iOS Lecture 3 Agenda

Slides

- Optionals (3.1)
  - History of NULL as a value [The Rise of “Worse is Better”](#)

    [The Billion-Dollar Mistake](#)
- Type Casting (3.2)
  - History of Object-oriented programming and message-passing
- Guard (3.3)
  - [Railway-oriented programming](#)
- Scope (3.4)
- Enumerations (3.5)
- [Error Handling](#)

Demonstrations

- Exit Status
- GitHub and Travis procedure

- Program Design and Code Style marking advice
  - Main principle: how easy is it to read, edit, and maintain your code?
  - **Functional separation**: Is the problem broken down into meaningful parts?
  - **Loose coupling**: Can parts be changed in isolation of each other?
  - **Extensibility**: Would it be easy to add more functionality? (more operations, more numerical accuracy, interactivity, variables, etc)
  - **Control flow**: Are all actions of the same type handled at the same level?
  - **Error handling**: Are errors detected at appropriate places? Can they be collected somewhere central?
  - **Marker's Discretion**: Anything you can do to help a project run smoothly.

Quiz 1 Review

- Constants and Variables
- Values and References
  - See Lecture 2 slides 76-77

Unit 3—Lesson 1: Optionals

Say

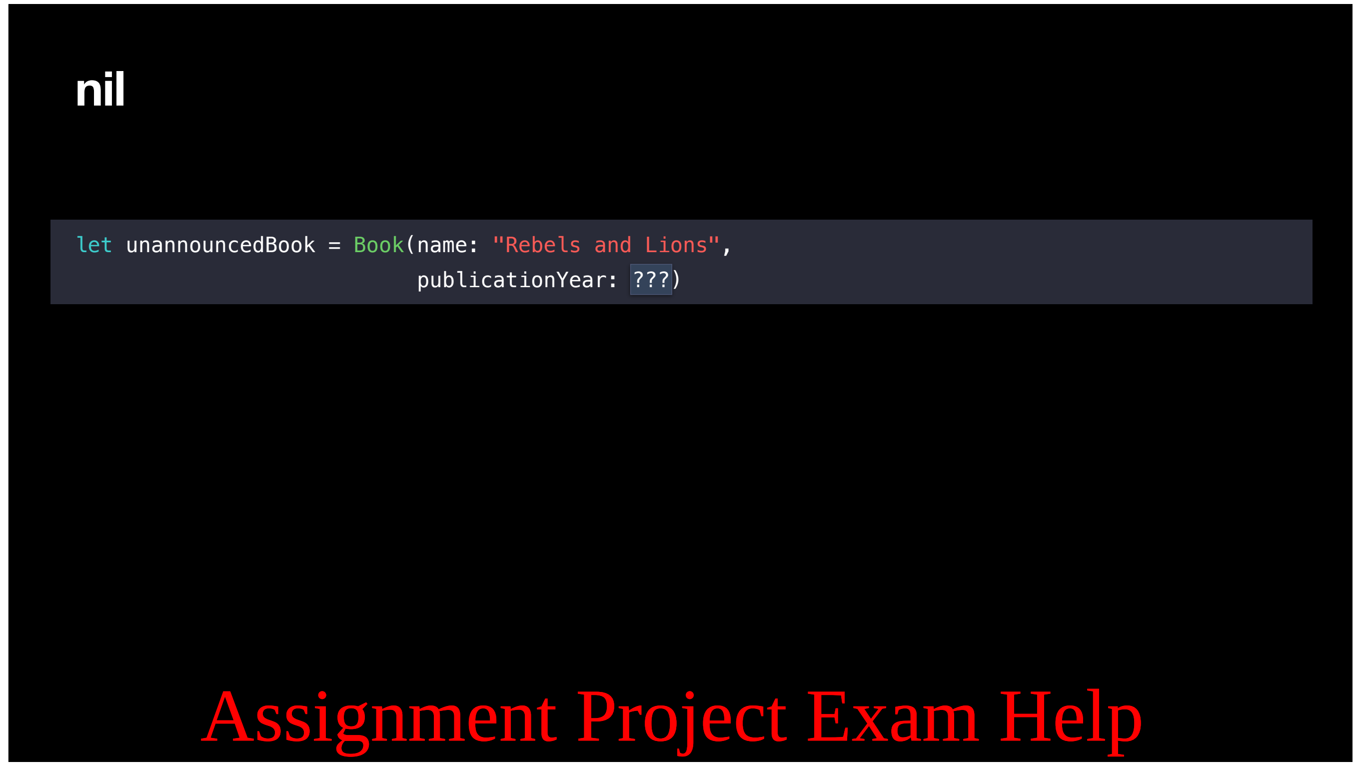
- Remember that safety is one of the big goals of Swift.
- Bugs are commonly created when you're referencing "not having" something.
- Swift has an interesting solution.

nil

```swift
struct Book {
  let name: String
  let publicationYear: Int
}

let firstHarryPotter = Book(name: "Harry Potter and the Sorcerer's Stone",
                            publicationYear: 1997)
let secondHarryPotter = Book(name: "Harry Potter and the Chamber of Secrets",
                             publicationYear: 1998)

let books = [firstHarryPotter, secondHarryPotter]
```

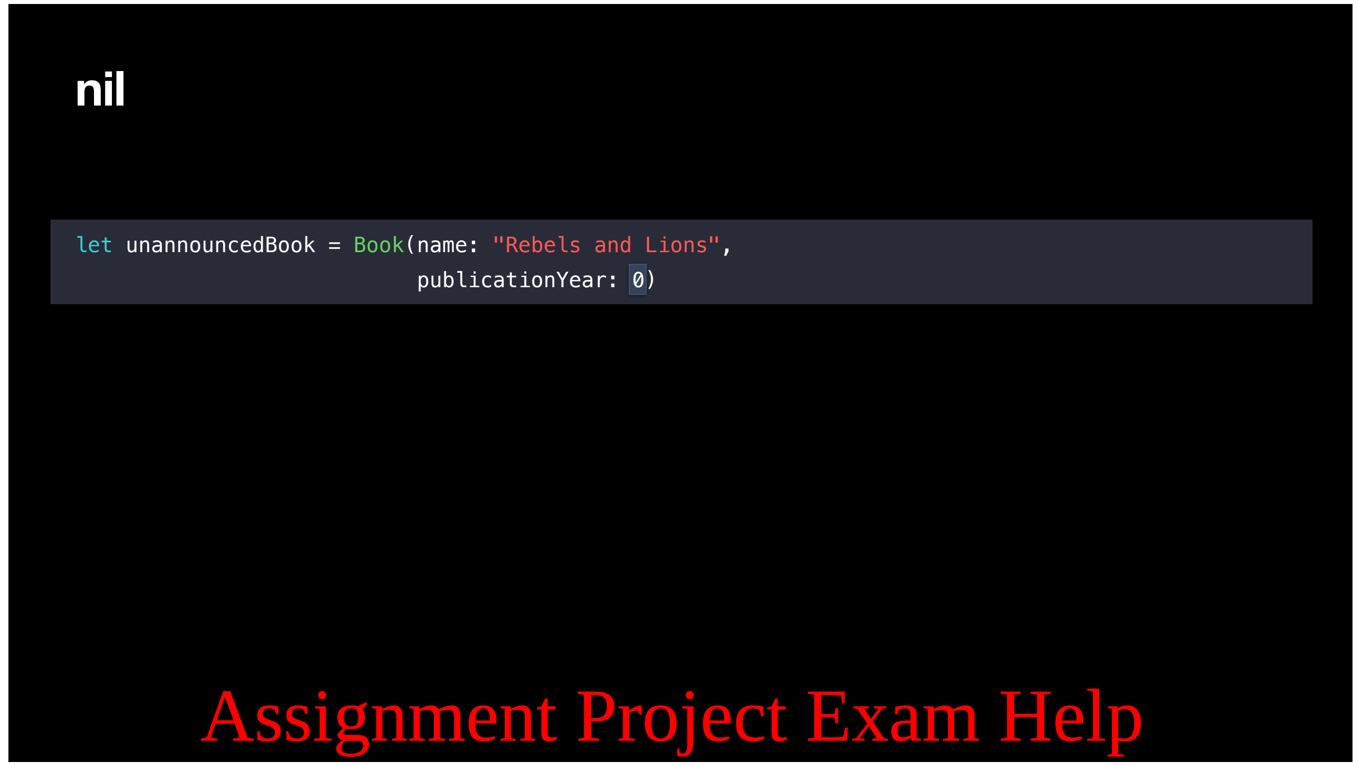

Say

- This slide shows a Book struct with name and publicationYear properties.



Say

• What if we want to represent a book that hasn't been published yet?



Say

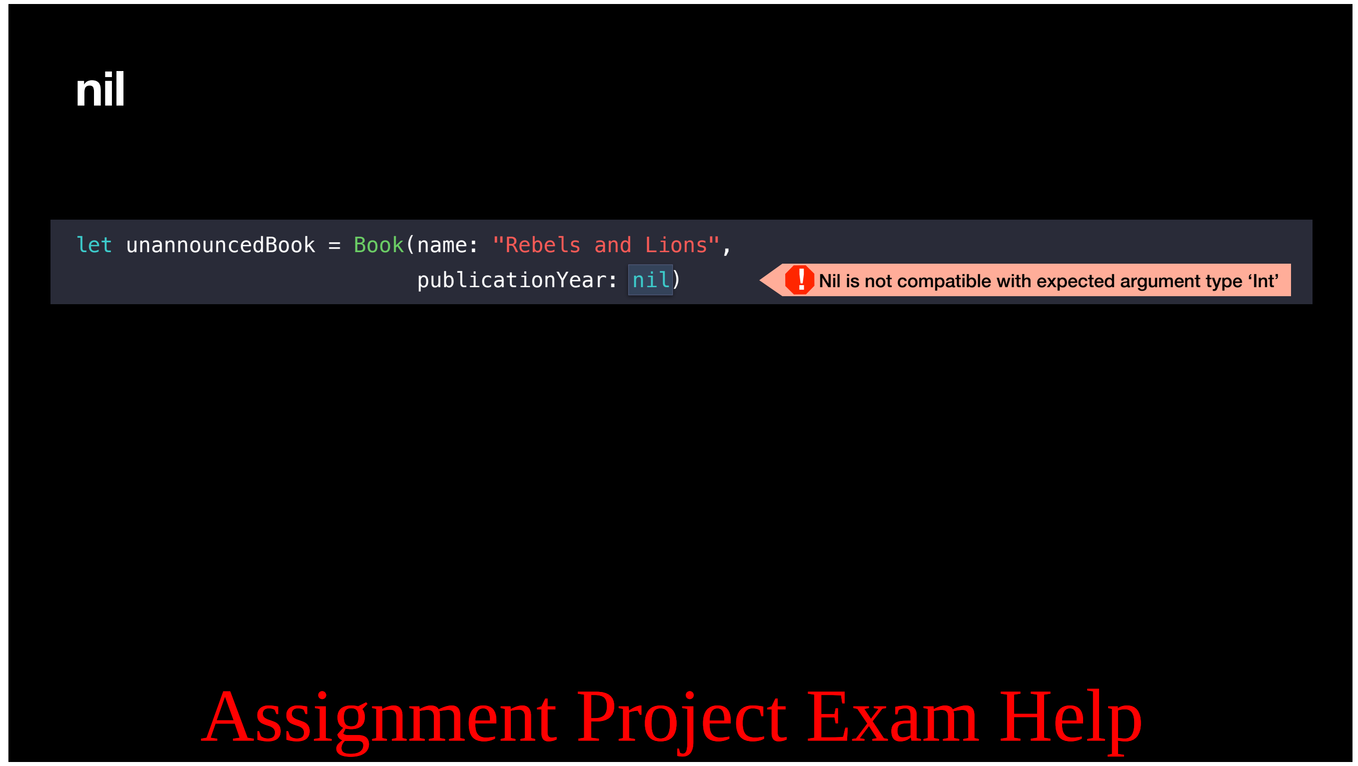
• Zero isn't accurate, because that would mean the book is over 2,000 years old.

nil

```swift
let unannouncedBook = Book(name: "Rebels and Lions",
                           publicationYear: 2017)
```


Say

- The current year isn't accurate either, because it may be released the next year. There's no known launch date.



Say

- *nil* represents the absence of a value, or nothing. Because there is no publicationYear value yet, publicationYear should be nil.
- That looks better, but the compiler throws an error.

Do

- Click to display the error.

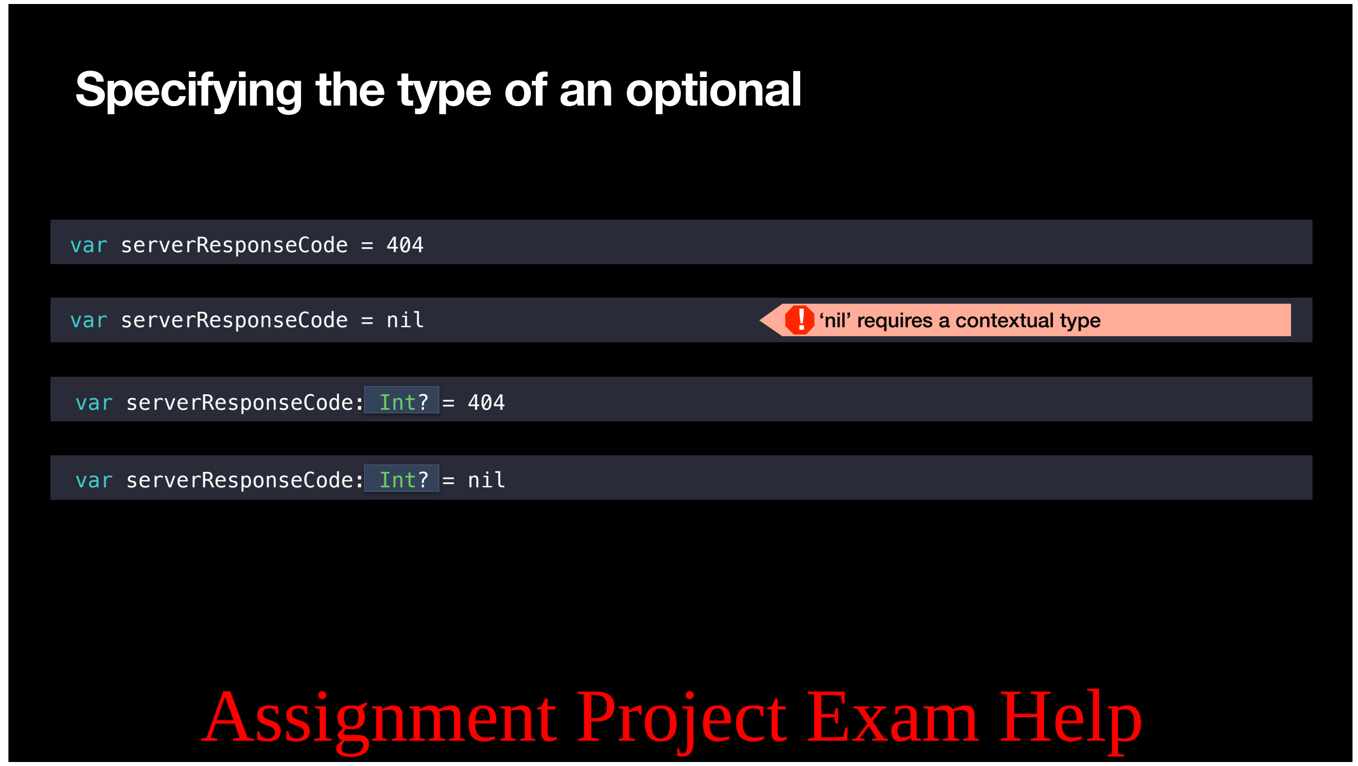
Say

- All instance properties must be set during initialization, and you can't pass nil to the `publicationYear` parameter because it expects an Int value.

```swift
struct Book {
  let name: String
  let publicationYear: Int?
}

let firstHarryPotter = Book(name: "Harry Potter and the Sorcerer's Stone",
                            publicationYear: 1997)
let secondHarryPotter = Book(name: "Harry Potter and the Chamber of Secrets",
                             publicationYear: 1998)

let books = [firstHarryPotter, secondHarryPotter]

let unannouncedBook = Book(name: "Rebels and Lions", publicationYear: nil)
```


Say

- This slide shows let publicationYear: Int? and shows that nil will now work.
- publicationYear may have a value or may be nil, which means "no value," not zero.
- Anything can be optional—string, struct, and so on.



Do and Say

- Click to display the first line. This defines a regular Int.
- Click to display the second line. This generates an error.
- Click to display the error. What type is it supposed to be? Int optional, String Optional, or other?
- Click to display the two ways to define an optional.

    var serverResponseCode: Int? = 404

- This means it's 404 now, but it might be no value at some point.

    var serverResponseCode: Int? = nil

- This means no value now, but it might be an Int at some point.
- Note that you can do var someInt: Int? and Swift makes it nil for you.

Working with optional values

Force-unwrap

```
if publicationYear != nil {
  let actualYear = publicationYear!
  print(actualYear)
}

let unwrappedYear = publicationYear!
```

error: Execution was interrupted

Say

- We're going to see several ways to safely deal with optionals.
- One way to test an optional is to compare it to nil.
- Describe "force unwrap":

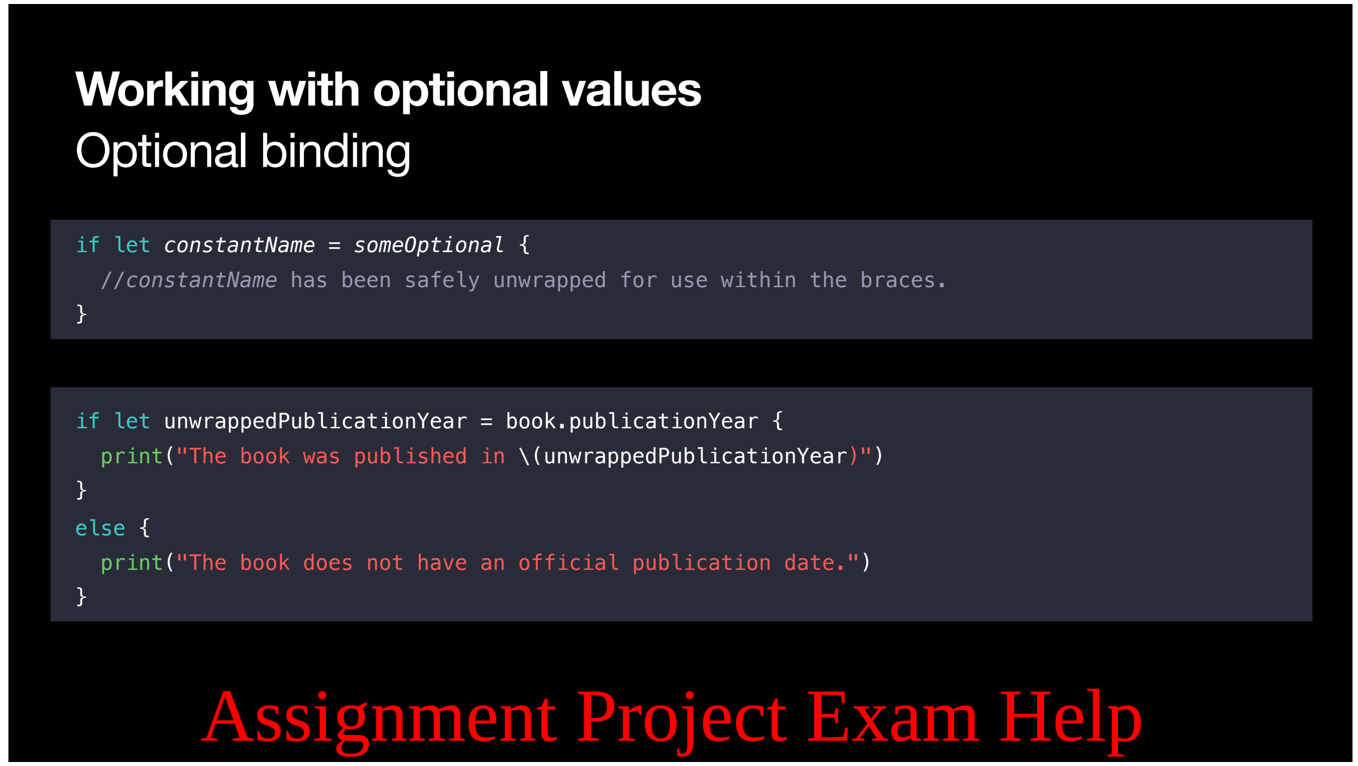
    "?" means variable may be nil. "!" means the app may crash.

Do

- Click to display the error.

Say

- If you use force unwrap and the value is nil, your app crashes.
- There are very few use cases for this—we’ll see one in iOS.



Say

- Because it's so common to need to work with optionals, Swift has some shortcuts that are simpler than an explicit comparison to nil.
- The first code example shows "if let" syntax: "optional binding."
- Note that "if var" also works.
- The second code example shows an "if let" example using the optional "publicationYear" property on Book from an earlier slide.

Functions and optionals

Return values

```
let string = "123"
let possibleNumber = Int(string)
```

```
let string = "Cynthia"
let possibleNumber = Int(string)
```


Say

- Int has an init that takes a String parameter. What should it do if the input doesn't make sense?
- This is a great use of optionals—the return type is Int? ("optional int")

Functions and optionals

Defining

```
func printFullName(firstName: String, middleName: String?, lastName: String)
```

```
func textFromURL(url: URL) -> String?
```


Say

- The first code example shows how to declare a function with an optional argument.
- The second code example shows how to declare a function that returns an optional.

Failable initializers

```swift
struct Toddler {
  var birthName: String
  var monthsOld: Int
}
```


Say

- We saw that init(string) can return nil. That's called a failable initializer—one that can return nil.
- Here's a "Toddler" class with an "age" property.

Failable initializers

```swift
struct Toddler {
  var birthName: String
  var monthsOld: Int

  init?(birthName: String, monthsOld: Int) {
    if monthsOld < 12 || monthsOld > 36 {
      return nil
    } else {
      self.birthName = birthName
      self.monthsOld = monthsOld
    }
  }
}
```


Say

- Let's say we want `Toddler` to return nil if someone tries to make an instance that's too old or too young:

  init?(name:String, monthsOld:Int)

- Note that the syntax is a bit odd here. You return nil, but you don't return the success instance.

Failable initializers

```
let possibleToddler = Toddler(birthName: "Joanna", monthsOld: 14)
if let toddler = possibleToddler {
  print("\(toddler.birthName) is \(toddler.monthsOld) months old")
} else {
  print("The age you specified for the toddler is not between 1 and 3 yrs of age")
}
```


Say

- The return type will be `Toddler?` so you need to unwrap with if/let or something.

Optional chaining

```
class Person {
  var age: Int
  var residence: Residence?
}

class Residence {
  var address: Address?
}

class Address {
  var buildingNumber: String?
  var streetName: String?
  var apartmentNumber: String?
}
```


Say

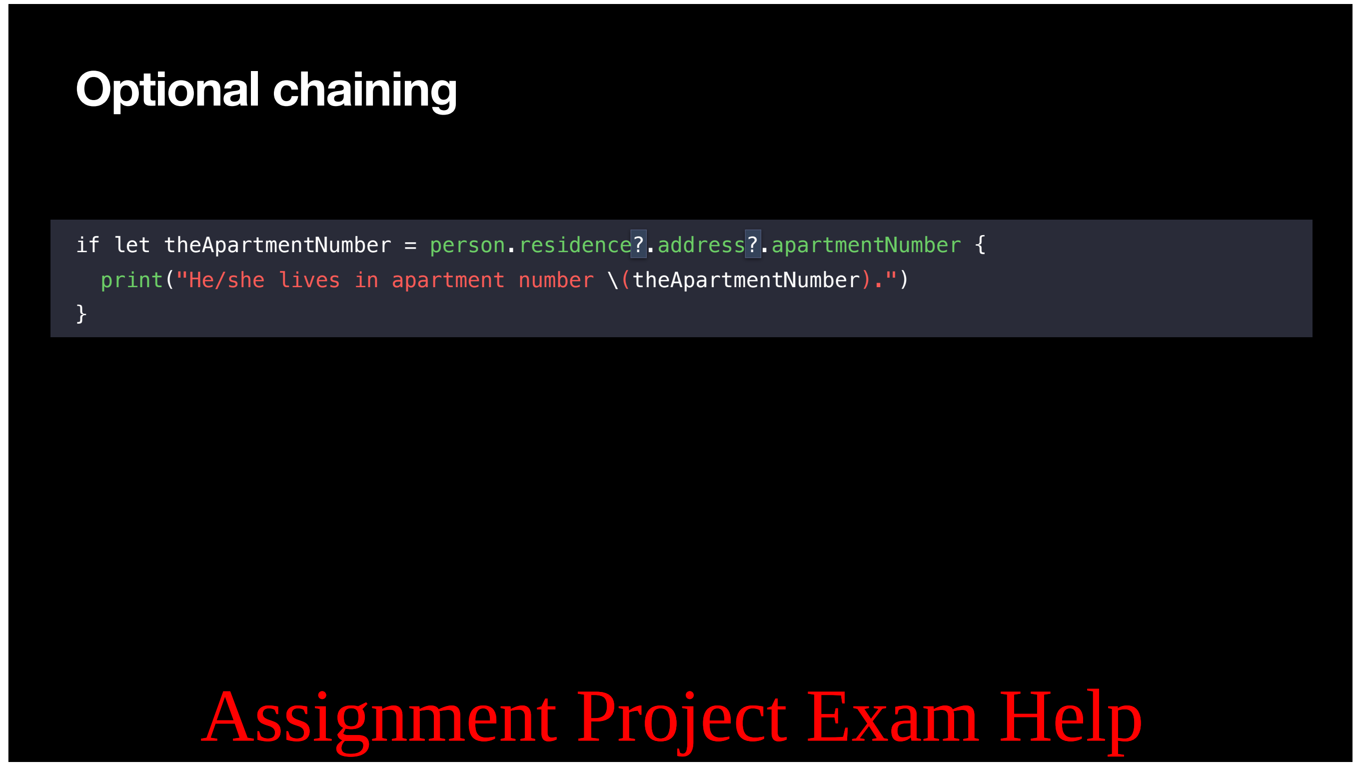
- The setup shows three classes with optional properties.
- Note that a Person can have a Residence, which can have an Address (Person > Residence > Address).

Optional chaining

```
if let theResidence = person.residence {
  if let theAddress = theResidence.address {
    if let theApartmentNumber = theAddress.apartmentNumber {
      print("He/she lives in apartment number \(theApartmentNumber).")
    }
  }
}
```


Say

- We can do more checking with if/let but we get the "pyramid of doom."



Say

- The optional chaining example: The if/let will short-circuit if it encounters nil.

Do

- Demo taking it one farther—add an Int():

  if let apartmentNumber = person.residence?.address?.Int(apartmentNumber)

Implicitly Unwrapped Optionals

```
class ViewController: UIViewController {
  @IBOutlet weak var label: UILabel!
}
```

Unwraps automatically

Should only be used when need to initialize an object without supplying the value and you'll be giving the object a value soon afterwards

Unit 3, Lesson 1

Lab: Optionals.playground

Open and complete the exercises in `Lab - Optionals.playground`

Unit 3—Lesson 2:
Type Casting and Inspection

Type inspection

```swift
func getClientPet() -> Animal {
  //returns the pet
}

let pet = getClientPet() //`pet` is of type `Animal`
```


Say

- If a function returns a class that has subclasses, we may need to know how to ask which subclass it is:

  func getClientPet() -> Animal

Type inspection

```
if pet is Dog {
  print("The client's pet is a dog")
} else if pet is Cat {
  print("The client's pet is a cat")
} else if pet is Bird {
  print("The client's pet is a bird")
} else {
  print("The client has a very exotic pet")
}
```


Note
- Shows the "is" syntax:
  if pet is Dog {}

Type inspection

```
let pets = allPets() //`pets` is of type `[Animal]`
var dogCount = 0, catCount = 0, birdCount = 0
for pet in pets {
  if pet is Dog {
    dogCount += 1
  } else if pet is Cat {
    catCount += 1
  } else if pet is Bird {
    birdCount += 1
  }
}
print("Brad looks after \(dogCount) dogs, \(catCount) cats, and \(birdCount) birds.")
```


Say
• Here’s another example.

Type casting

```swift
func walk(dog: Dog) {
  print("Walking \(dog.name)")
}

func cleanLitterBox(cat: Cat) {. . .}

func cleanCage(bird: Bird) {. . .}

for pet in pets {
  if pet is Dog {
    walk(dog: pet) // Compiler error. The compiler sees `pet` as an `Animal`, not a `Dog`.
  }
  . . .
}
```


Say
- Sometimes we have a type that's too vague, and we need to tell or ask the compiler that we know—or want to know—if it can be more precise.
- If we have a function that takes "Dog" and we have an instance that's "Animal," the compiler complains.
- So we need to:
  - Ask if it's a Dog
  - Tell the compiler that we know it's a Dog

Type casting

```
for pet in pets {
  if let dog = pet as? Dog {
    walk(dog: dog)
  } else if let cat = pet as? Cat {
    cleanLitterBox(cat: cat)
  } else if let bird = pet as? Bird {
    cleanCage(bird: bird)
  }
}
```


Say

- if let theDog = pet as? Dog {}
- If "pet" is of type "Dog," then the code runs.
- Remember, Apple says that "?" may be nil and "!" may crash.

Any

```swift
var items: [Any] = [5, "Bill", 6.7, Dog()]
```


Say

- Although Swift is strongly typed, you can specify a nonspecific type, when appropriate.
- "Any" is any type.
- .AnyObject is any class.
- If a collection is "Any" or "AnyObject," you may need to do introspection.

Any

```
var items: [Any] = [5, "Bill", 6.7, Dog()]
let firstItem = items[0]

if firstItem is Int {
  print("The first element is an integer")
} else if firstItem is String {
  print("The first element is a string")
} else {
  print("The first element is neither an integer nor a string")
}
```


Note
• Shows "is" introspection.

Any

```swift
var items: [Any] = [5, "Bill", 6.7, Dog()]

if let firstItem = items[0] as? Int {
  print(firstItem + 4)
}
```

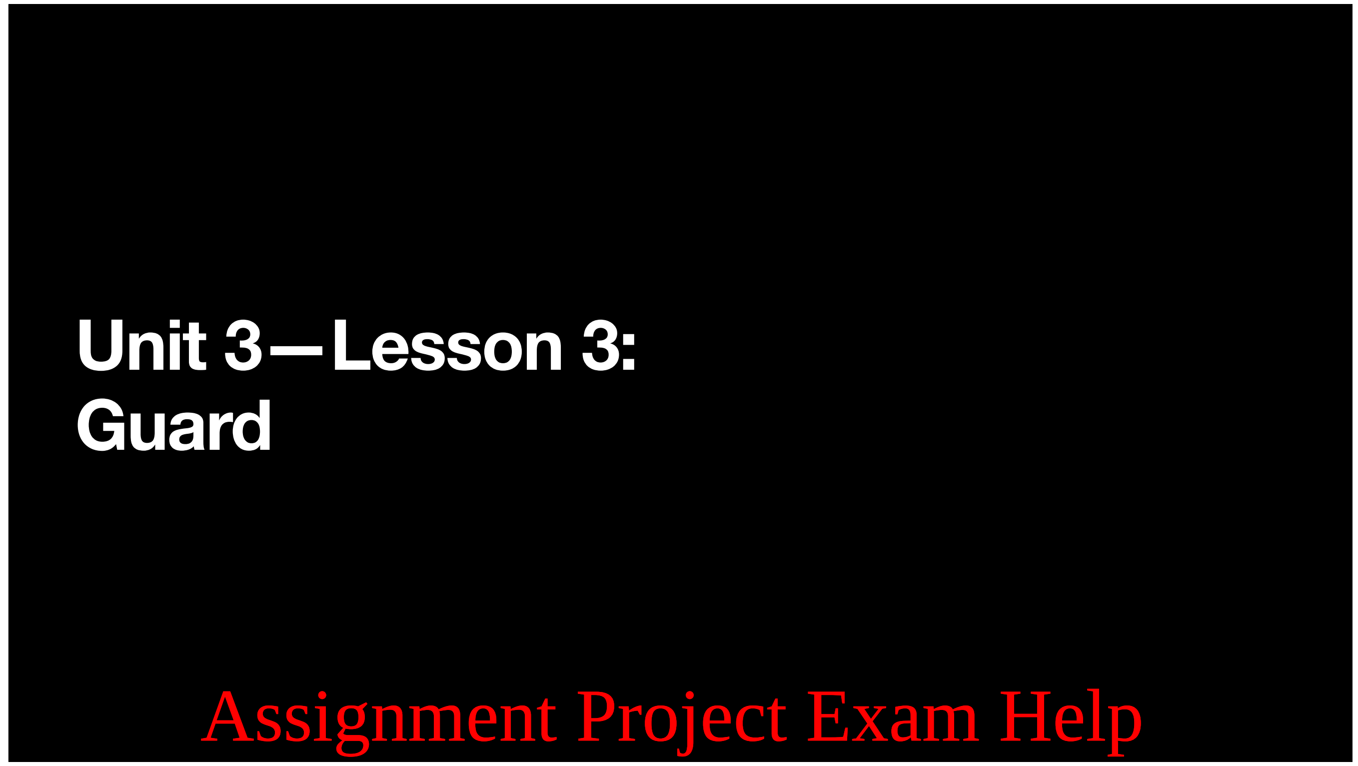
```
9
```


Say

- As we saw with optionals, if/let also works with "as?".
- Being type-specific is better when possible—the compiler can help you avoid errors—but "Any" and "AnyObject" are OK when necessary.

Unit 3—Lesson 2
Lab: Type Casting and Inspection

Open and complete the exercises in `Lab - Type Casting.playground`



Say

- Recall the earlier discussion about how nested "if" statements can become difficult to parse.
- Swift includes "guard" as another way to make code more readable.

```
func singHappyBirthday() {
  if birthdayIsToday {
    if invitedGuests > 0 {
      if cakeCandlesLit {
        print("Happy Birthday to you!")
      } else {
        print("The cake candle's haven't been lit.")
      }
    } else {
      print("It's just a family party.")
    }
  } else {
    print("No one has a birthday today.")
  }
}
```


Say

- When code has many if statements, the important code is indented really far in.
- Also, the "else" clauses end up far away from the "if." It can be very hard to read.

```
func singHappyBirthday() {
  guard birthdayIsToday else {
    print("No one has a birthday today.")
    return
  }

  guard invitedGuests > 0 else {
    print("It's just a family party.")
    return
  }

  guard cakeCandlesLit else {
    print("The cake's candles haven't been lit.")
    return
  }

  print("Happy Birthday to you!")
}
```


Say
- "guard" is a short circuit—usually to return.
- The "all conditions ok" code ends up not indented.

guard

```
guard condition else {
  //false: execute some code
}

//true: execute some code
```


Say

- Like an if/else statement, the "else" clause runs if the condition is false

guard

```swift
func divide(_ number: Double, by divisor: Double) {
  if divisor != 0.0 {
    let result = number / divisor
    print(result)
  }
}
```

```swift
func divide(_ number: Double, by divisor: Double) {
  guard divisor != 0.0 else { return }

  let result = number / divisor
  print(result)
}
```


Say

- This slide compares using "if" to using "guard."

guard with optionals

```swift
if let eggs = goose.eggs {
  print("The goose laid \(eggs.count) eggs.")
}
//`eggs` is not accessible here
```

```swift
guard let eggs = goose.eggs else { return }
//`eggs` is accessible hereafter
print("The goose laid \(eggs.count) eggs.")
```


Say

- Here's another example of the "good" path being indented. "guard" can also help with unwrapping optionals.
- Note that the optional-bound value "eggs" is only in scope inside the if/let, but the "guard" one is available everywhere

guard with optionals

```
func processBook(title: String?, price: Double?, pages: Int?) {
  if let theTitle = title, let thePrice = price, let thePages = pages {
    print("\(theTitle) costs $\(thePrice) and has \(thePages) pages.")
  }
}
```

```
func processBook(title: String?, price: Double?, pages: Int?) {
  guard let theTitle = title, let thePrice = price, let thePages = pages else { return }
  print("\(theTitle) costs $\(thePrice) and has \(thePages) pages.")
}
```


Say
- When you use "guard let": If the guard succeeds, the "let" variable is in scope after the guard.
- Just like if/let can unwrap multiple optionals, "guard" can too:
  guard let theTitle = title, let thePrice = price else { return }
- This is a simple example: Try to imagine several things happening after the "guard" but not indented.

Unit 3—Lesson 3

Lab: Guard

Open and complete the exercises in `Lab - Guard.playground`

Unit 3—Lesson 4:
Constant and Variable Scope

Scope

Global scope—Defined outside of a function

Local scope—Defined within braces ({ })

```swift
var globalVariable = true

if globalVariable {
 let localVariable = 7
}
```


Note

• The example shows global and local variables.

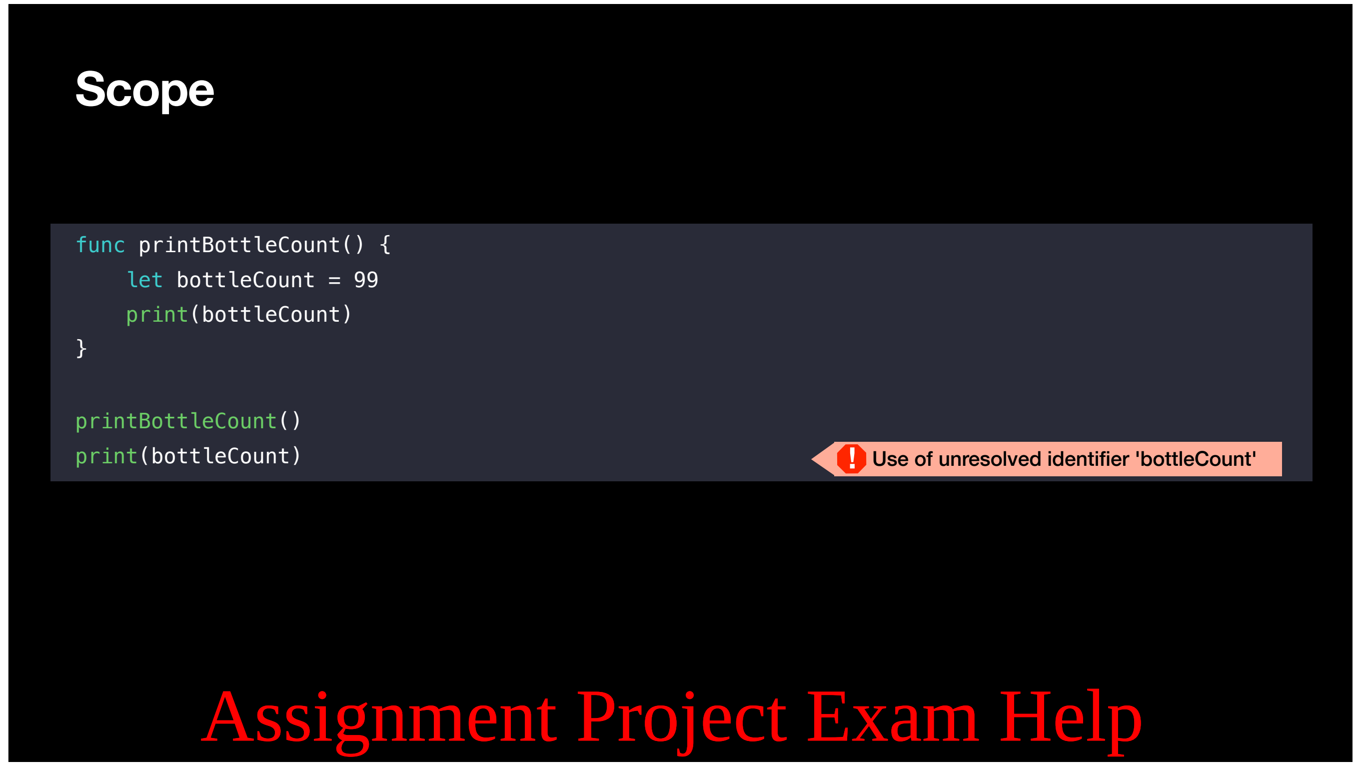
Scope

```
var age = 55

func printMyAge() {
  print("My age: \(age)")
}

print(age)
printMyAge()
```

```
55
My age: 55
```


Note

• Shows "age" as a global variable, available everywhere.



Note

- Shows "bottleCount" as a local variable, not available outside of the function.

Scope

```
func printTenNames() {
  var name = "Richard"
  for index in 1...10 {
    print("\(index): \(name)")
  }
  print(index)
  print(name)
}

printTenNames()
```

Use of unresolved identifier 'index'

Say
- "name" is OK because it's at the top level of the function.
- "index" is more tightly defined. It's part of the for/in loop, so it's available only inside the for/in loop.
- "name" is also available inside the for/in loop.

Variable shadowing

```
let points = 100

for index in 1...3 {
  let points = 200
  print("Loop \(index): \(points+index)")
}
print(points)
```

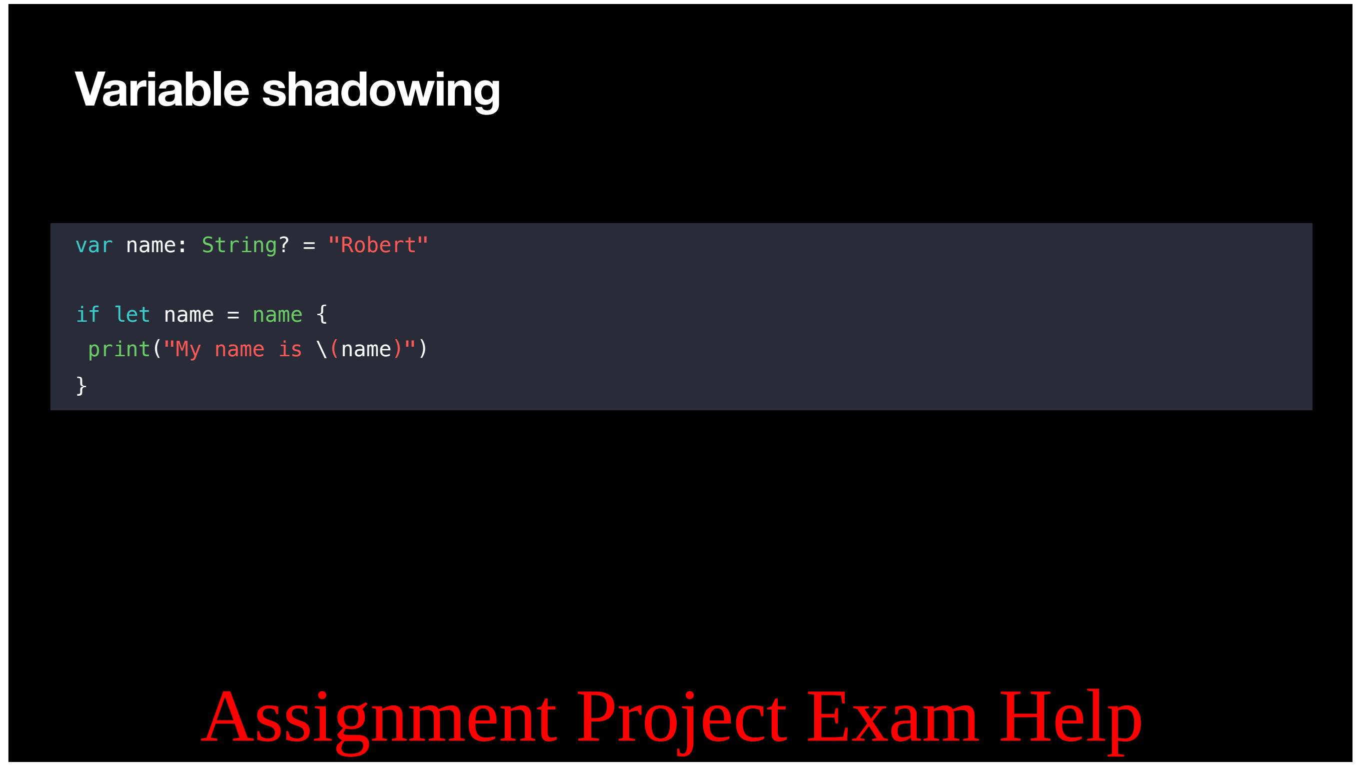
```
Loop 1: 201
Loop 2: 202
Loop 3: 203
100
```


Do

- Walk through the example. It prints 200, then 100.

Say

- Be careful when you use variable shadowing. It shouldn't be too hard to parse, but is it worth the extra effort compared to just using different variable names?



Say

- This is somewhat common to do:
  if let name = name
- Using this is a stylistic call. Not everyone agrees about using this format.

Variable shadowing

```swift
func exclaim(name: String?) {
  if let name = name {
    print("Exclaim function was passed: \(name)")
  }
}
```

```swift
func exclaim(name: String?) {
  guard let name = name else { return }
  print("Exclaim function was passed: \(name)")
}
```


Say

- "guard" has the same caveats as the if/let example.

Shadowing and initializers

```
struct Person {
  var name: String
  var age: Int
}

let todd = Person(name: "Todd", age: 50)
print(todd.name)
print(todd.age)
```

```
Todd
50
```


Say

- The memberwise initializer of a Struct uses argument names that are the same as the property names.
- That makes for readable code, because the argument names are also used as property accessors.

Shadowing and initializers

```swift
struct Person {
  var name: String
  var age: Int

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


Say
- Here's what an initializer looks like.

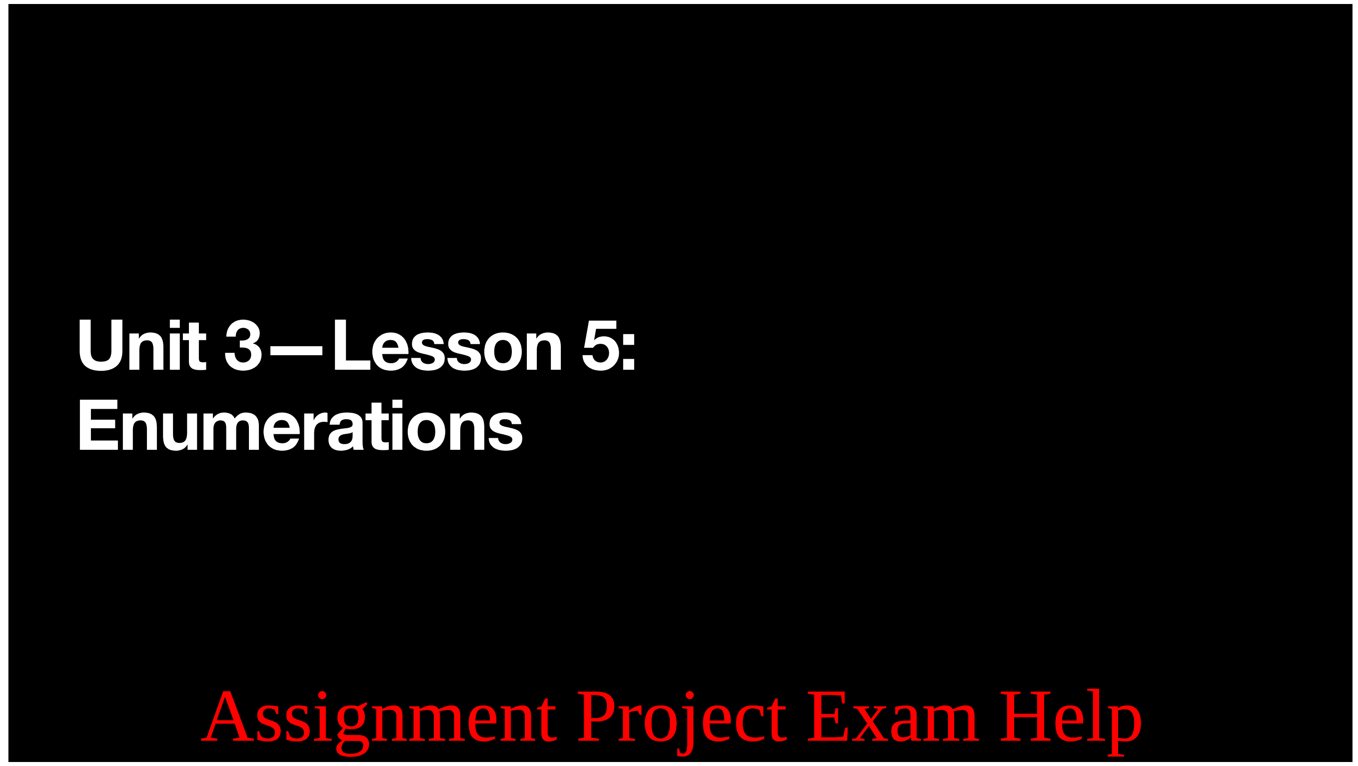
Do
- Click to highlight "self" in the init method.

Say
- Why do we need "self" here? Answer: Shadowing—we need to be precise to satisfy the compiler.

Unit 3—Lesson 4

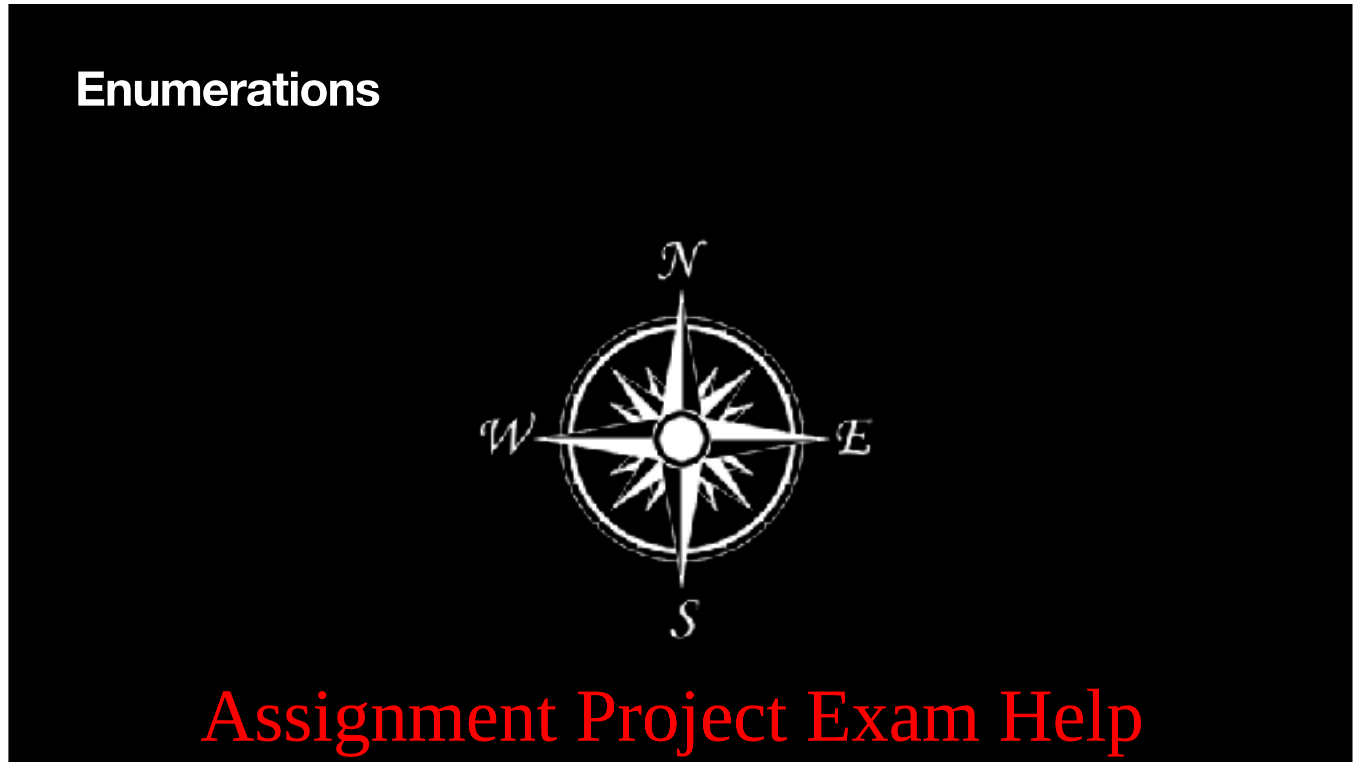
Lab: Constant and Variable Scope

Open and complete the exercises in `Lab - Scope.playground`



Say

- Enumerations in Swift are similar to things in other languages, but they’re also much more powerful than in languages like C.



Say

- In the simplest form, "enum" lets us group related things into a Type, such as the points of a compass (north, east, south, west).

Enumerations

```swift
enum CompassPoint {
  case north
  case east
  case south
  case west
}
```

Do
• Click to highlight "enum."
• Click again to highlight "case."
Say
• The type name is "CompassPoint."
• An instance of CompassPoint can be any one of the case values.
• Follow the naming conventions : uppercase Type name, lowercase case items.

Enumerations

```swift
enum CompassPoint {
  case north, east, south, west
}
```

```swift
var compassHeading = CompassPoint.west
```

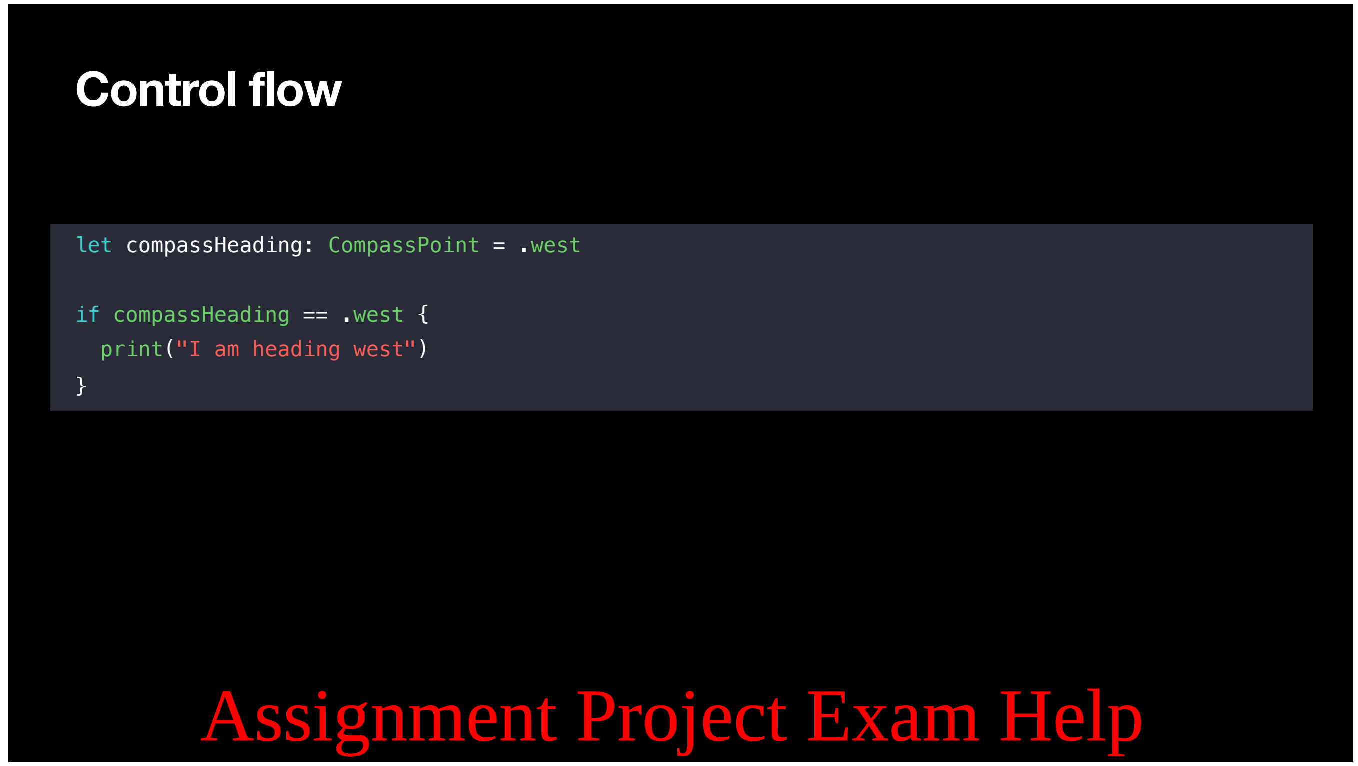
```swift
var compassHeading: CompassPoint = .west
compassHeading = .north
```


Do
• Click to display using the CompassPoint enum.
Say
• Note that you can have a comma-separated list of items in a single case.
Do
• Click to display one style of declaring an enum variable.
Say
• As with everything else, if the Swift compiler can't unambiguously infer the type, we have to help.
Do
• Click to display another way to declare an enum variable.
• Click to display changing the value with an enum.

Control flow

```swift
let compassHeading: CompassPoint = .west

switch compassHeading {
  case .north:
    print("I am heading north")
  case .east:
    print("I am heading east.")
  case .south:
    print("I am heading south")
  case .west:
    print("I am heading west")
}
```


Say
- You can switch on enum cases.



Say

- You can use "==" for enum cases.

Type safety benefits

```swift
struct Movie {
  var name: String
  var releaseYear: Int
  var genre: String
}

let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: "Aminated")
```


Say

- Can you identify the mistake?

Do

- Click to highlight the typo.

Type safety benefits

```swift
enum Genre {
  case animated, action, romance, documentary, biography, thriller
}

struct Movie {
  var name: String
  var releaseYear: Int
  var genre: Genre
}

let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: .animated)
```

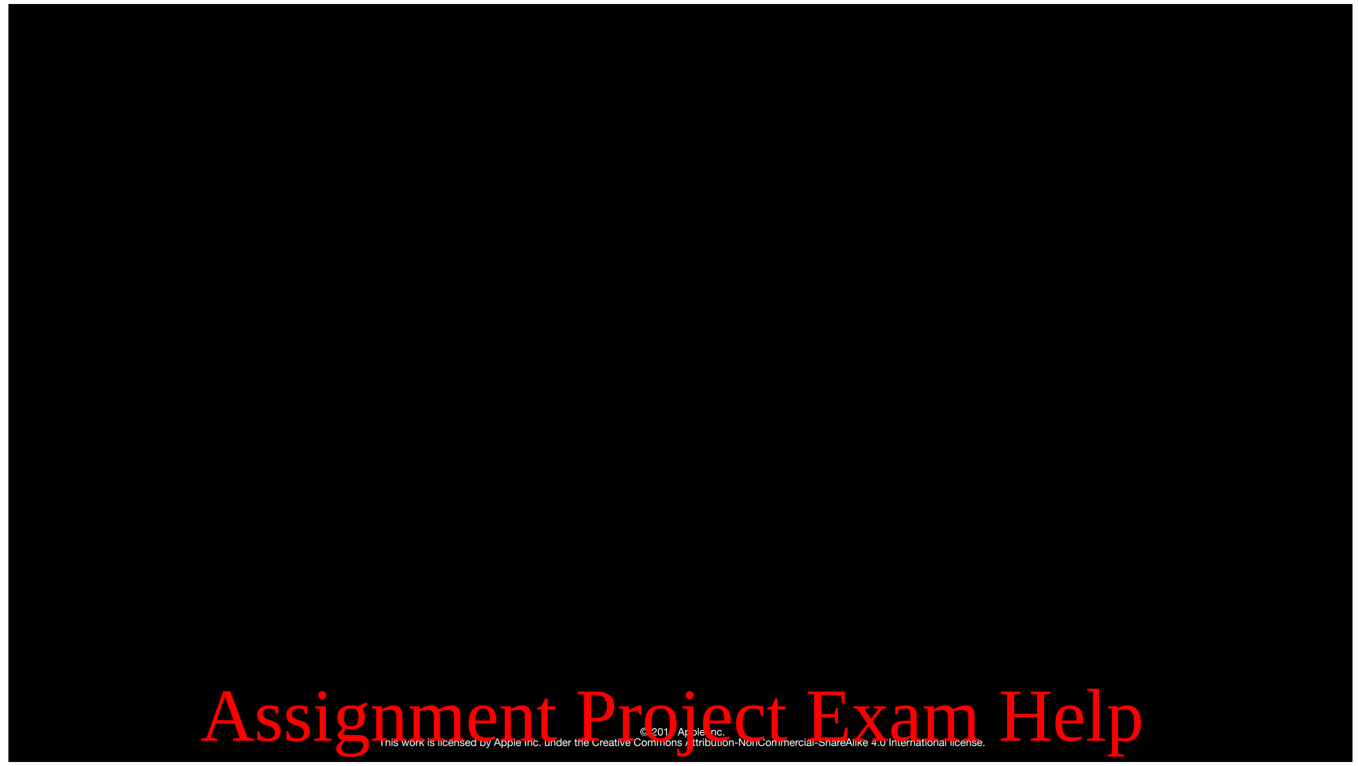

Say

- Because we know at coding time what all the genres are, we can create an enum to specify them.
- Then when we pass in ".animated", the compiler will catch typographical errors.

Unit 3—Lesson 5

Lab: Enumerations

Open and complete the exercises in `Lab - Enumerations.playground`


This work is licensed by Apple Inc. under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license.

Error Handling

On This Page

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.

Some operations aren't guaranteed to always complete execution or produce a useful output. Optionals are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure, so that your code can respond accordingly.

As an example, consider the task of reading and processing data from a file on disk. There are a number of ways this task can fail, including the file not existing at the specified path, the file not having read permissions, or the file not being encoded in a compatible format. Distinguishing among these different situations allows a program to resolve some errors and to communicate to the user any errors it can't resolve.

> NOTE
>
> Error handling in Swift interoperates with error handling patterns that use the `NSError` class in Cocoa and Objective-C. For more information about this class, see Error Handling in *Using Swift with Cocoa and Objective-C (Swift 4.1)*.

Representing and Throwing Errors

In Swift, errors are represented by values of types that conform to the `Error` protocol. This empty protocol indicates that a type can be used for error handling.

Swift enumerations are particularly well suited to modeling a group of related error conditions, with associated values allowing for additional information about the nature of an error to be communicated. For example, here's how you might represent the error conditions of operating a vending machine inside a game:


```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```


Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a `throw` statement to throw an error. For example, the following code throws an error to indicate that five additional coins are needed by the vending machine:

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

Handling Errors

When an error is thrown, some surrounding piece of code must be responsible for handling the error—for example, by correcting the problem, trying an alternative approach, or informing the user of the failure.

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a `do`-`catch` statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

When a function throws an error, it changes the flow of your program, so it's important that you can quickly identify places in your code that can throw errors. To identify these places in your code, write the `try` keyword —or the `try?` or `try!` variation—before a piece of code that calls a function, method, or initializer that can throw an error. These keywords are described in the sections below.

> NOTE
>
> Error handling in Swift resembles exception handling in other languages, with the use of the `try`, `catch` and `throw` keywords. Unlike exception handling in many languages—including Objective-C—error handling in Swift does not involve unwinding the call stack, a process that can be computationally expensive. As such, the performance characteristics of a `throw` statement are comparable to those of a `return` statement.

Propagating Errors Using Throwing Functions

To indicate that a function, method, or initializer can throw an error, you write the `throws` keyword in the function's declaration after its parameters. A function marked with `throws` is called a *throwing function*. If the function specifies a return type, you write the `throws` keyword before the return arrow (->).

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

A throwing function propagates errors that are thrown inside of it to the scope from which it's called.

> NOTE
> Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

In the example below, the `VendingMachine` class has a `vend(itemNamed:)` method that throws an appropriate `VendingMachineError` if the requested item is not available, is out of stock, or has a cost that exceeds the current deposited amount:

```
struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}
```


The implementation of the `vend(itemNamed:)` method uses `guard` statements to exit the method early and throw appropriate errors if any of the requirements for purchasing a snack aren't met. Because a `throw` statement immediately transfers program control, an item will be vended only if all of these requirements are met.

Because the `vend(itemNamed:)` method propagates any errors it throws, any code that calls this method must either handle th

example, the `buyFavoriteSnack(person:vendingMachine:)` in the example below is also a throwing function, and any errors that the `vend(itemNamed:)` method throws will propagate up to the point where the `buyFavoriteSnack(person:vendingMachine:)` function is called.

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

In this example, the `buyFavoriteSnack(person: vendingMachine:)` function looks up a given person's favorite snack and tries to buy it for them by calling the `vend(itemNamed:)` method. Because the `vend(itemNamed:)` method can throw an error, it's called with the `try` keyword in front of it.

Throwing initializers can propagate errors in the same way as throwing functions. For example, the initializer for the `PurchasedSnack` structure in the listing below calls a throwing function as part of the initialization process, and it handles any errors that it encounters by propagating them to its caller.

```
struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}
```


Handling Errors Using Do-Catch

You use a `do-catch` statement to handle errors by running a block of code. If an error is thrown by the code in the `do` clause, it is matched against the `catch` clauses to determine which one of them can handle the error.

Here is the general form of a `do-catch` statement:

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
} catch {
    statements
}
```

You write a pattern after `catch` to indicate what errors that clause can handle. If a `catch` clause doesn't have a pattern, the clause matches any error and binds the error to a local constant named `error`. For more information about pattern matching, see Patterns.

For example, the following code matches against all three cases of the `VendingMachineError` enumeration.

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
```

```
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    pri
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

In the above example, the `buyFavoriteSnack(person:vendingMachine:)` function is called in a `try` expression, because it can throw an error. If an error is thrown, execution immediately transfers to the `catch` clauses, which decide whether to allow propagation to continue. If no pattern is matched, the error gets caught by the final `catch` clause and is bound to a local `error` constant. If no error is thrown, the remaining statements in the `do` statement are executed.

The `catch` clauses don't have to handle every possible error that the code in the `do` clause can throw. If none of the `catch` clauses handle the error, the error propagates to the surrounding scope. However, the error must be handled by *some* surrounding scope—either by an enclosing `do`-`catch` clause that handles the error or by being inside a throwing function. For example, the above example can be written so any error that isn't a `VendingMachineError` is instead caught by the calling function:

```
func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Invalid selection, out of stock, or not enough money.")
    }
}

do {
    try nourish(with: "Beet-Flavored Chips")
} catch {
    print("Unexpected non-vending-machine-related error: \(error)")
}
// Prints "Invalid selection, out of stock, or not enough money."
```


In the `nourish(with:)` function, if `vend(itemNamed:)` throws an error that's one of the cases of the `VendingMachineError` enumeration, `nourish(with:)` handles the error by printing a message. Otherwise, `nourish(with:)` propagates the error to its call site. The error is then caught by the general `catch` clause.

Converting Errors to Optional Values

You use `try?` to handle an error by converting it to an optional value. If an error is thrown while evaluating the `try?` expression, the value of the expression is `nil`. For example, in the following code `x` and `y` have the same value and behavior:

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

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

If `someThrowingFunction()` throws an error, the value of `x` and `y` is `nil`. Otherwise, the value of `x` and `y` is the value that the function returned. Note that `x` and `y` are an optional of whatever type `someThrowingFunction()` returns. Here the function returns an integer, so `x` and `y` are optional integers.

Using `try?` lets
example, the following code uses several approaches to fetch data, or returns `nil` if all of the approaches fail.

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk() { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

Disabling Error Propagation

Sometimes you know a throwing function or method won't, in fact, throw an error at runtime. On those occasions, you can write `try!` before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

For example, the following code uses a `loadImage(atPath:)` function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Specifying Cleanup Actions

You use a `defer` statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any necessary cleanup that should be performed regardless of *how* execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as `return` or `break`. For example, you can use a `defer` statement to ensure that file descriptors are closed and manually allocated memory is freed.

A `defer` statement defers execution until the current scope is exited. This statement consists of the `defer` keyword and the statements to be executed later. The deferred statements may not contain any code that would transfer control out of the statements, such as a `break` or a `return` statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they're written in your source code. That is, the code in the first `defer` statement executes last, the code in the second `defer` statement executes second to last, and so on. The last `defer` statement in source code order executes first.


```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

The above example uses a `defer` statement to ensure that the `open(_:)` function has a corresponding call to `close(_:)`.

> NOTE
>
> You can use a `defer` statement even when no error handling code is involved.