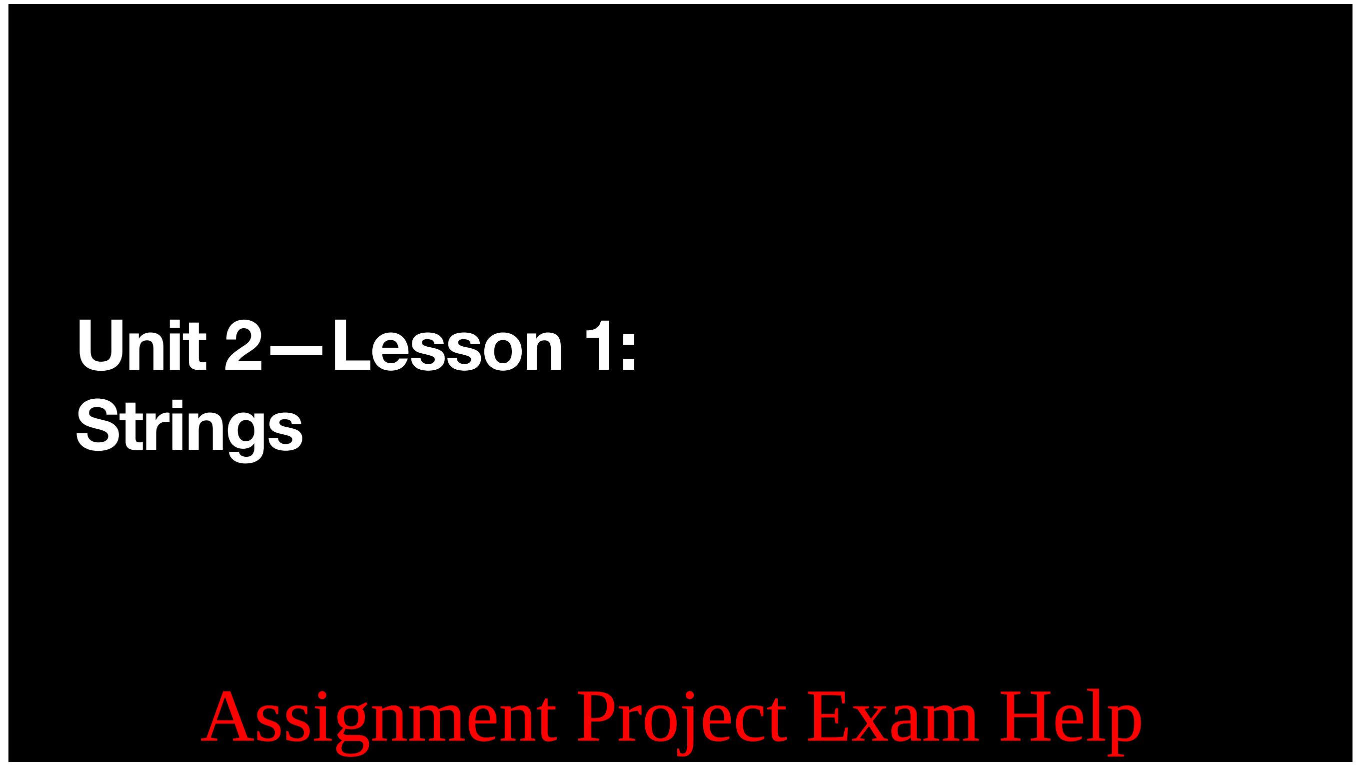
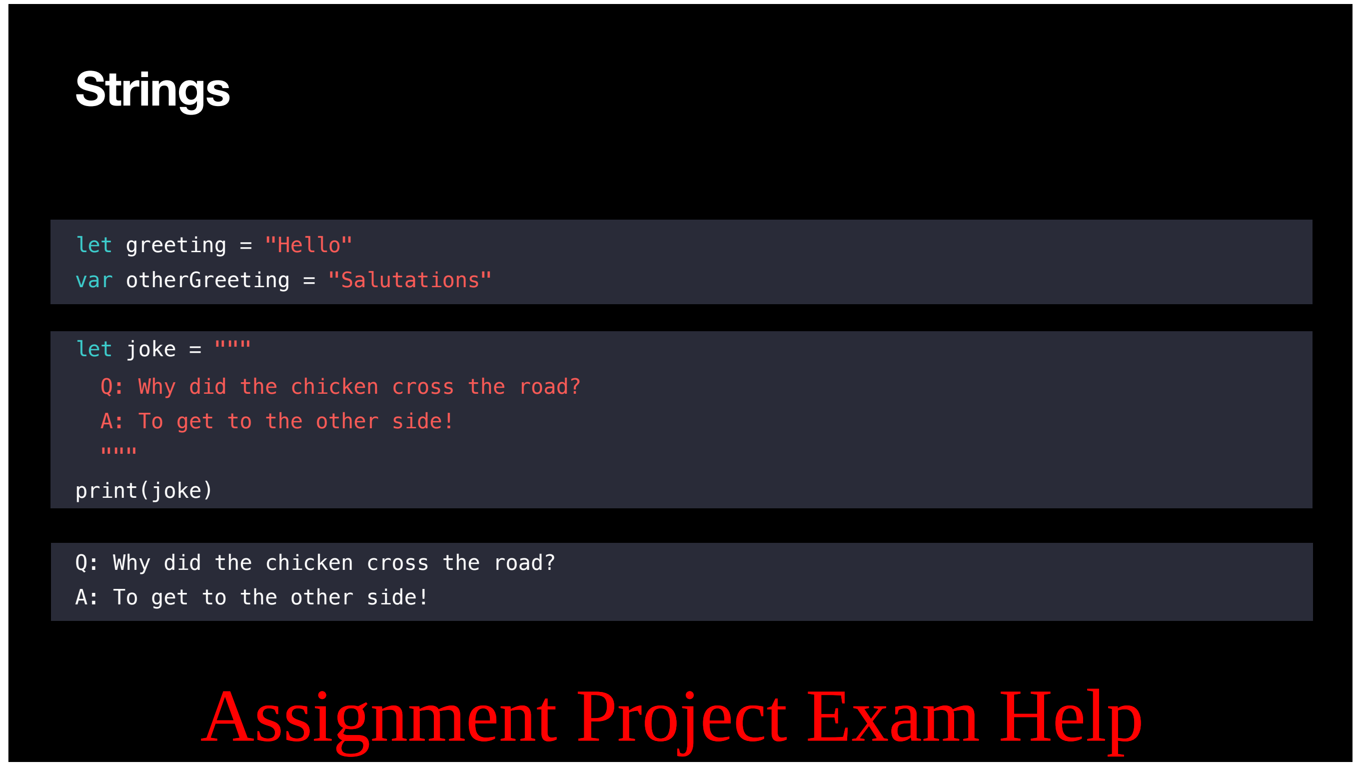


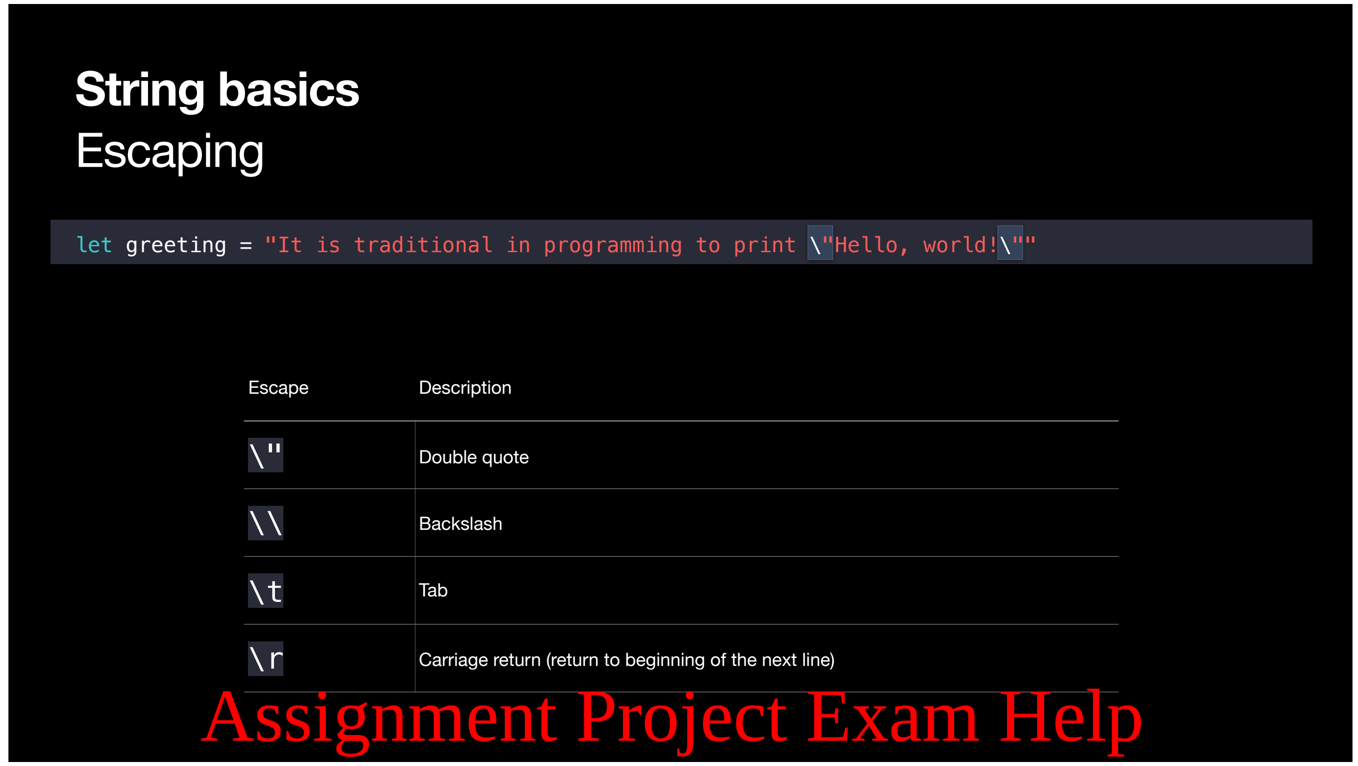
Say

- Because Swift needs to be used for things that other languages are used for, we need a String type.



Say

- String instances can be created with the `""` literal.
- `let` strings can't be changed.
- `var` strings can be changed.
- Unicode is supported.
- If your string literal needs to be multiple lines, simply surround your set of characters with three double quotation marks `"""`. In its multiline form, the string literal includes all of the lines between its opening and closing quotes. The string begins on the first line after the opening quotes and ends on the line before the closing quotes.



Do

• Click to display the escape quotes.

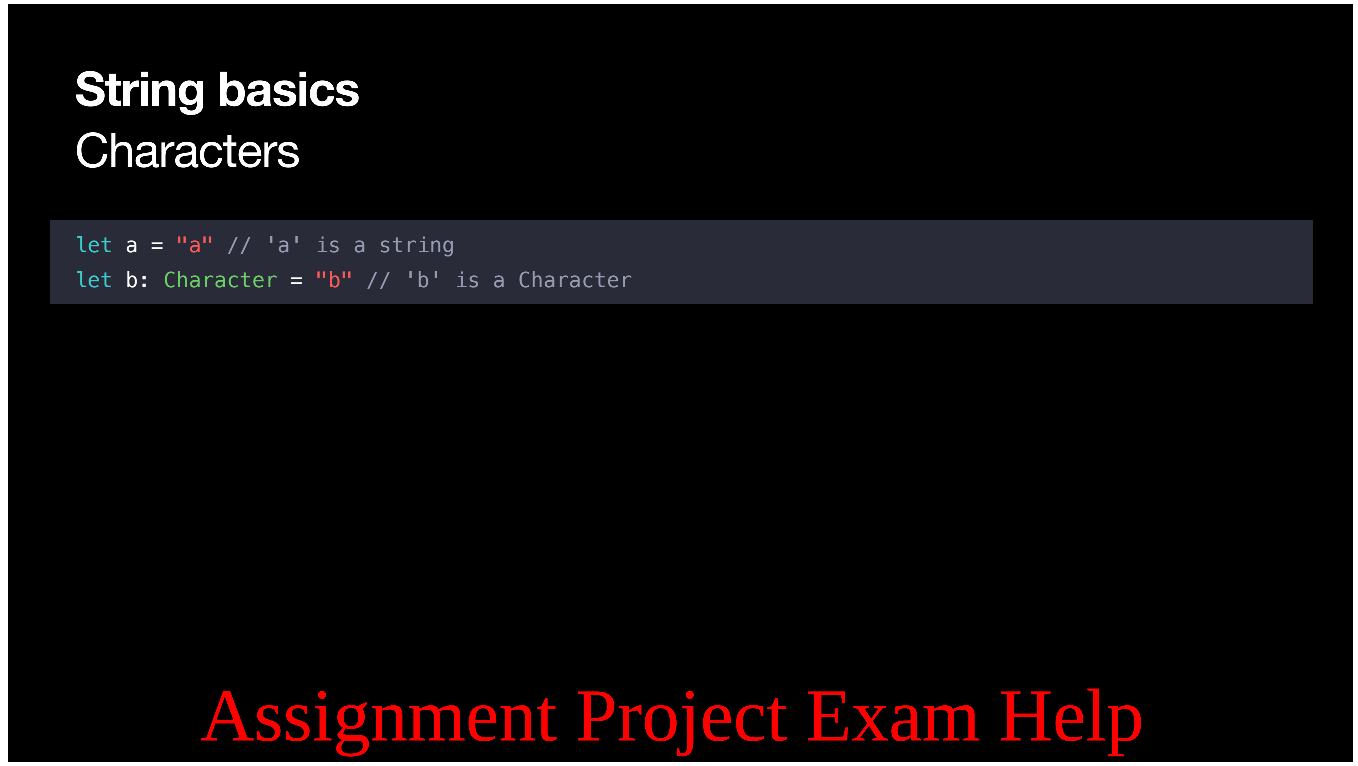
Say

• If the string will include double quotes, you'll need to use the backslash (\), known in Swift as the escape character.

String basics

Empty strings

```
var myString = ""

if myString.isEmpty {
  print("The string is empty")
}
```

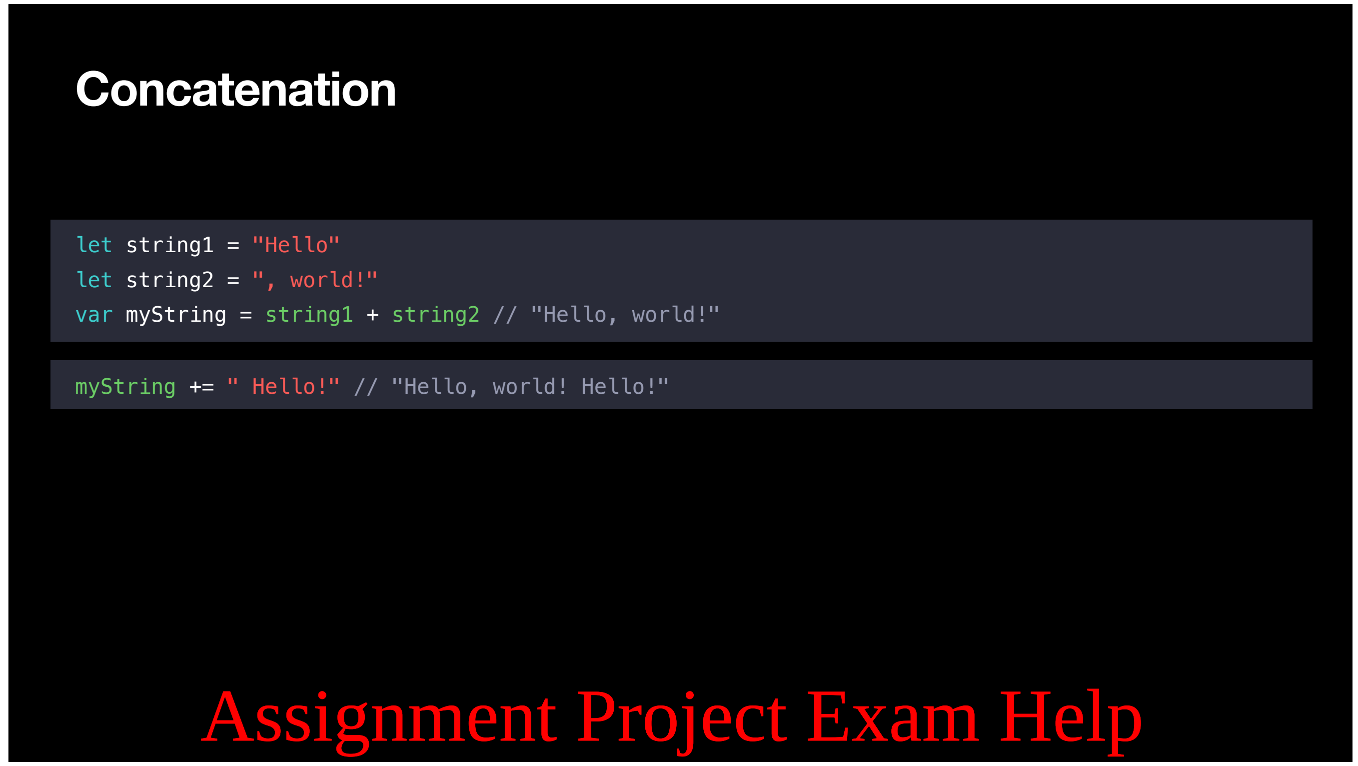


Say

- If you need a Character instance, you can still use "", but you need to specify the Character type.
- Swift strings are not a collection of characters, but you can get the characters (someString.characters).

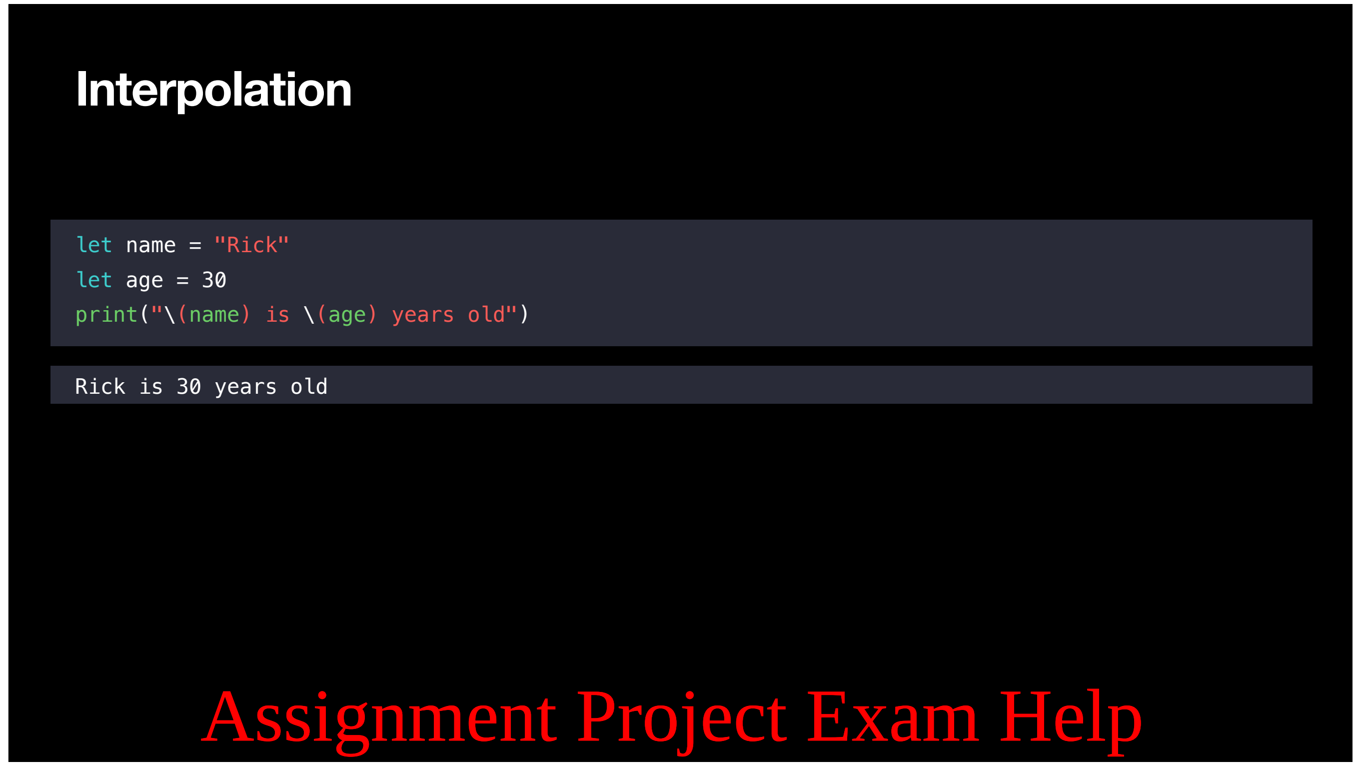
Do

- Show String under swiftdoc.org and point out the "extended grapheme clusters."
- Explain that if you need all the details of how String is built and stored, you can find it.



Say

- If you have two constant strings, you can use + to combine them into a new string.
- If you have a variable string, you can use += to append to it.
- As strings grow in complexity, the use of the + operator can make code tricky to handle. In the code above, for example, you might forget to add a space before "Hello!"



Say

- You can insert the raw value of a constant or variable into a String by preceding the name with a backslash \ and wrapping the name in parentheses ().

Interpolation

Expressions

```
let a = 4
let b = 5
print("If a is \(a) and b is \(b), then a + b equals \(a+b)")
```

```
If a is 4 and b is 5, then a + b equals 9
```

String equality and comparison

```swift
let month = "January"
let otherMonth = "January"
let lowercaseMonth = "january"

if month == otherMonth {
  print("They are the same")
}

if month != lowercaseMonth {
  print("They are not the same.")
}
```

```
They are the same.
They are not the same.
```

String equality and comparison

Ignoring case

```swift
let name = "Johnny Appleseed"
if name.lowercased() == "joHnnY aPPleseeD".lowercased() {
  print("The two names are equal.")
}
```

```
The two names are equal.
```


Say

- You can use the lowercased() method to normalize the two strings, comparing an all-lowercase version of the string with an all-lowercase version of the calling string.

String equality and comparison

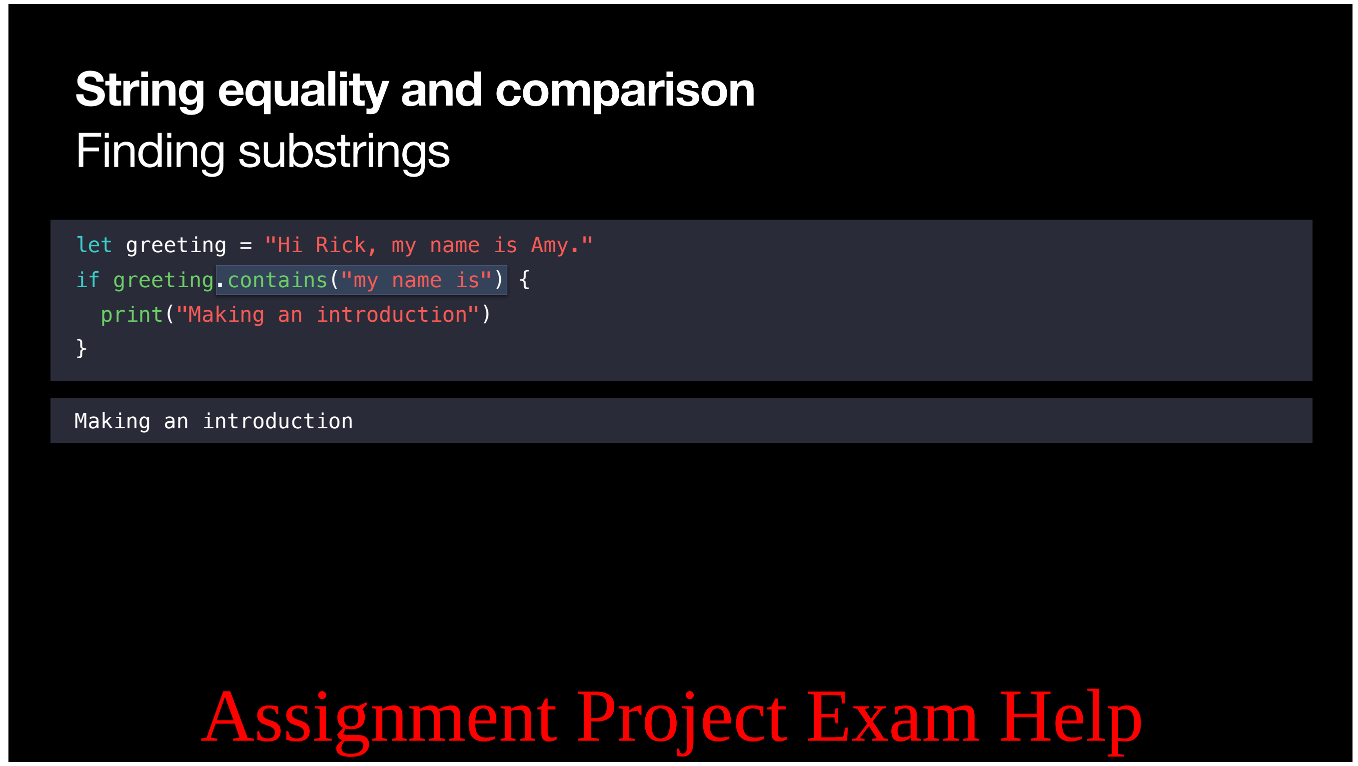
Prefix and suffix

```
let greeting = "Hello, world!"

print(greeting.hasPrefix("Hello"))
print(greeting.hasSuffix("world!"))
print(greeting.hasSuffix("World!"))
```

```
true
true
false
```

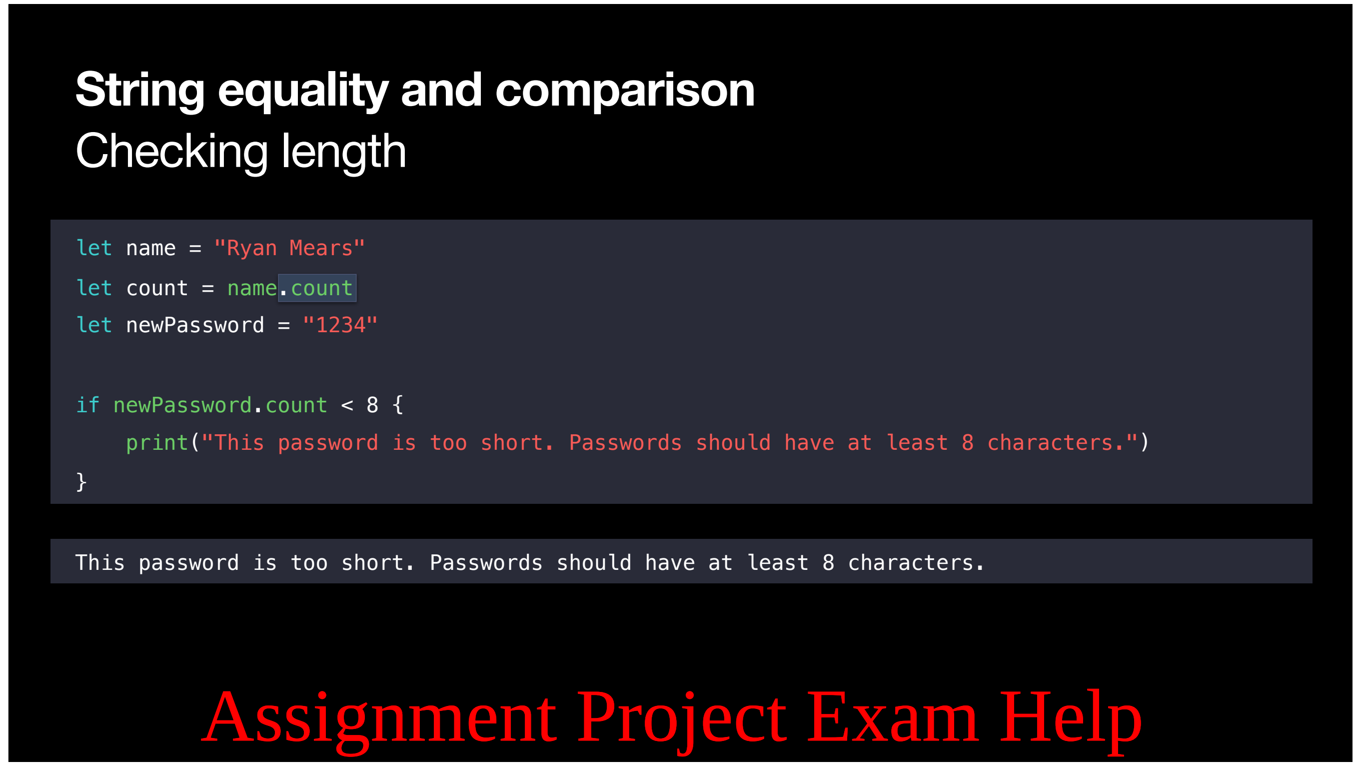

Say

- Note that they are case sensitive.



Say

- Use the contains(_:) method to return a Boolean value that indicates whether or not the substring was found.



Say

- You can use the count property to determine the number of characters.

String equality and comparison

Using switch

```
let someCharacter: Character = "e"
switch someCharacter {
  case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel.")
  default:
    print("\(someCharacter) is not a vowel.")
}
```

```
e is a vowel.
```


Say

- You can use the switch statement to pattern-match multiple values of strings or characters and respond accordingly.

Unicode

```
let cow = "🐮"
let credentials = "résumé"
let myBook = "私の本"
print("∞".characters.count)
```

```
1
```


Say

- Note that the size of a string in bytes is not equal to the number of characters.

Unit 2—Lesson 1
Lab: Strings

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

Unit 2—Lesson 2: Functions

Say

- Because most programs don't just run linear code, we need functions.

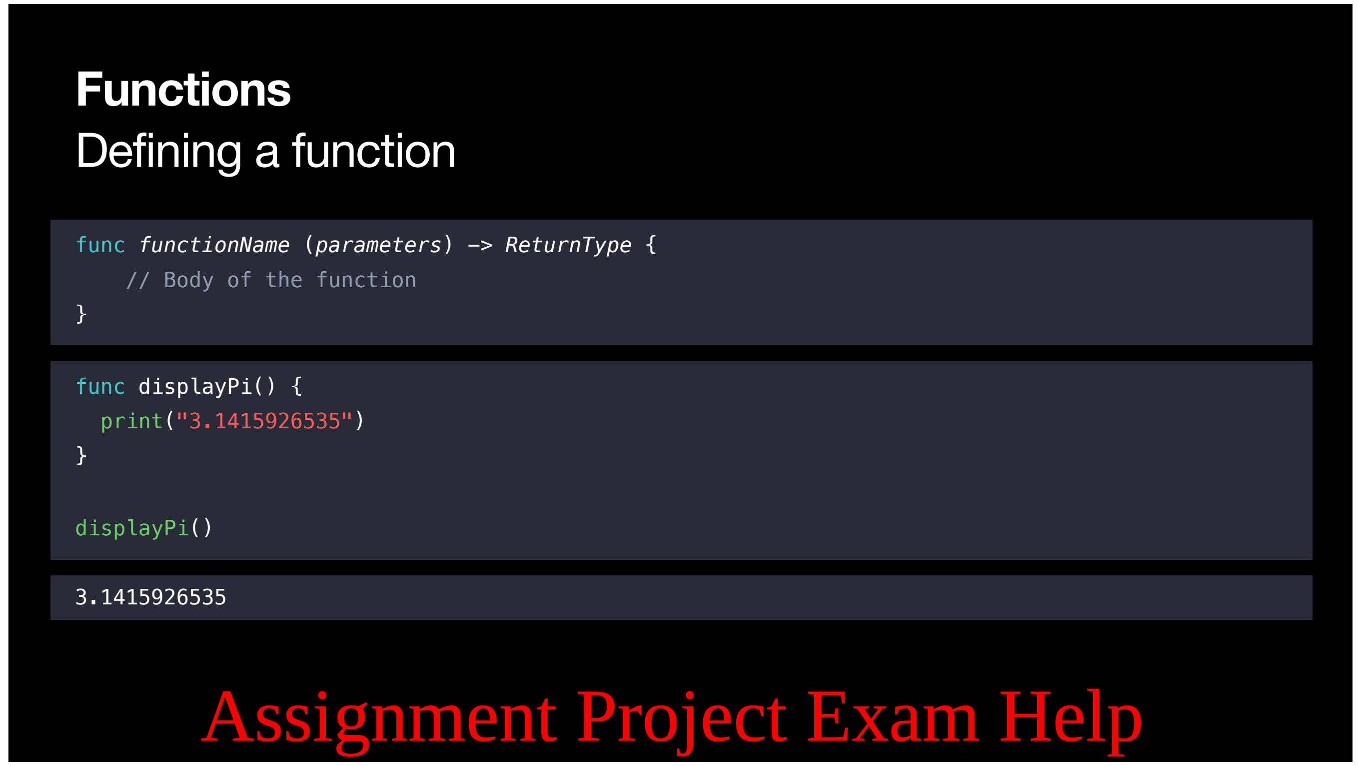
Functions

```
tieMyShoes()
```

```
makeBreakfast(food: "scrambled eggs", drink: "orange juice")
```


Say

- `tieMyShoes()`—No parameters required; shoes are always tied the same way.
- `makeBreakfast(food: ["eggs", "smoothie"])`—Use parameters when you want to do the work differently at different times.



Note

- This slide shows the syntax for defining a function.
- It also shows an implementation and call of a function with no arguments and no return.

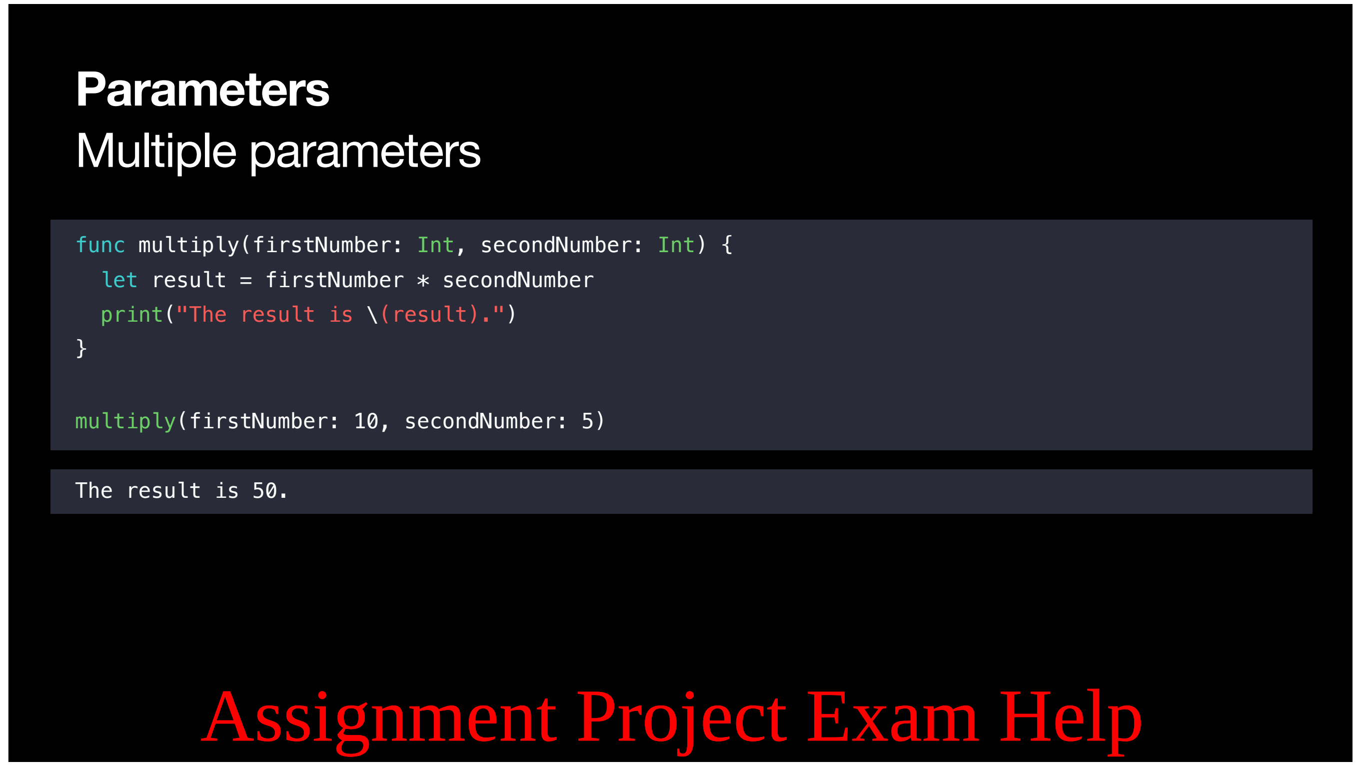
Parameters

```swift
func triple(value: Int) {
  let result = value * 3
  print("If you multiply \(value) by 3, you'll get \(result).")
}

triple(value: 10)
```

```
If you multiply 10 by 3, you'll get 30.
```

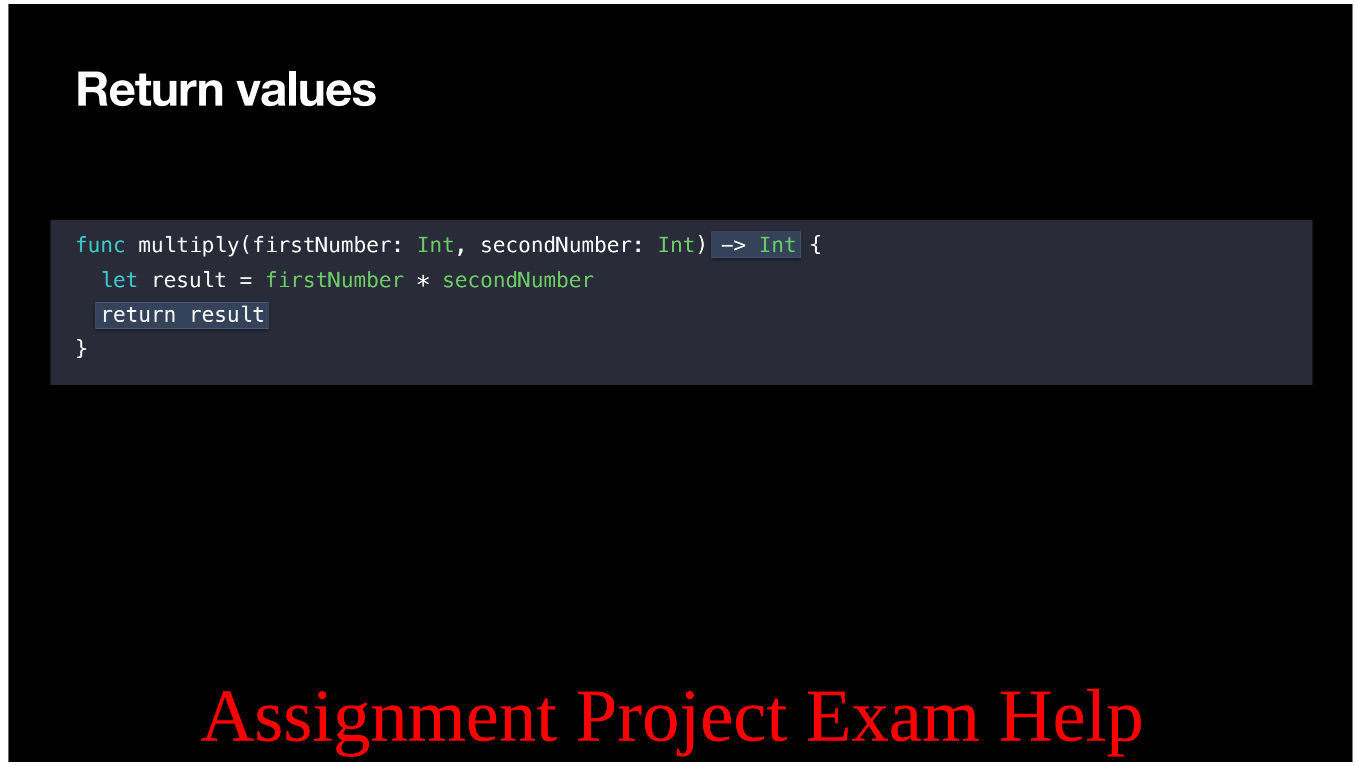

Note

- This slide shows an implementation and call of function with a parameter.



Say

- This slide shows using more than one argument.
- Properly named arguments help document the function ("self-documenting code").
- Note that named arguments must be passed in name order.



Do

- Talk about functions returning a result.
- Click to highlight `-> Int` to show the return type.
- Click to highlight `return result` to show how the function returns the result.

Return values

```swift
func multiply(firstNumber: Int, secondNumber: Int) -> Int {
 return firstNumber * secondNumber
}
```

```swift
let myResult = multiply(firstNumber: 10, secondNumber: 5)
print("10 * 5 is \(myResult)")
```

```swift
print("10 * 5 is \(multiply(firstNumber: 10, secondNumber: 5))")
```


Do

- Click to display capturing the return value, then printing it.
- Click again to display calling the function in String interpolation.

Argument labels

```swift
func sayHello(firstName: String) {
  print("Hello, \(firstName)!")
}

sayHello(firstName: "Amy")
```


Say

- Commonly use the same label internally and externally.

Argument labels

```swift
func sayHello(to: String, and: String) {
  print("Hello \(to) and \(and)")
}

sayHello(to: "Luke", and: "Dave")
```


Say

- Sometimes having the same names inside and when calling isn't so good.

Argument labels

External names

```
func sayHello(to person: String, and anotherPerson: String) {
  print("Hello \(person) and \(anotherPerson)")
}

sayHello(to: "Luke", and: "Dave")
```


Say

- It may be better to have different descriptive names for calling and implementing.

Argument labels
Omitting labels

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

```swift
func add(_ firstNumber: Int, to secondNumber: Int) -> Int {
    return firstNumber + secondNumber
}

let total = add(14, to: 6)
```


Say

- First is example of a function students have used that omitted the label.
- The second example defines a function that omits the label for the first parameter.

Default parameter values

```swift
func display(teamName: String, score: Int = 0) {
  print("\(teamName): \(score)")
}

display(teamName: "Wombats", score: 100)
display(teamName: "Wombats")
```

```
Wombats: 100
Wombats: 0
```


Say

- Functions can have default values for parameters, and if so, you don't need to pass them.
- Note that functions can have more than one parameter with a default value.
- This is a tricky language feature; go to the documentation to see the full function.

Unit 2—Lesson 2

Lab: Functions

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

Unit 2—Lesson 3: Structures

Note

- This is a long presentation.
- The recommendation is to stop partway through and ask students to complete a few of the lab exercises.
- Then resume the lecture, followed by students completing the exercises.

Structures

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

Capitalize type names

Use lowercase for property names

Say

- Use the struct keyword.
- Note that because structs are value types, if you have a struct with var properties and an instance created into a let variable, the properties aren't changeable.

Structures

Accessing property values

```swift
struct Person {
  var name: String
}

let person = Person(name: "Jasmine")
print(person.name)
```

```
Jasmine
```


Say

- Use the dot syntax to access properties.

Structures

Adding functionality

```
struct Person {
  var name: String

 func sayHello() {
    print("Hello there! My name is \(name)!")
  }

}

let person = Person(name: "Jasmine")
person.sayHello()
```

```
Hello there! My name is Jasmine!
```


Say

- Structures can have behavior.
- "Methods" are functions on a type.

Instances

```swift
struct Shirt {
  var size: String
  var color: String
}

let myShirt = Shirt(size: "XL", color: "blue")

let yourShirt = Shirt(size: "M", color: "red")
```

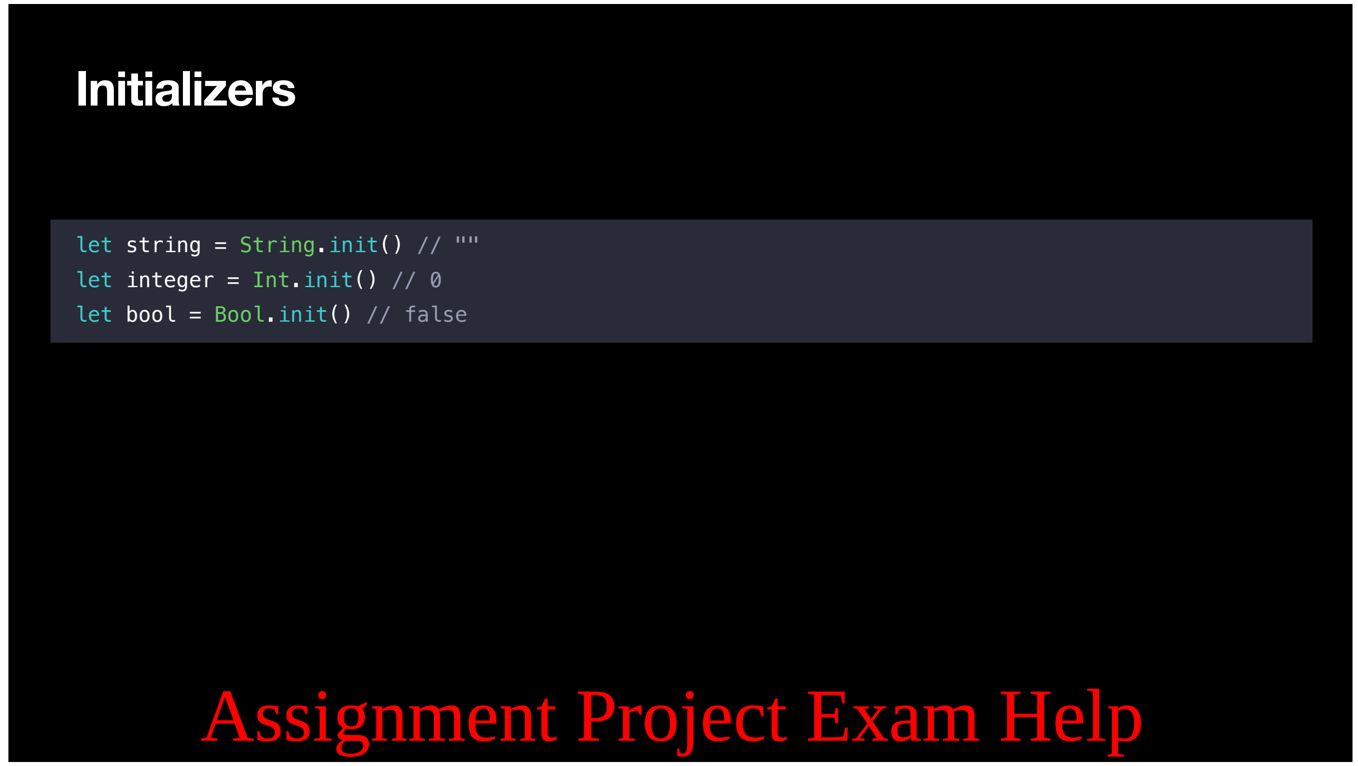

Note

• This slide shows creating two instances of a struct.

```swift
struct Car {
  var make: String
  var year: Int
  var color: String

  func startEngine() {...}

  func drive() {...}

  func park() {...}

  func steer(direction: Direction) {...}
}

let firstCar = Car(make: "Honda", year: 2010, color: "blue")
let secondCar = Car(make: "Ford", year: 2013, color: "black")

firstCar.startEngine()
firstCar.drive()
```

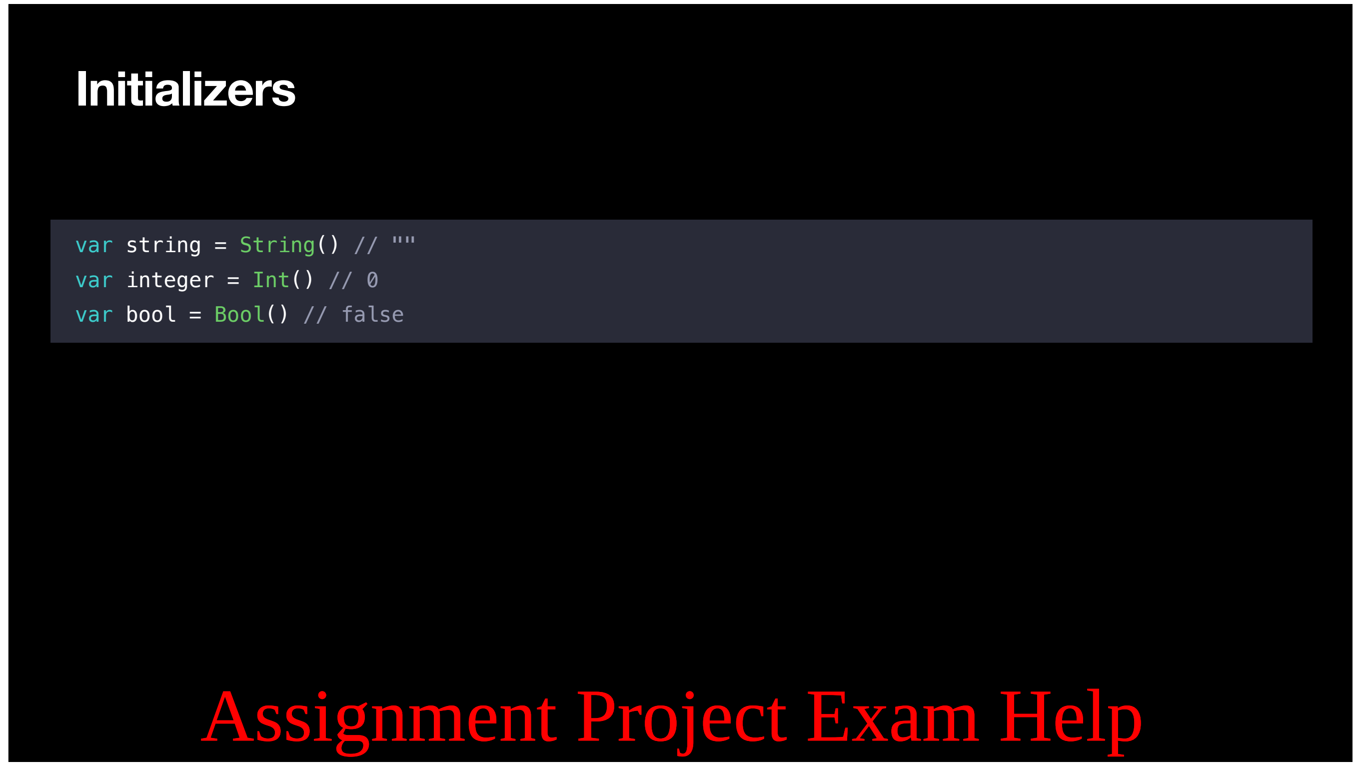

Note

- This slide shows how to call methods on an instance: The firstCar has driven away, and the secondCar is still sitting there, not running.



Say

- The standard library types all have `init()`, which returns an empty or default instance.



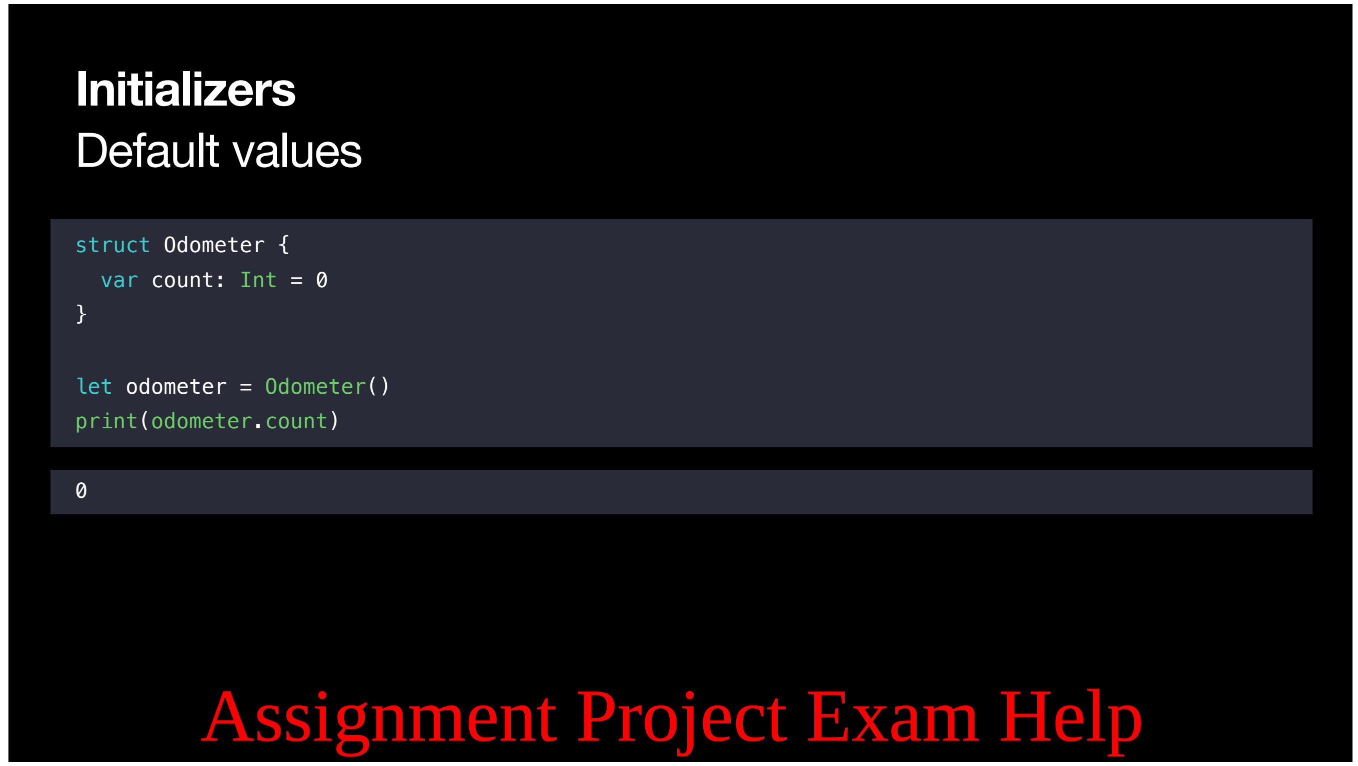
Say

• Whenever you’re using a new type, look at the inits. How can I get an instance?

Note

• This slide shows the () shortcut.

• Go to String in the docs and show the inits. They can all be run with or without the init. part.

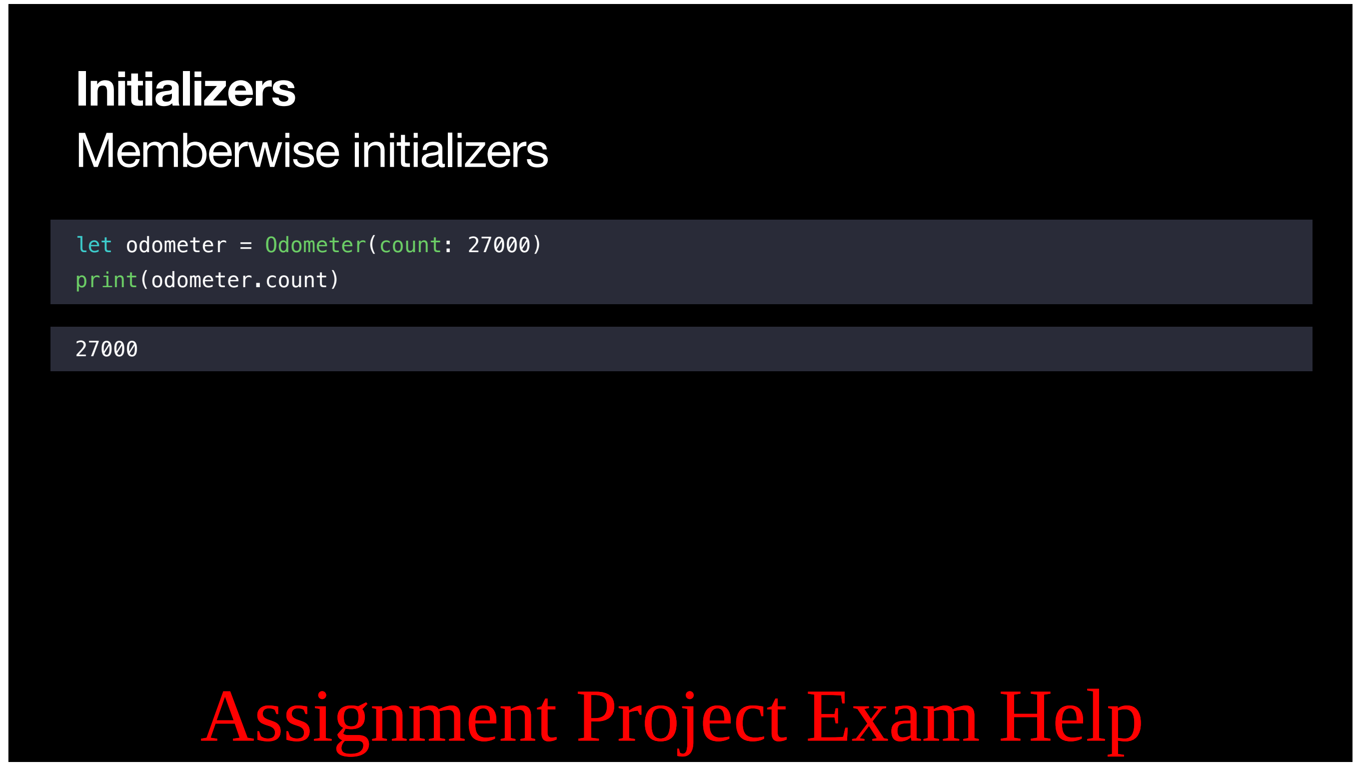


Say

- For types we create: If all the stored properties of your struct have default values, the compiler writes the no-argument initializer for you
- init() creates an instance with default values.

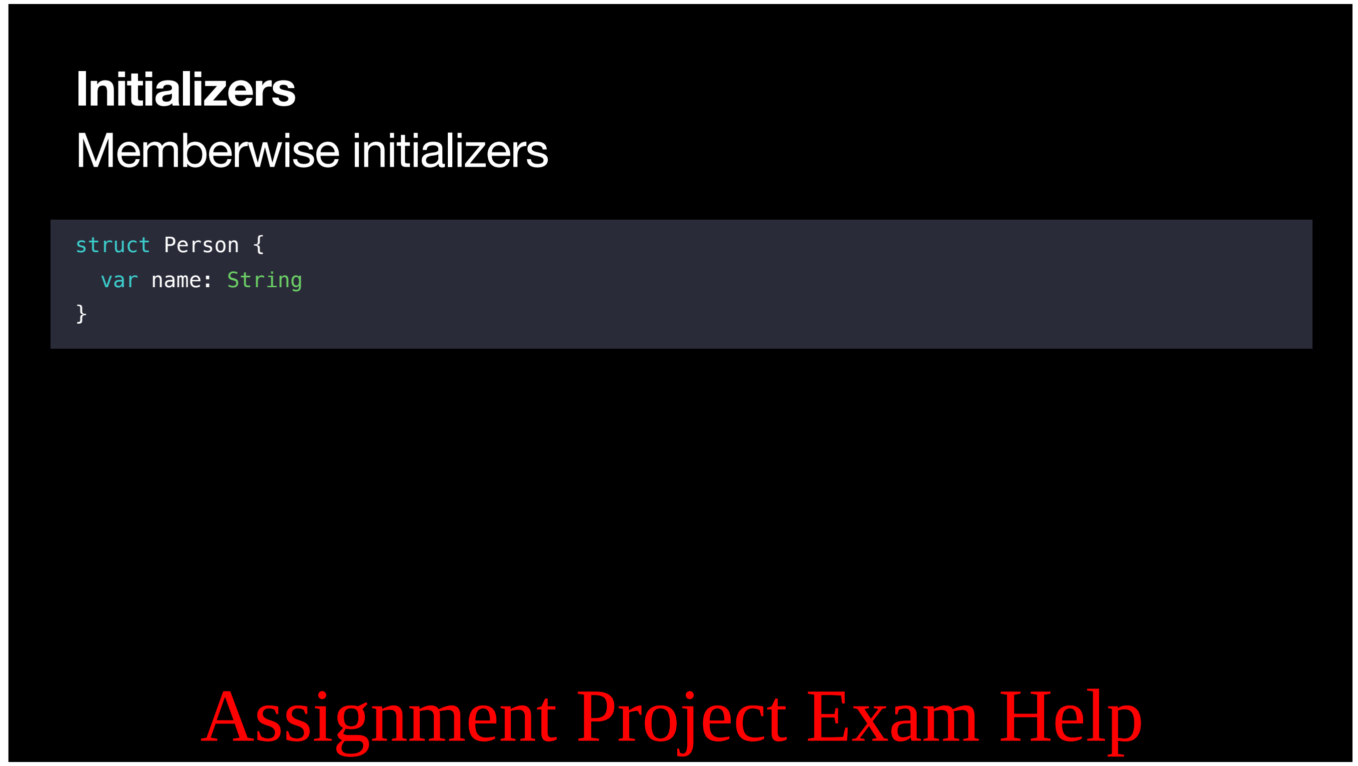
Note

- Some time during this part of the lesson, you should say that before initialization completes, all properties need a value.



Say

- Structs always get a memberwise initializer from the compiler, whether or not you have default values.
- We saw that Odometer has a default value, but we can override that by calling the memberwise initializer.



Say

- Person with a "name" property probably shouldn't have a default var name: String (no default value, so no init())
- So call the memberwise initializer.

Note

- The next slide builds upon this one.

Initializers
Memberwise initializers

```
struct Person {
  var name: String

  func sayHello() {
    print("Hello there!")
  }
}

let person = Person(name: "Jasmine") // Memberwise initializer
```

```swift
struct Shirt {
  let size: String
  let color: String
}

let myShirt = Shirt(size: "XL", color: "blue") // Memberwise initializer


struct Car {
  let make: String
  let year: Int
  let color: String
}

let firstCar = Car(make: "Honda", year: 2010, color: "blue") // Memberwise initializer
```


Note

- This slide shows two structs and their respective memberwise initializers.

Initializers

Custom initializers

```swift
struct Temperature {
  var celsius: Double
}

let temperature = Temperature(celsius: 30.0)
```

```swift
let fahrenheitValue = 98.6
let celsiusValue = (fahrenheitValue - 32) / 1.8

let newTemperature = Temperature(celsius: celsiusValue)
```


Say

- In this example, the memberwise initializer requires you to calculate the CelsiusValue before you initialize a newTemperature object.

Do

- Click to display using the Fahrenheit example.

Say

- What if we want to create one from a Fahrenheit value? We'd have to convert to Celsius in code and then pass that in.

```
struct Temperature {
  var celsius: Double

  init(celsius: Double) {
    self.celsius = celsius
  }

  init(fahrenheit: Double) {
    celsius = (fahrenheit - 32) / 1.8
  }
}

let currentTemperature = Temperature(celsius: 18.5)
let boiling = Temperature(fahrenheit: 212.0)

print(currentTemperature.celsius)
print(boiling.celsius)

18.5
100.0
```


Say

- Instead of calculating Fahrenheit > Celsius outside the struct, make an initializer that takes a Fahrenheit value and converts it for us.
- When we write our own initializers, we must make sure all properties are set to something before we're done, so this one is valid.
- Show String in the docs and point out the many init(...) methods.
- Note that as soon as we write ANY init methods, we no longer get:
  - The init() that we get by having default values
  - The memberwise initializer

Note

- If we write inits in an extension, we still get the compiler-written ones.

Unit 2—Lesson 3

Lab: Structures

Open and complete the following exercises in `Lab – Structures.playground`:

- Exercise - Structs, Instances, and Default Values
- App Exercise - Workout Tracking

Instance methods

```swift
struct Size {
  var width: Double
  var height: Double

  func area() -> Double {
    return width * height
  }
}

var someSize = Size(width: 10.0, height: 5.5)

let area = someSize.area() // Area is assigned a value of 55.0
```


Say

- The methods so far are "instance methods," meant to be called on an instance.
- Calling area() will return different values depending on the width and height of the receiving instance.

In the following example, a simple structure stores mileage data about a specific Car object. Before looking at the code, consider what data the mileage counter needs to store and what actions it needs to perform.

1. Store the mileage count to be displayed on an odometer
2. Increment the mileage count to update the mileage when the car drives
3. Potentially reset the mileage count if the car drives beyond the number of miles that can be displayed on the odometer

The last two require modifying the count property within the struct.

```swift
struct Odometer {
  var count: Int = 0 // Assigns a default value to the 'count' property.

  mutating func increment() {
    count += 1
  }

  mutating func increment(by amount: Int) {
    count += amount
  }

  mutating func reset() {
    count = 0
  }
}

var odometer = Odometer() // odometer.count defaults to 0
odometer.increment() // odometer.count is incremented to 1
odometer.increment(by: 15) // odometer.count is incremented to 16
odometer.reset() // odometer.count is reset to 0
```


Note

- Odometer type—Can have A and B trip odometers like a car does; each its own instance with its own count property value.
- Note mutating—If a method on a value type changes a property, it must be annotated with mutating (another example of Swift safety).
- Note that mutating isn't required for Classes.

Computed properties

```swift
struct Temperature {
  let celsius: Double
  let fahrenheit: Double
  let kelvin: Double
}

let temperature = Temperature(celsius: 0, fahrenheit: 32, kelvin: 273.15)
```


Say

- Let’s say we want to be able to get Celsius, Fahrenheit, and kelvin from a Temperature instance.
- Here’s a bad way to do it: properties for all three and a memberwise initializer.
- It’s bad because the caller has to calculate all three values to pass in.

```swift
struct Temperature {
  var celsius: Double
  var fahrenheit: Double
  var kelvin: Double

  init(celsius: Double) {
    self.celsius = celsius
    fahrenheit = celsius * 1.8 + 32
    kelvin = celsius + 273.15
  }

  init(fahrenheit: Double) {
    self.fahrenheit = fahrenheit
    celsius = (fahrenheit - 32) / 1.8
    kelvin = celsius + 273.15
  }

  init(kelvin: Double) {
    self.kelvin = kelvin
    celsius = kelvin - 273.15
    fahrenheit = celsius * 1.8 + 32
  }
}

let currentTemperature = Temperature(celsius: 18.5)
let boiling = Temperature(fahrenheit: 212.0)
let freezing = Temperature(kelvin: 273.15)
```


Say

- Another way would be an initializer for each scale of temperature.
- It’s better for the inits—only one value is needed.
- It’s still challenging for the state. Imagine that this thing can somehow do measurements itself; any time the temperature changes, all three properties would need to be updated.

```swift
struct Temperature {
  var celsius: Double

  var fahrenheit: Double {
    return celsius * 1.8 + 32
  }
}

let currentTemperature = Temperature(celsius: 0.0)
print(currentTemperature.fahrenheit)

32.0
```


Say

- Computed properties would potentially simplify the state—have one Celsius property and compute the others.
- You would probably still want an init for each kind of temperature, but only one property for state.

Note

- Computed properties are "get" and optionally "set," and if there's only the getter you can omit the "get."
- Computed properties must be "var."

Challenge

Add support for Kelvin

Modify the following to allow the temperature to be read as Kelvin

```
struct Temperature {
  let celsius: Double

  var fahrenheit: Double {
    return celsius * 1.8 + 32
  }

}
```

Hint: Temperature in Kelvin is Celsius + 273.15

Do

- Add the kelvin-computed property as a demo or walkthrough:

```
var kelvin: Double {
      return Celsius + 273.15
}
```

```
struct Temperature {
  let celsius: Double

  var fahrenheit: Double {
    return celsius * 1.8 + 32
  }

  var kelvin: Double {
    return celsius + 273.15
  }

}

let currentTemperature = Temperature(celsius: 0.0)
print(currentTemperature.kelvin)

273.15
```


Note

- Here’s the solution for the challenge.

Property observers

```swift
struct StepCounter {
    var totalSteps: Int = 0 {
        willSet {
            print("About to set totalSteps to \(newValue)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
```


Say

- Swift allows you to observe any property and respond to the changes in the property's value.
- willSet is an observer that defines a block of code that will be called before a property's value is set.
  You will have access to the new value that will be set to the property in a constant value named newValue.
- didSet defines a block of code that will be called after a property's value has been set.
  You will have access to the previous value of the property in a constant value named oldValue.

Property observers

```
var stepCounter = StepCounter()
stepCounter.totalSteps = 40
stepCounter.totalSteps = 100
```

```
About to set totalSteps to 40
Added 40 steps
About to set totalSteps to 100
Added 60 steps
```

Type properties and methods

```
struct Temperature {
  static var boilingPoint = 100.0

  static func convertedFromFahrenheit(_ temperatureInFahrenheit: Double) -> Double {
    return(((temperatureInFahrenheit - 32) * 5) / 9)
  }

}

let boilingPoint = Temperature.boilingPoint

let currentTemperature = Temperature.convertedFromFahrenheit(99)

let positiveNumber = abs(-4.14)
```


Say

- Use the static keyword on a property to make it "one per type."
- Ask the type for the property.
- The naming convention is that types are capitalized and everything else is lowercase. This helps you see what’s going on.

Copying

```swift
var someSize = Size(width: 250, height: 1000)
var anotherSize = someSize

someSize.width = 500

print(someSize.width)
print(anotherSize.width)
```

```
500
250
```


Say

- Struct is a "value type"—copied on assignment when passed into a method or function and when returned from a function.
- This shows that if a struct is copied and then a change is made to the original, the copy doesn't change.

Note

- This is conceptual—"copy on write" is really how it works. A copy isn't made unless a property changes.

self

```swift
struct Car {
  var color: Color

  var description: String {
    return "This is a \(self.color) car."
  }
}
```


Say

- "self" is the instance itself.
- This shows that "self.color" works to access properties.

self

When not required

Not required when property or method names exist on the current object

```
struct Car {
  var color: Color

  var description: String {
    return "This is a \(color) car."
  }
}
```

self
When required

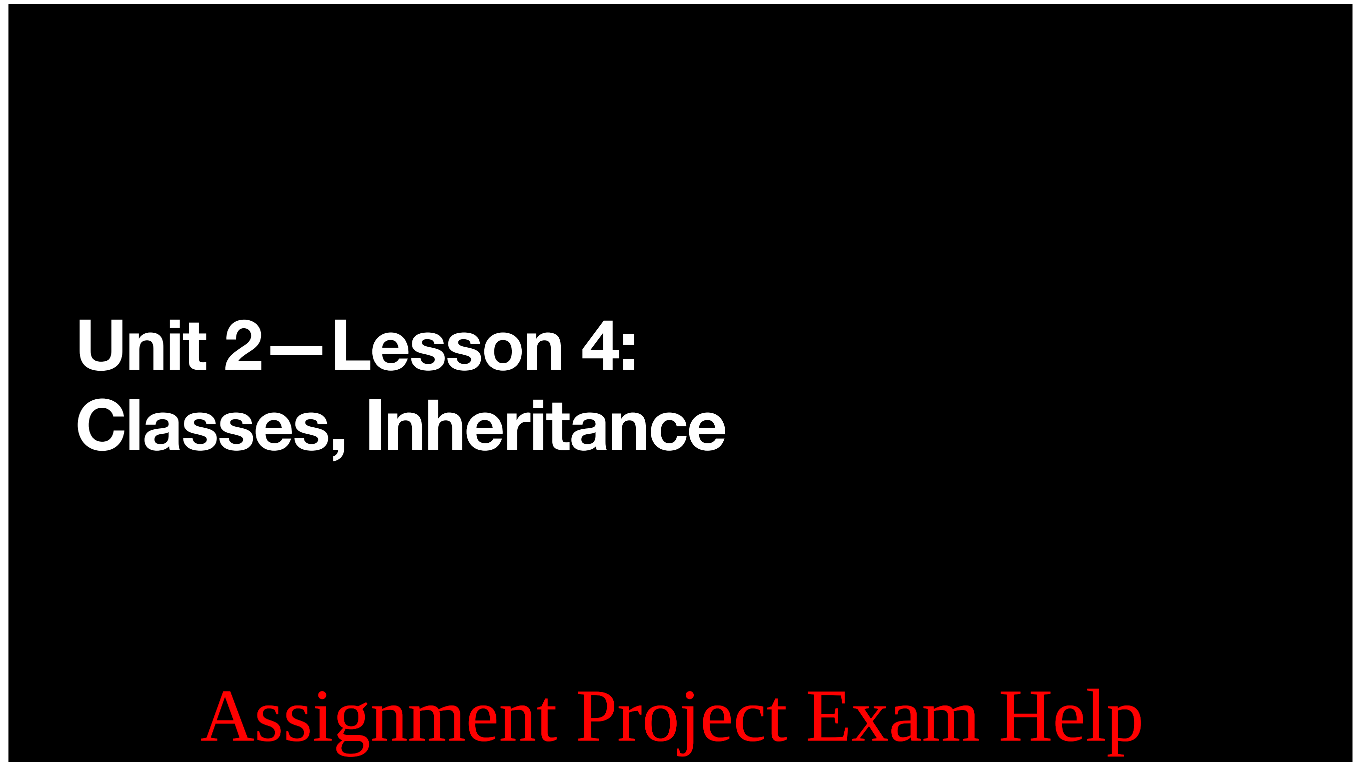
```
struct Temperature {
  var celsius: Double

  init(celsius: Double) {
    self.celsius = celsius
  }
}
```


Say

- In this example, the argument to init is the same as the property, so "self" is required to disambiguate.

Unit 2—Lesson 3
Lab: Structures

Open and complete the remaining exercises in
`Lab - Structures.playground`



Note

- Depending upon how long it takes for you to go through the early slides, you may want to break after about 15 minutes and ask students to complete a few of the lab exercises.
- Then resume the lecture and complete the remaining exercises at the end.

Say

- We're going to see that classes and structures are very similar.
- The main differences we'll see in this lesson are inheritance, and value versus reference types.

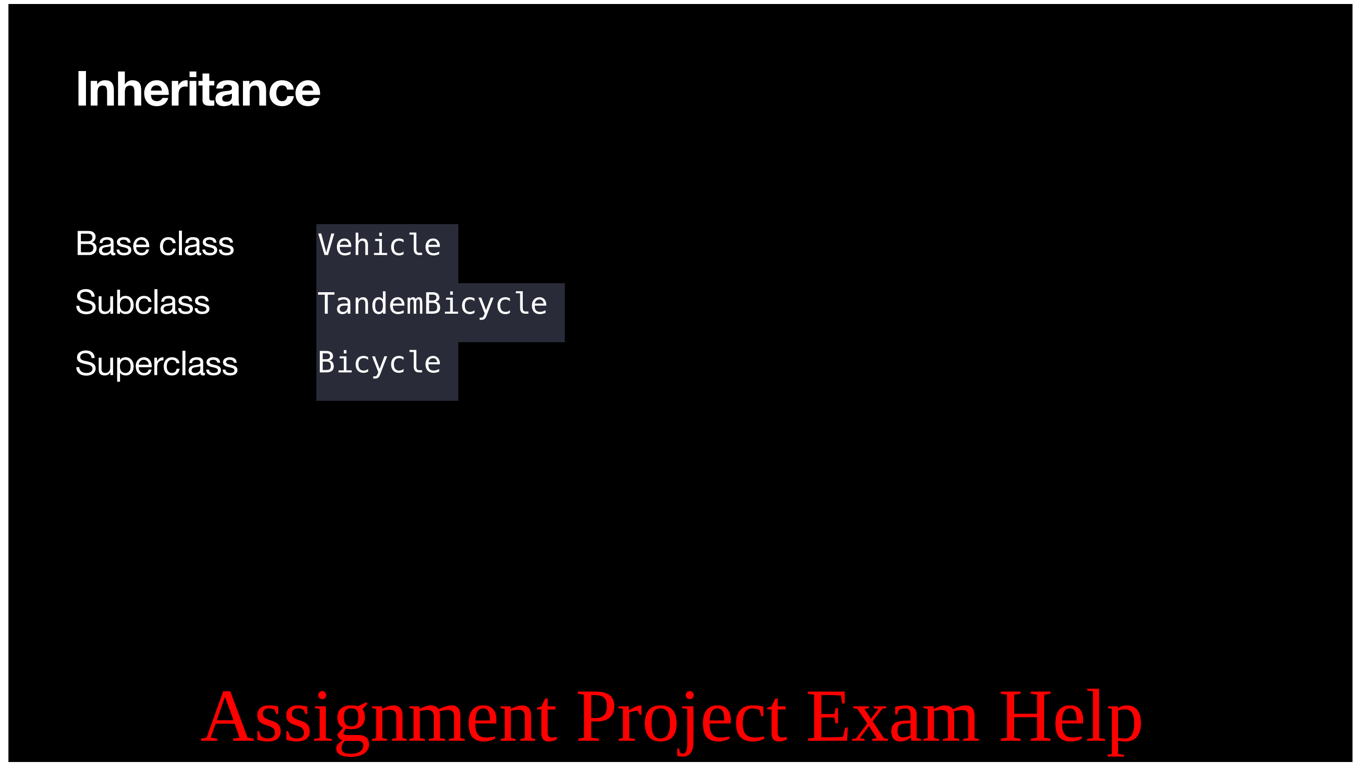
```swift
class Person {
  let name: String

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

  func sayHello() {
    print("Hello there!")
  }
}

let person = Person(name: "Jasmine")
print(person.name)
person.sayHello()

Jasmine
Hello there!
```


Say

- Use the class keyword to create a class
- In this example, that's the only difference from a struct.
- As part of explanation for class, comparing it to struct, discuss how the capitalization indicates that "Person" is the class or type and "person" is an instance.

Do

- Ask: "Why is self required in the init?"
- Answer: Because it’s ambiguous otherwise.



Say

- Use the class keyword to create a class
- In this example, that's the only difference from a struct.
- As part of explanation for class, comparing it to struct, discuss how the capitalization indicates that "Person" is the class or type and "person" is an instance.

Do

- Ask: "Why is self required in the init?"
- Answer: Because it’s ambiguous otherwise.

Inheritance

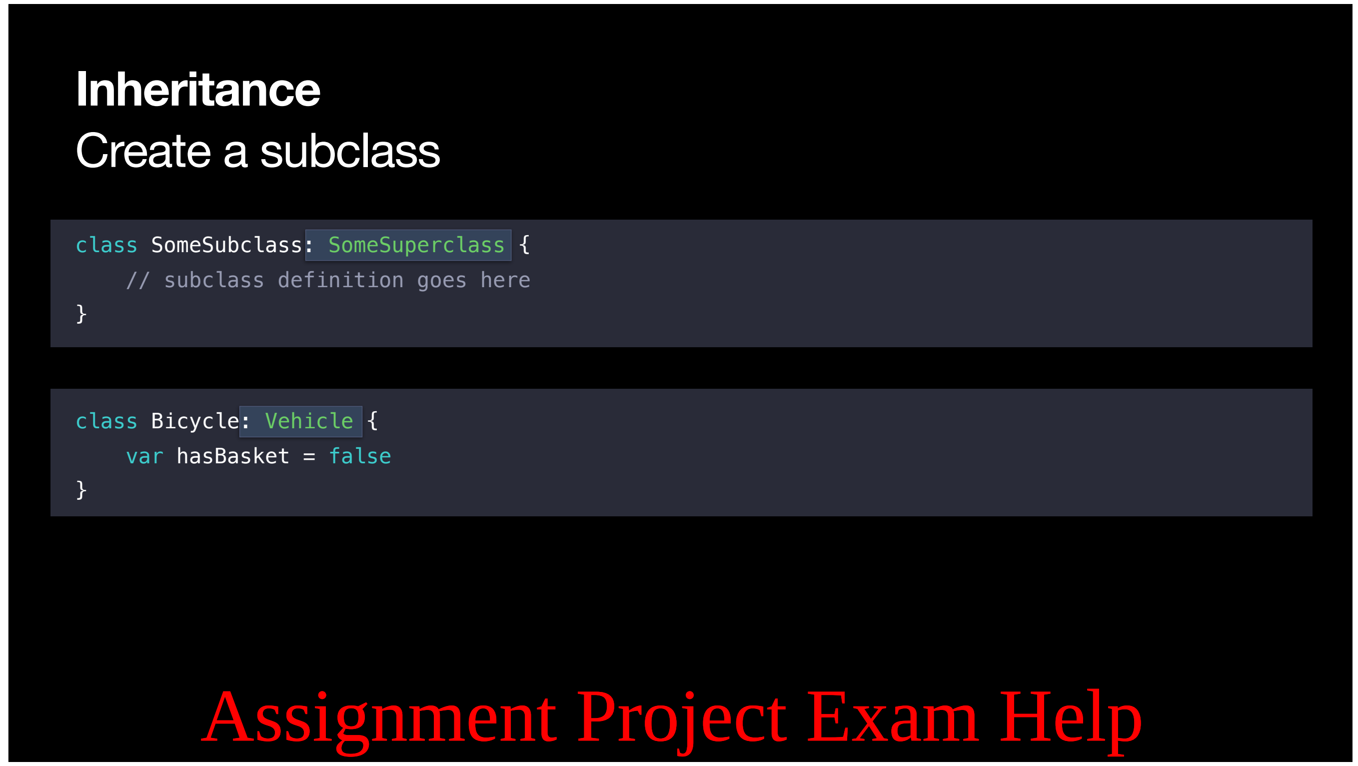
Defining a base class

```
class Vehicle {
   var currentSpeed = 0.0

   var description: String {
      return "traveling at \(currentSpeed) miles per hour"
   }

   func makeNoise() {
      // do nothing — an arbitrary vehicle doesn't necessarily make a noise
   }
}

let someVehicle = Vehicle()
print("Vehicle: \(someVehicle.description)")
```

```
Vehicle: traveling at 0.0 miles per hour
```


Note

- This slide shows a `Vehicle` class.



Note

- This slide shows how to make a subclass.
- This slide shows `Bicycle` as subclass of `Vehicle`: `Bicycle` adds a `hasBasket` property.

Inheritance

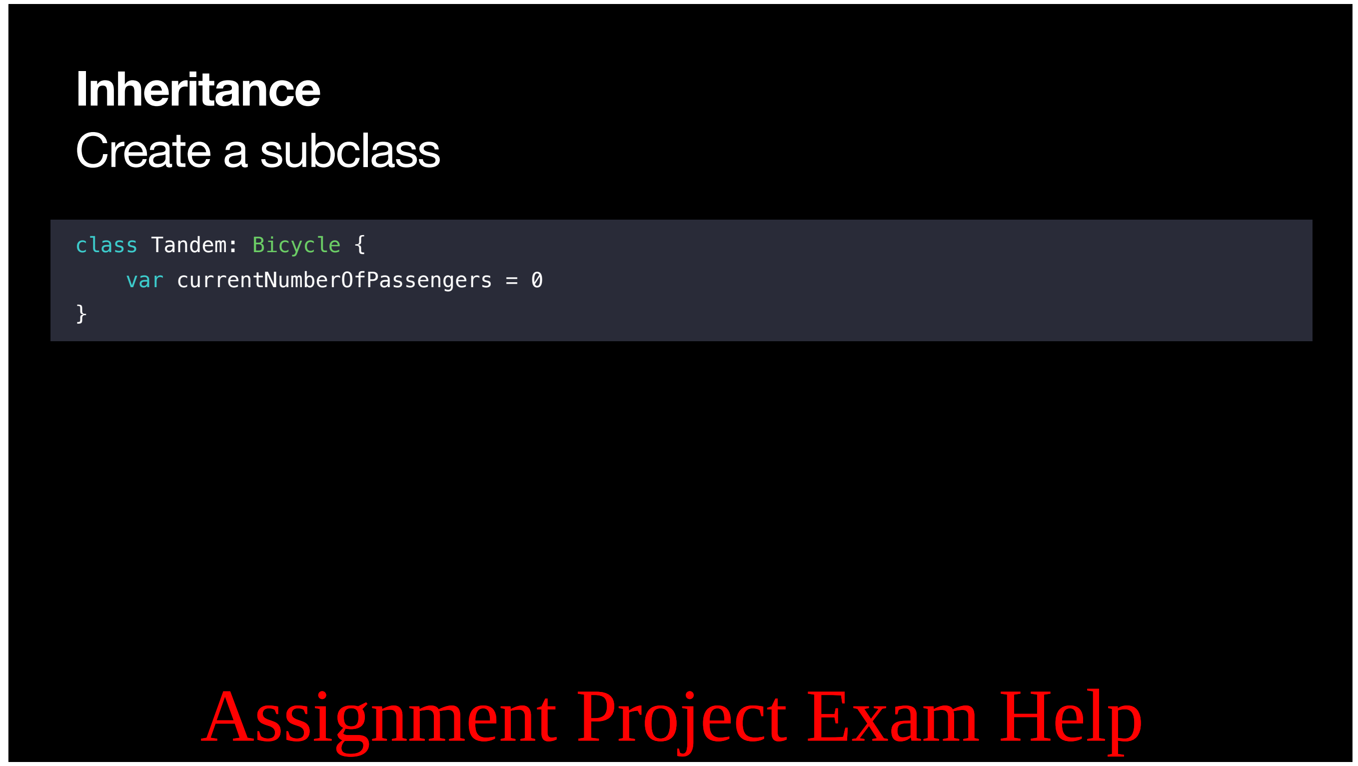
Create a subclass

```swift
class Bicycle: Vehicle {
    var hasBasket = false
}

let bicycle = Bicycle()
bicycle.hasBasket = true

bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
```

```
Bicycle: traveling at 15.0 miles per hour
```

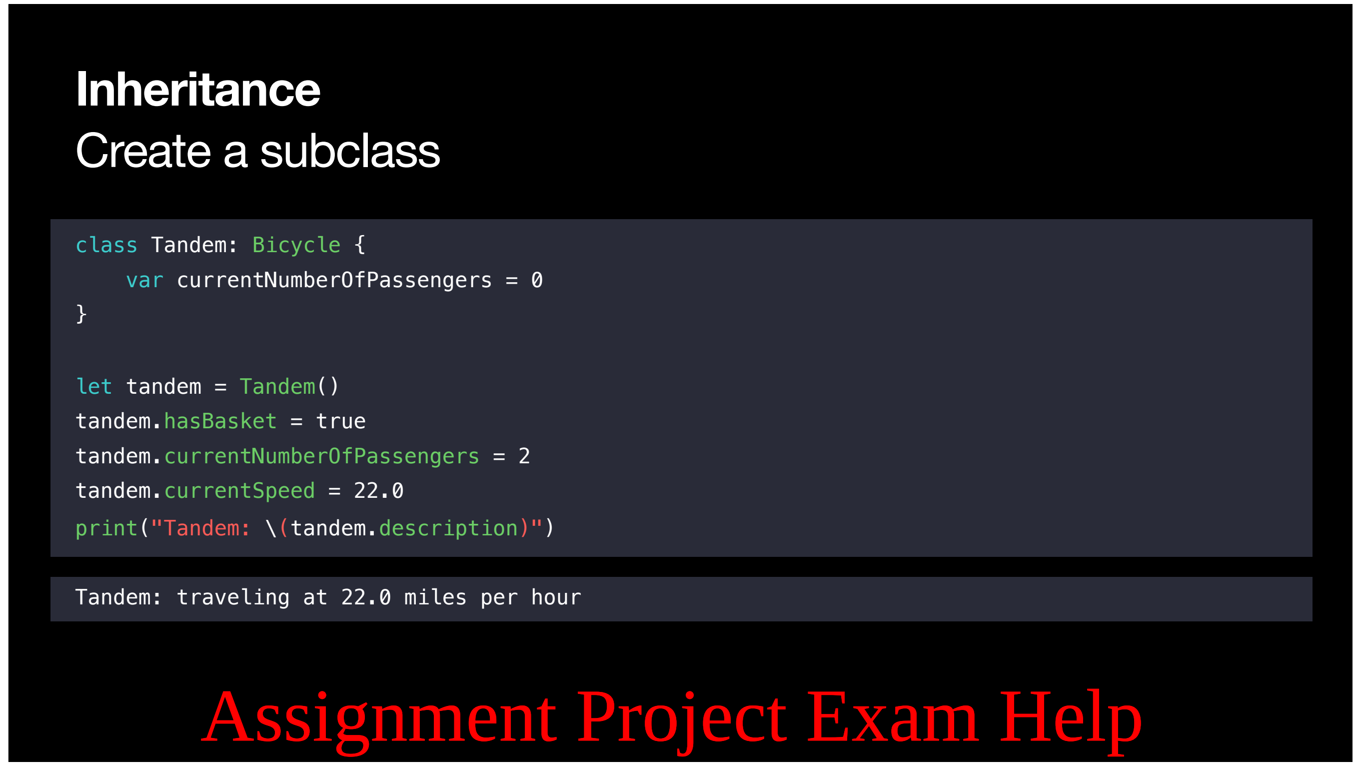

Note

- This slide shows how to:
  - Make an instance of Bicycle
  - Set a Vehicle property on bicycle (currentSpeed)
  - Set a Bicycle property on bicycle (hasBasket)



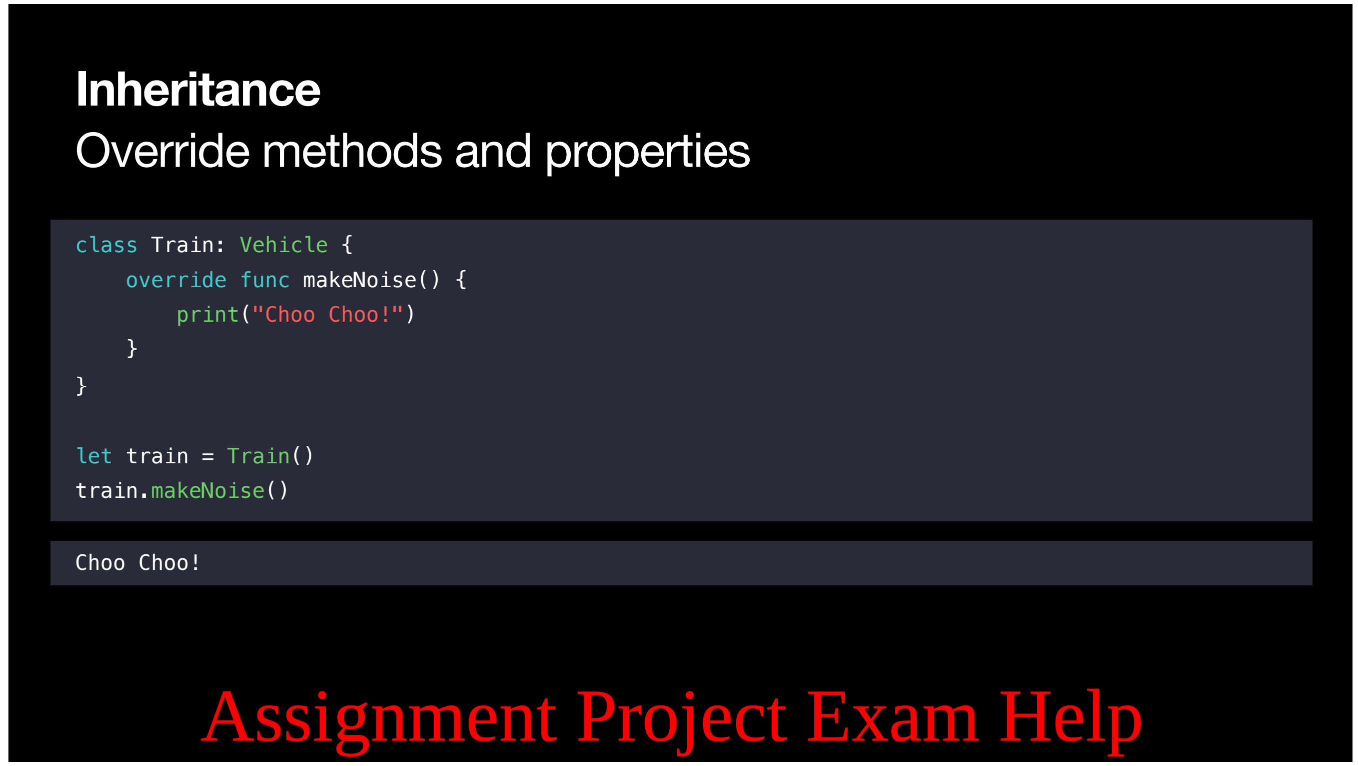
Note

- This slide shows TandemBike as a subclass of `Bicycle`.
- Side note: Classes DO get an empty init() if the class has default values for all properties and you don't write any other inits (and the "write your extra inits in an extension and you still get the compiler-generated ones probably applies to classes).
- Side note: Classes get deinit. Value types don’t.



Note

- This slide shows:
  - How to make an instance of `Tandem`
  - Setting properties of all levels of the hierarchy



Say

- This slide shows `Train` overriding `makeNoise()`.
- The `override` keyword is required for Swift coding safety.
- You can immediately tell that code is overriding something.
- You can't override something by mistake—the compiler will complain.

Inheritance

Override methods and properties

```swift
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
```


Say

- This slide shows `Car` overriding `description`.
- Properties can also be overridden with a getter.

Inheritance

Override methods and properties

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}

let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
```

```
Car: traveling at 25.0 miles per hour in gear 3
```


Note

- This slide shows using `Car` and calling `description`.

Inheritance

Override initializer

```
class Person {
  let name: String

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

class Student: Person {
  var favoriteSubject: String
}
```

Class ‘Student’ has no initializers

Do

- Click to display the error.

Note

- The fix is on the next slide.

Inheritance

Override initializer

```
class Person {
  let name: String

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

class Student: Person {
  var favoriteSubject: String
  init(name: String, favoriteSubject: String) {
    self.favoriteSubject = favoriteSubject
    super.init(name: name)
  }
}
```

References

- When you create an instance of a class:
  - Swift returns the address of that instance
  - The returned address is assigned to the variable
- When you assign the address of an instance to multiple variables:
  - Each variable contains the same address
  - Update one instance, and all variables refer to the updated instance

```
class Person {
  let name: String
  var age: Int

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

var jack = Person(name: "Jack", age: 24)
var myFriend = jack

jack.age += 1

print(jack.age)
print(myFriend.age)

25
25
```


Note

- This slide shows two references to the same instance of a `Person` class.
- It also shows that if you change a property, both references reflect that.

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

var jack = Person(name: "Jack", age: 24)
var myFriend = jack

jack.age += 1

print(jack.age)
print(myFriend.age)


25
24
```


Say

- This slide shows:
  - The same code implemented as a `Person` struct
  - That changing the value of a property of one instance doesn't change the other instance
- That's because `myFriend` is a copy of `jack`.

Memberwise initializers

- Swift does not create memberwise initializers for classes
- Common practice is for developers to create their own for their defined classes

Say

- Classes don't get memberwise initializers.
- Classes DO get an empty `init()` if the class has default values for all properties and you don't write any other inits.

Class or structure?

- Start new types as structures
- Use a class:
  - When you're working with a framework that uses classes
  - When you want to refer to the same instance of a type in multiple places
  - When you want to model inheritance

Unit 2, Lesson 4

Lab: Classes.playground

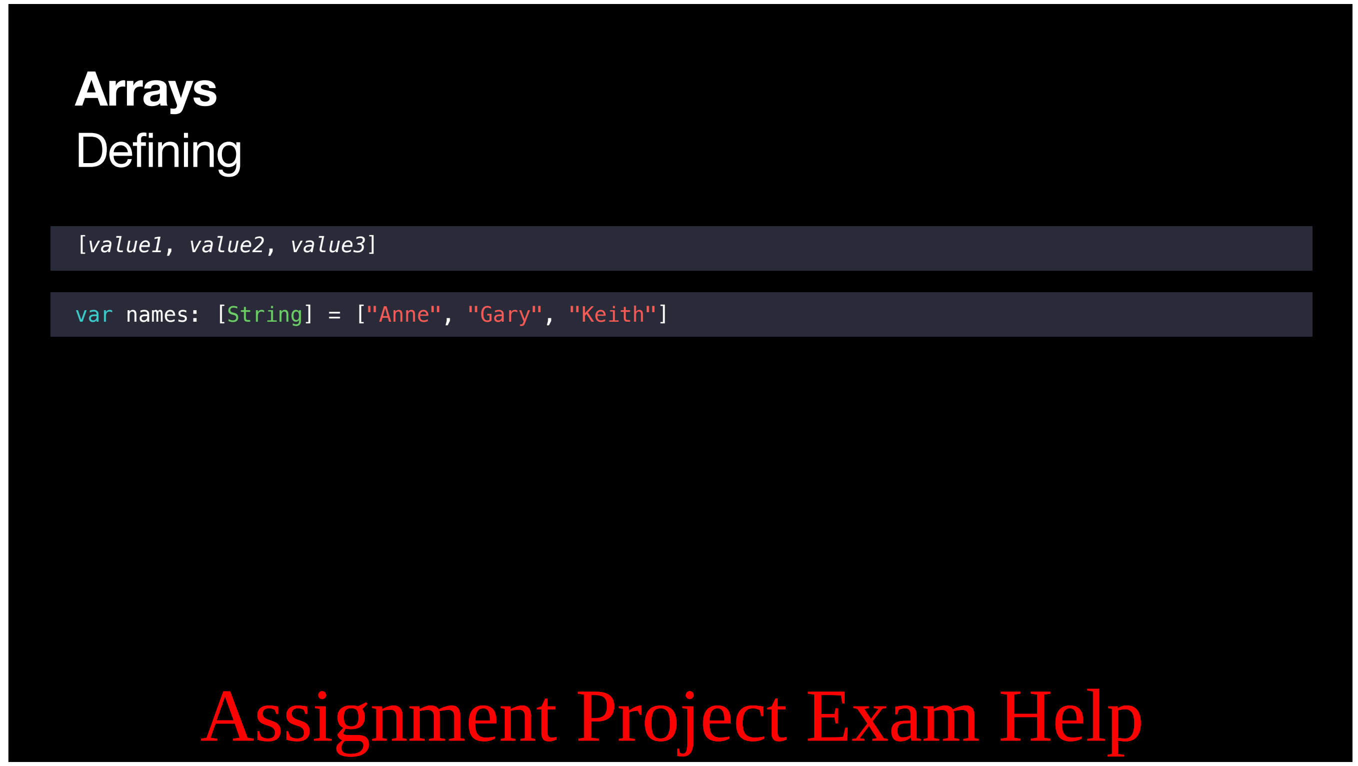
Open and complete the exercises in `Lab - Classes.playground`

Unit 2—Lesson 5: Collections

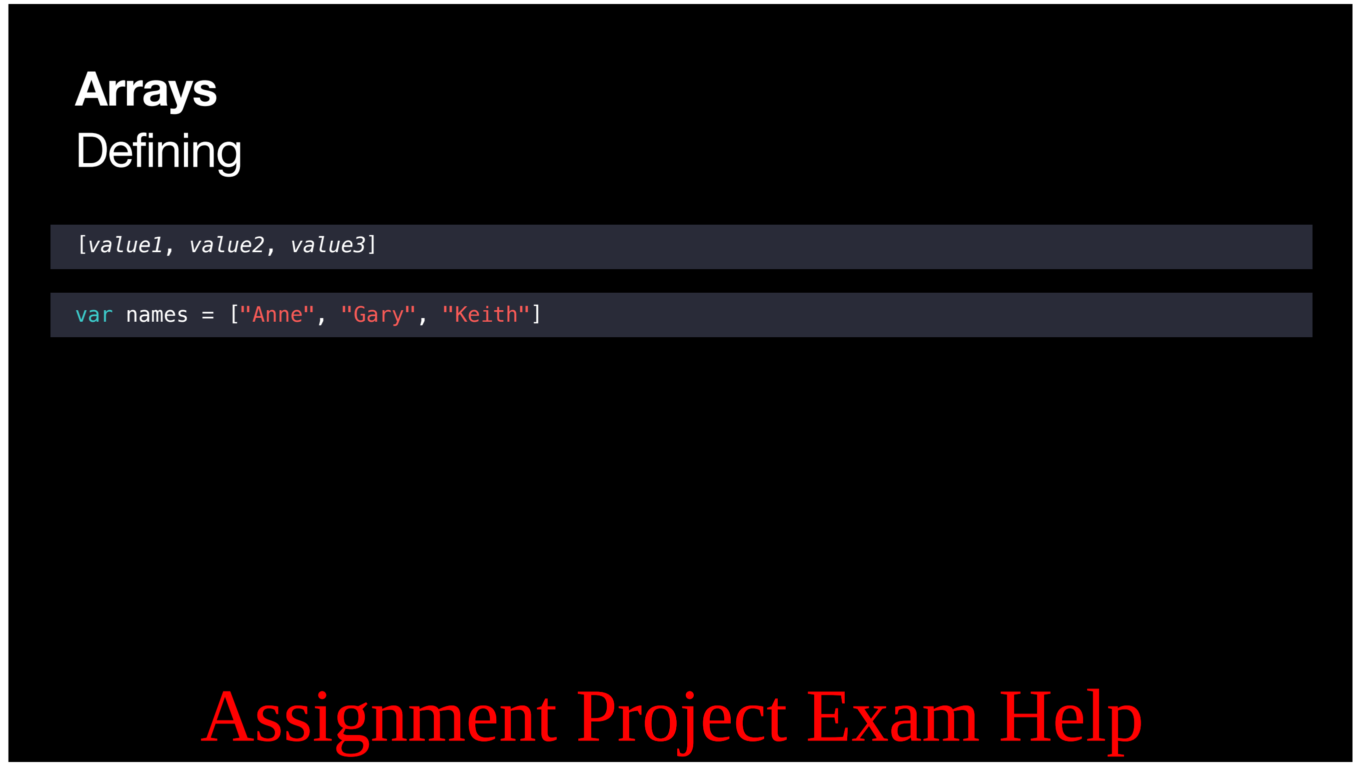
Collection types

Array

Dictionary



Note
• Shows the syntax: [value1, value2, value3, …].
Say
• True or false: Swift should be able to infer the "[String]" part.
• Answer: True (as shown on the next slide).



Say
• The compiler can do type inference for collections, too.
Note
• Shows var names as type inferred to [String].

Arrays

Defining

```
var numbers = [1, -3, 50, 72, -95, 115]
```


Say

- The var numbers = [1, -3, 50] would be [Int].
- What if you want to restrict the array to Int8? Just specify it.

Arrays

Defining

```
var numbers: [Int8] = [1, -3, 50, 72, -95, 115]
```

Arrays
contains

```
let numbers = [4, 5, 6]
if numbers.contains(5) {
  print("There is a 5")
}
```

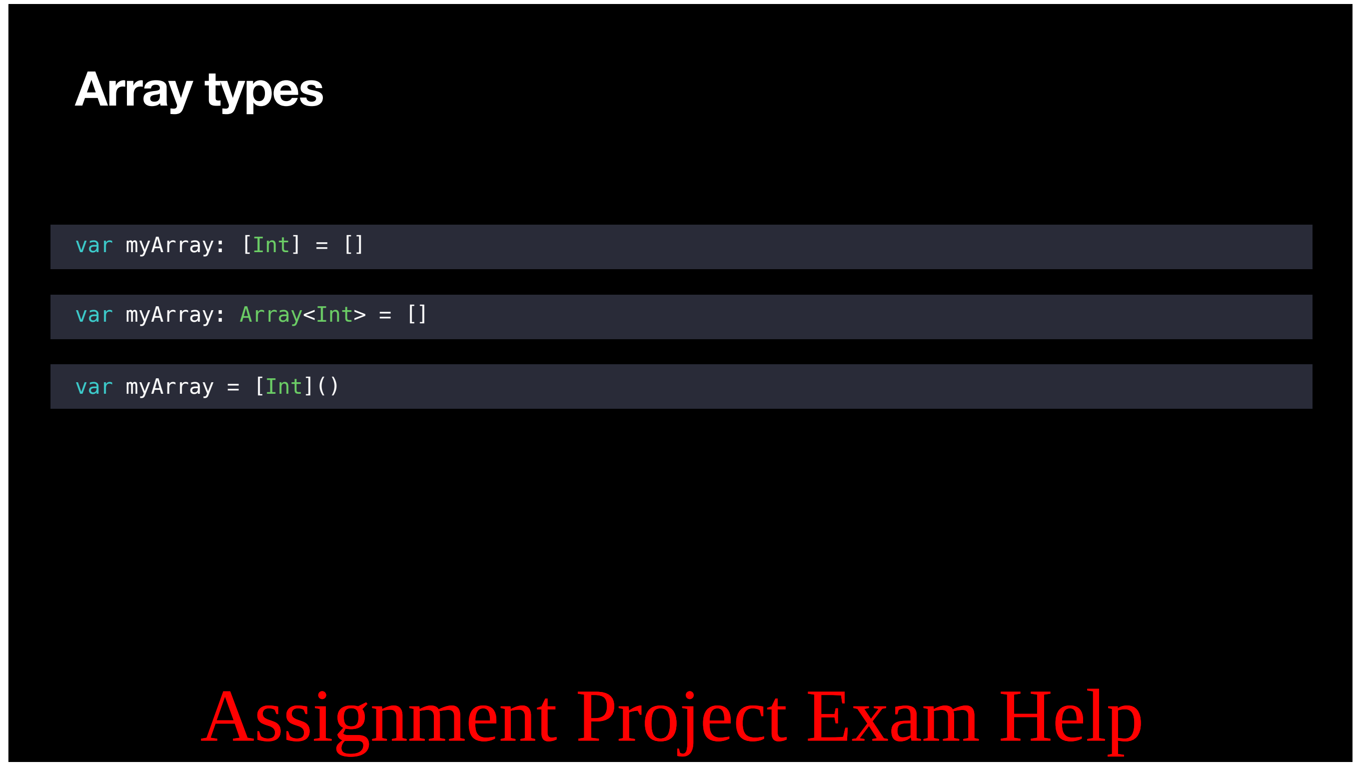
```
There is a 5
```


Say

• someArray.contains(someInstance) returns true if true.

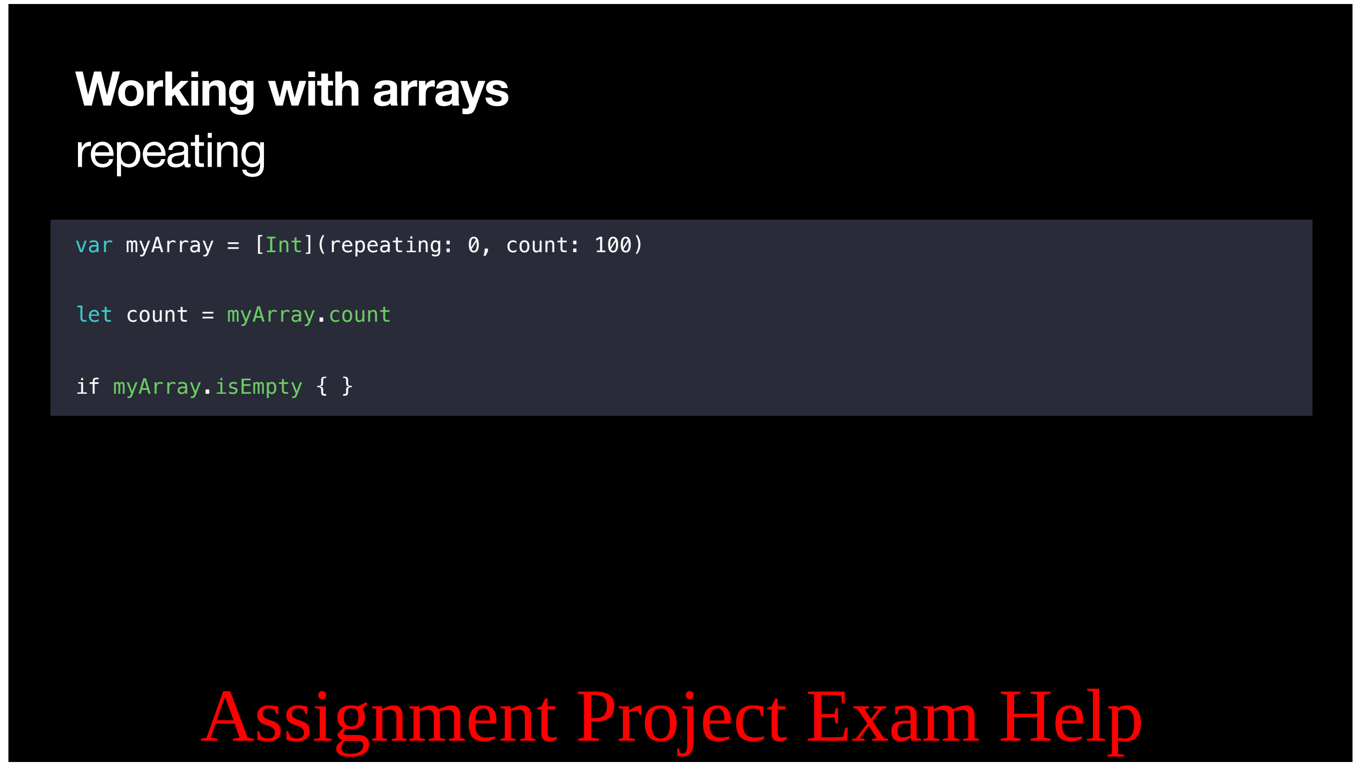
Note

• Contains uses == (the Equatable protocol).



Note

- Shows the many ways to declare the type of things in an array.



Note

- Shows the repeating Array initializer:
  var myArray = [Int](repeating: 0, count: 100)
  Creates an array filled with 100 zeros.

- Shows someArray.count (returns Int).
- Shows someArray.isEmpty (returns Bool).

Working with arrays

Accessing or setting a specific item

```
var names = ["Anne", "Gary", "Keith"]
let firstName = names[0]
print(firstName)
```

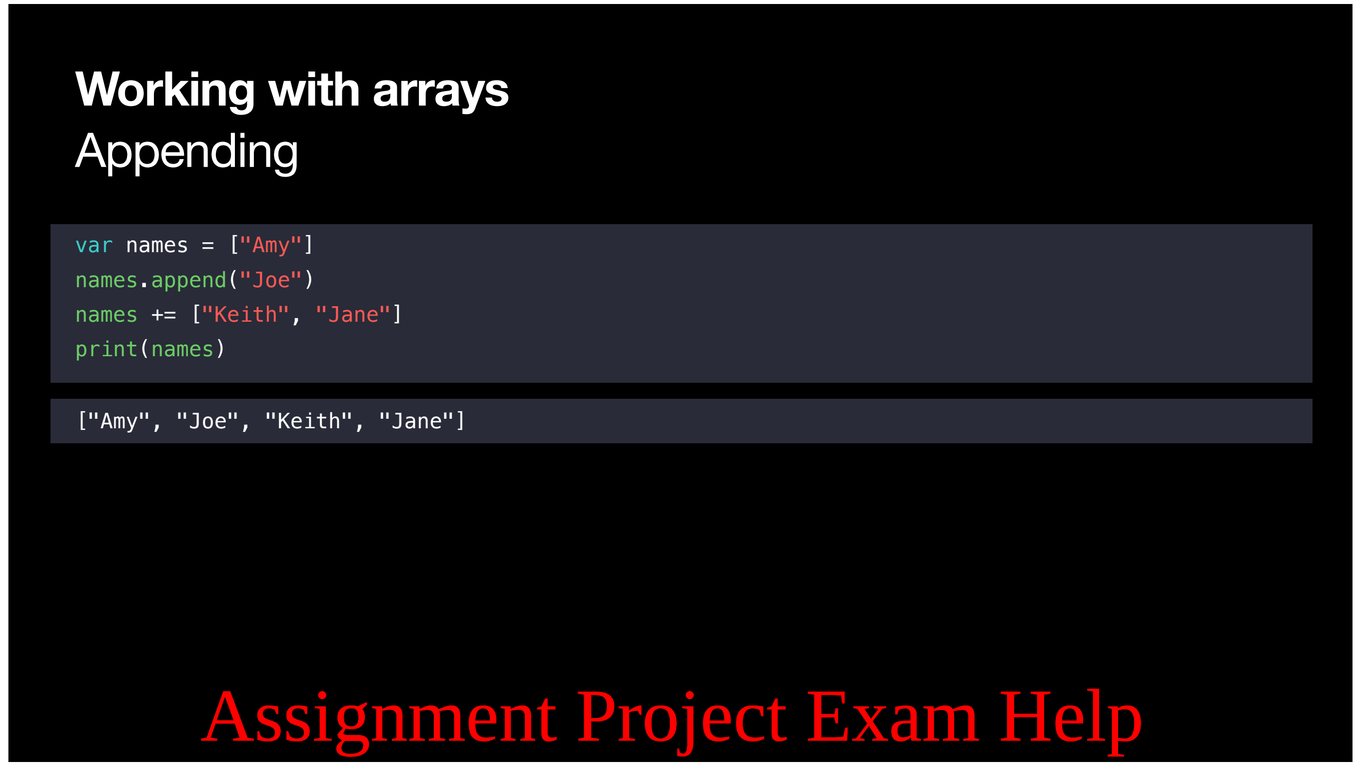
```
Anne
```

```
names[1] = "Paul"
print(names)
```

```
["Anne", "Paul", "Keith"]
```

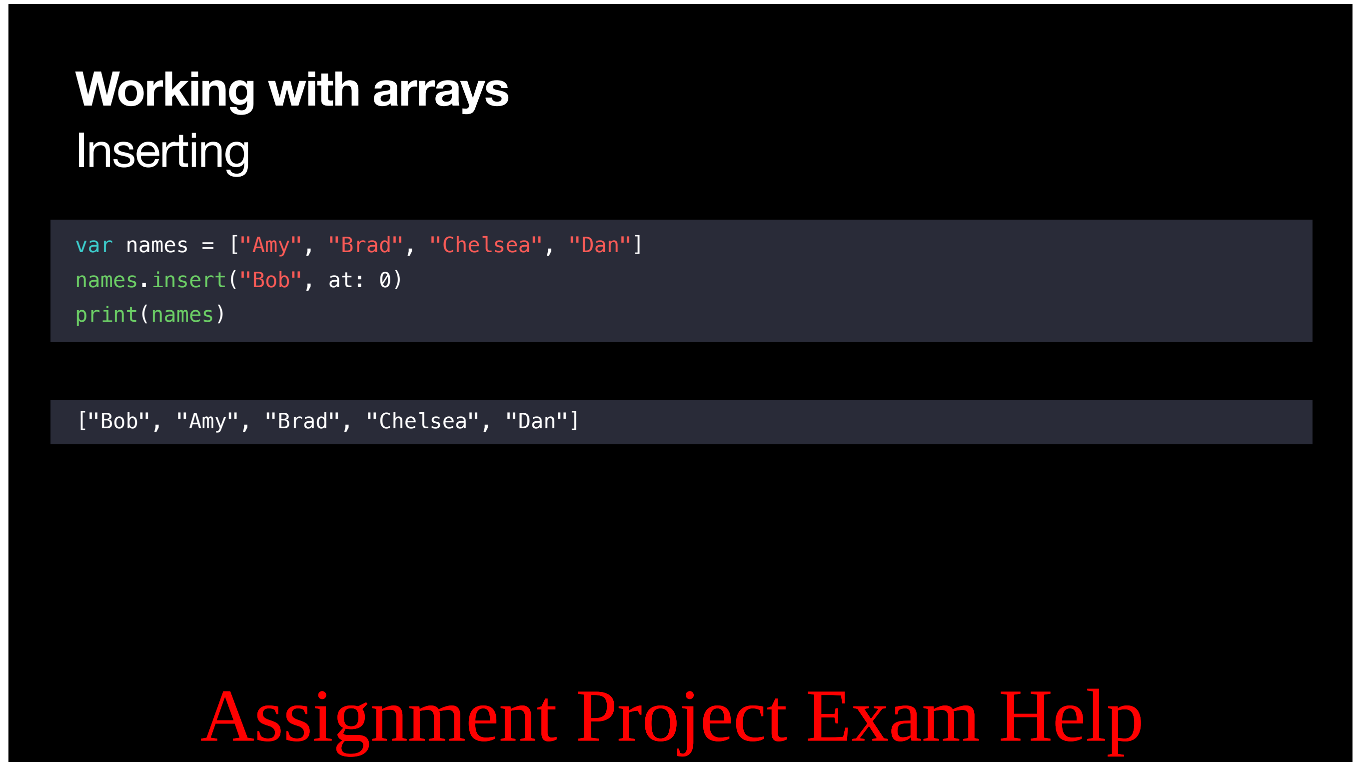

Say

• Change or retrieve a value by using array subscripting syntax which is similar to other languages.



Note

- Shows someArray.append("object").
- append() is limited to 1 item.
- append from array is: someArray.append(contentsOf: someOtherArray)
- Shows someArray += ["Keith", "Jane"].



Note

- Shows someArray.insert("object" at: 0).

Say

- Note that there's also someArray.insert(contentsOf: someOtherArray at:).

Working with arrays
Removing

```swift
var names = ["Amy", "Brad", "Chelsea", "Dan"]
let chelsea = names.remove(at:2)
let dan = names.removeLast()
print(names)
```

```
["Amy", "Brad"]
```

```swift
names.removeAll()
print(names)
```

```
[]
```


Say

- Note that remove(at:) returns the removed item.

Note

- There's also removeFirst() and removeSubrange().

Working with arrays

```
var myNewArray = firstArray + secondArray
```

Working with arrays

Arrays within arrays

```
let array1 = [1,2,3]
let array2 = [4,5,6]
let containerArray = [array1, array2]
let firstArray = containerArray[0]
let firstElement = containerArray[0][0]
print(containerArray)
print(firstArray)
print(firstElement)
```

```
[[1, 2, 3], [4, 5, 6]]
[1, 2, 3]
1
```

Dictionaries

```
[key1 : value1, key2: value2, key3: value3]
```

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]
```

```
var myDictionary = [String: Int]()
var myDictionary = Dictionary<String, Int>()
var myDictionary: [String: Int] = [:]
```


Note

Shows literal syntax and ways to instantiate.

Add/remove/modify a dictionary

Adding or modifying

```swift
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

scores["Oli"] = 399

let oldValue = scores.updateValue(100, forKey: "Richard")
```


Say
- scores["oli"] = 399 will update or insert a value for "oli".
- scores.updateValue(100, forKey: "Richard")
  - Returns old value if there is one.
  - Returns nil if not.

Add/remove/modify a dictionary

Adding or modifying

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

scores["Oli"] = 399

if let oldValue = scores.updateValue(100, forKey: "Richard") {
  print("Richard's old value was \(oldValue)")
}
```

```
Richard's old value was 500
```


Say

- Use if-let to run code only if a value was returned.
- If there wasn't an existing value, the code with the brackets won't be executed.

Add/remove/modify a dictionary

Removing

```swift
var scores = ["Richard": 100, "Luke": 400, "Cheryl": 800]
scores["Richard"] = nil
print(scores)

if let oldValue = scores.removeValue(forKey: "Luke") {
  print("Luke's score was \(oldValue) before he stopped playing")
}
print(scores)
```

```
["Cheryl": 800, "Luke": 400]
Luke's score was 400 before he stopped playing
["Cheryl": 800]
```


Say

- scores["oli"] = nil removes it if present.

  scores.removeValue(forKey: "oli")
- Returns old value if there is one.
- Returns nil if not.

Accessing a dictionary

```swift
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

let players = Array(scores.keys) //["Richard", "Luke", "Cheryl"]
let points = Array(scores.values) //[500, 400, 800]

if let myScore = scores["Luke"] {
  print(myScore)
}
```

```
400
```

```swift
if let henrysScore = scores["Henry"] {
  print(henrysScore)
}
```


Say
- What's the return type?
- Answer: Int? because there may not be a key/value pair for the key.

Unit 2—Lesson 5

Lab: Collections

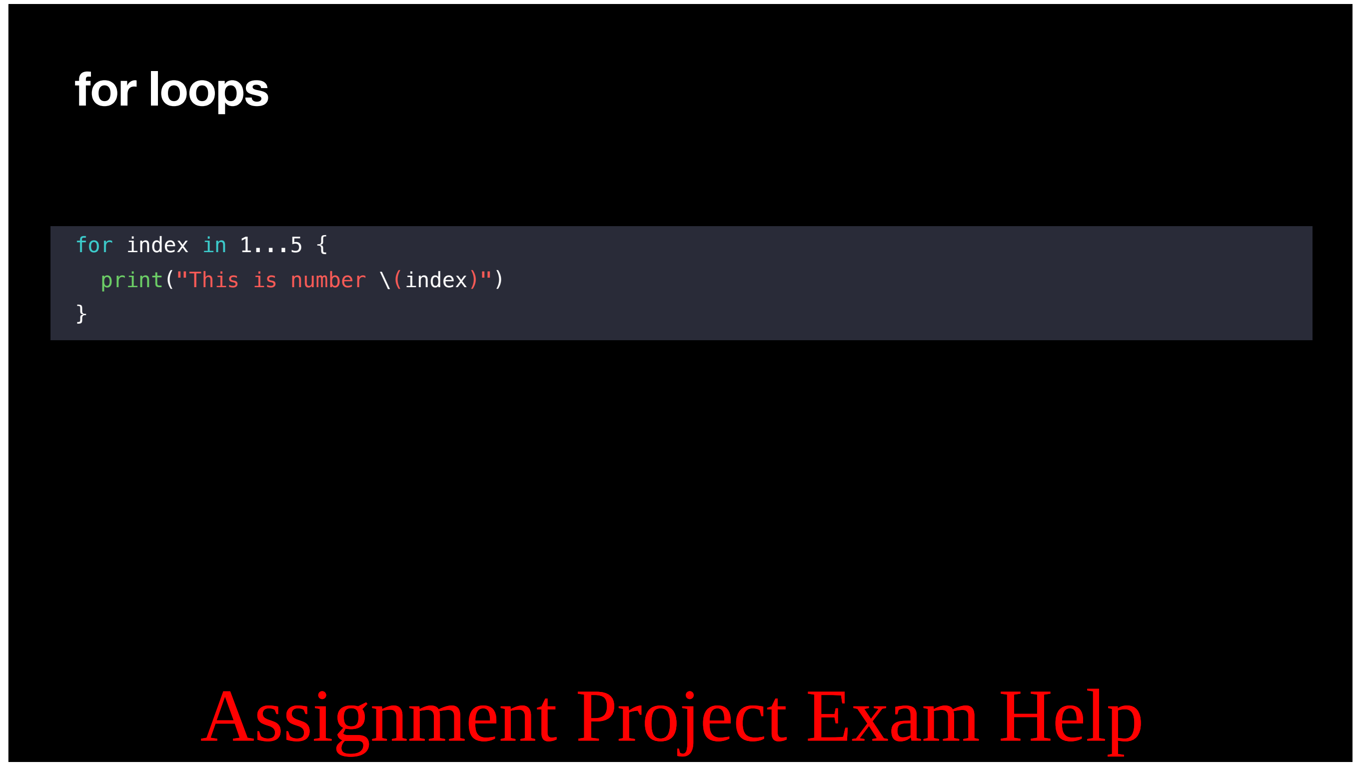
Open and complete the exercises in `Lab - Collections.playground`

Unit 2—Lesson 6: Loops

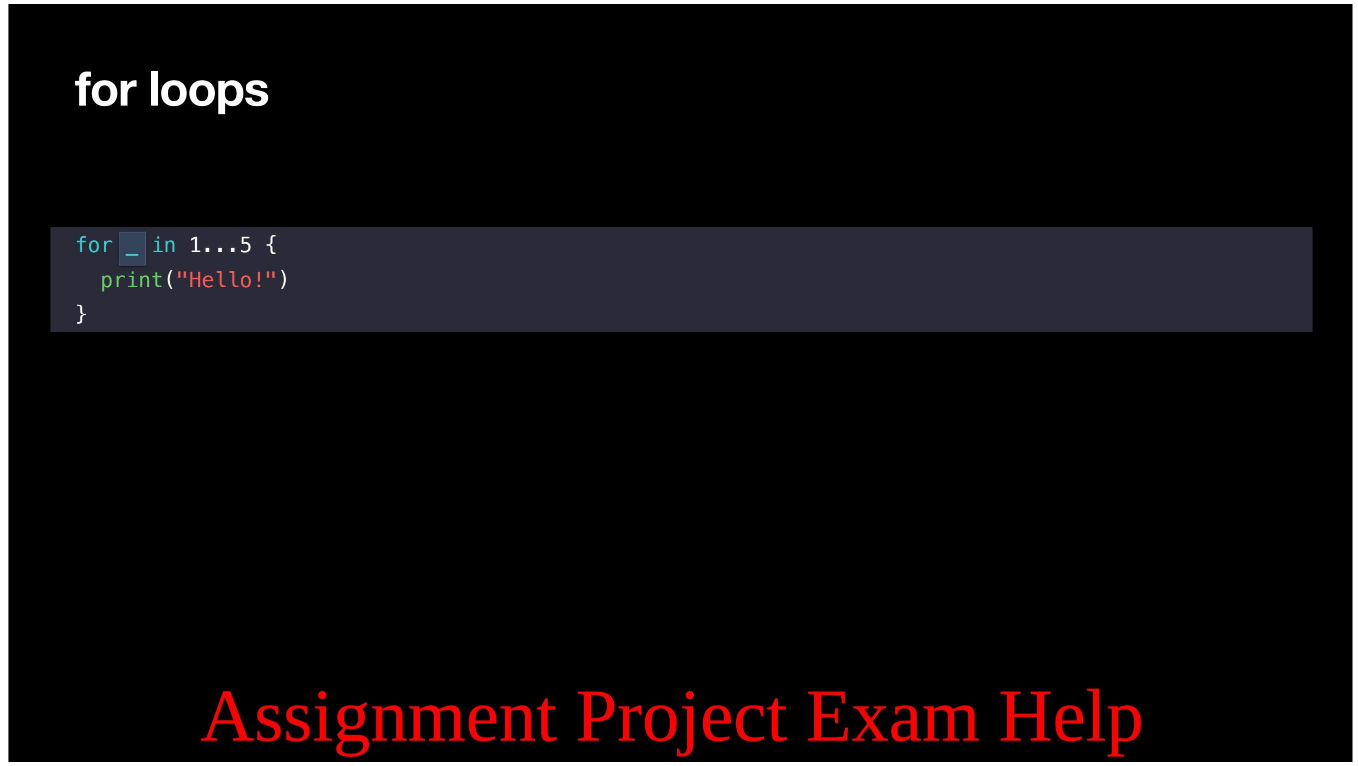
Loops

```
for
while
```


Say
- Swift provides several different methods for looping.
- In this lesson we'll focus on just two: for and while.



Say
- "for" loops work with ranges—it has a nice, concise syntax:
  for index in 1...5 {}



Do
• Click to highlight the underscore "_".
Say
• Use the underscore if you don't need to use the index value.

for loops

```swift
let names = ["Joseph", "Cathy", "Winston"]
for name in names {
  print("Hello \(name)")
}
```

```swift
for letter in "ABCDEFG".characters {
  print("The letter is \(letter)")
}
```


Do
• Click to highlight "names" as the range being iterated over.
Say
• This is often used to iterate over a collection.
Do
• Click again to highlight a string as a range of characters.

for loops

```swift
for (index, letter) in "ABCDEFG".characters.enumerated() {
  print("\(index): \(letter)")
}
```


Say

• enumerated() gives you the item and the index.

Note

• (index, letter) is a tuple.

for loops

```swift
let vehicles = ["unicycle" : 1, "bicycle" : 2, "tricycle" : 3, "quad bike" : 4]
for (vehicleName, wheelCount) in vehicles {
  print("A \(vehicleName) has \(wheelCount) wheels")
}
```


Say
- Iterating over a Dictionary also gives you a tuple.
- It’s unordered because Dictionary is unordered.

while loops

```
var numberOfLives = 3

while numberOfLives > 0 {
  playMove()
  updateLivesCount()
}
```


Say

- Condition is at the top. The loop may happen zero times.

while loops

```
var numberOfLives = 3

while numberOfLives > 0 {
  print("I still have \(numberOfLives) lives.")
}
```


Say

• The code inside the loop needs to eventually make the condition false.

while loops

```
var numberOfLives = 3
var stillAlive = true

while stillAlive {
  print("I still have \(numberOfLives) lives.")
  numberOfLives -= 1
  if numberOfLives == 0 {
    stillAlive = false
  }
}
```

Control transfer statements

```
for counter in -10...10 {
  print(counter)
  if counter == 0 {
    break
  }
}
```

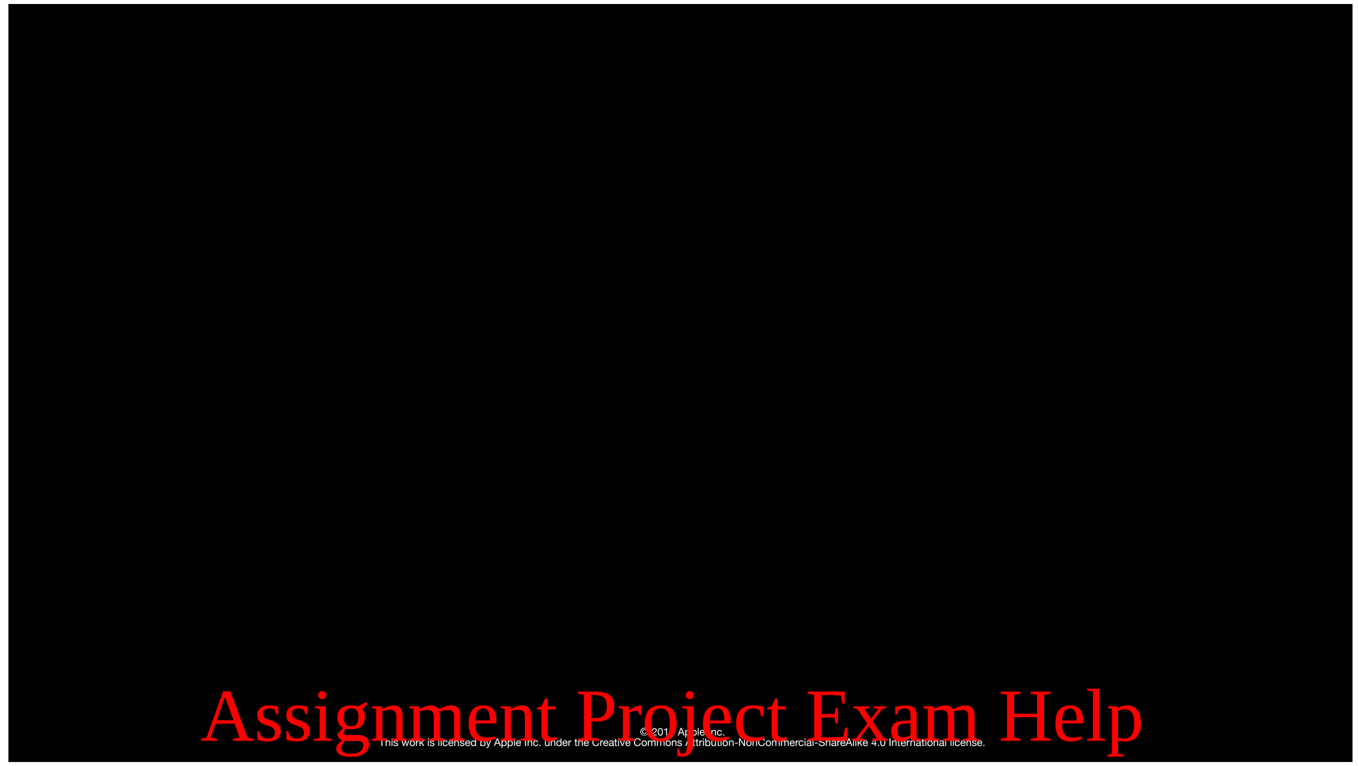
```
-10
-9
...
0
```


Note
• Shows that "break" exits a loop.

Unit 2—Lesson 6

Lab: Loops

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



© 2017 Apple Inc.
This work is licensed by Apple Inc. under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license.