: Int  
  
    mutating func transfer(amount: Int,  
                           to destination: inout BankAccount) throws {  
        if funds < amount {  
            throw BankError.insufficientFunds  
        }  
  
        destination.funds += amount  
        funds -= amount  
    }  
}
```

Needed to throw
errors



Disclaimer: If you are programming for an actual bank, please do not use this code

Error Handling

Handling Errors

```
var alice = BankAccount(funds: 0)
var bob = BankAccount(funds: 200)

do {
    try alice.transfer(amount: 100, to: &bob)
} catch {
    print("Couldn't transfer money: \(error)")
}
```

Error Handling

Other Ways of Handling Errors

```
func somethingThatCouldFail() throws -> Int {  
    /* ... */  
}
```

Crashes on error `try!` somethingThatCouldFail()

Returns nil on error `try?` somethingThatCouldFail()

Review

- Declaring constants and variables is easy with **type inference**
- **Optionals** make handling nil/null much safer
- **Functions and closures** are first-class types
- **Classes, structs, and enums** let us organize code and data
- **Protocols and extensions** make defining and standardizing behavior easy

This is barely scratching the surface!

Protocol extensions	Sequence and collection methods	Generic types	
Hash sets	Type casting	Associated types	guard and defer
Option sets	Result builders	Unsafe pointers	Observation
Mirrors	Key path expressions	Dynamic member lookup	Macros
C/C++/Objective-C interop	Generic constraints	Regex support	

And much more!

Check out the resources at <https://www.seas.upenn.edu/~cis1951/resources/>

Homework 1

Text Adventure Game

- Make an adventure game using Swift language constructs
- We'll provide the UI, you bring the gameplay
- Will be released **Monday, 1/29**
- Due after 2 weeks on **Monday, 2/12**
- Be creative!

Mystery of the Enchanted Forest

[Reset](#)

You are deep within the forest. It's eerily quiet here. There's a path leading south.

You notice something shiny beneath the foliage. It looks like a sword!

look

You are deep within the forest. It's eerily quiet here. There's a path leading south.

You notice something shiny beneath the foliage. It looks like a sword!

take sword

You take the Magic Sword! Your power increases.

look

You are deep within the forest. It's eerily quiet here. There's a path leading south.

fight dragon

There is no dragon here to fight.

south

You are at the entrance of the forest. Paths lead north and east.

east

This is the dragon's lair. The air is thick with the smell of sulfur.

The Dragon of the Enchanted Forest is here, guarding its treasure!

fight dragon

With the Magic Sword in your hand, you slay the Dragon of the Enchanted Forest!

Congratulations! You have completed your quest.

