Lab 0 — https://github.com/williams-cs/cs326_lab0__stephenfreund.git

commit fb073994808f5457656ec2a8e12cf276203df722
Author: freund <freund@cs.williams.edu>
Date: Thu Aug 23 17:54:35 2018 -0400

    Initial Commit

Prepared By: freund at Mon Sep 3 07:21:13 EDT 2018
CS326 Markdown Example

This is the raw markdown used to generate some sample HTML.

Bulleted List

- Moo
- Cow

Numbered List

1. Moo
2. Cow

Formatting

Can be **bold** or *italic*

Code

You can have short fragments in line or longer multi-line fragments:

```python
let x = "moo"
print(x)
```

Links

Click here to go somewhere.
## Objects and Classes

Use `class` followed by the class's name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it is in the context of a class. Likewise, method and function declarations are written the same way.

```swift
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

- Experiment:
  - Add a constant property with `let`, and add another method that takes an argument.
  - Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

```swift
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

- This version of the `Shape` class is missing something important: an initializer to set up the class when an instance is created. Use `init` to create one.

```swift
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

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

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

- Notice how `self` is used to distinguish the `name` property from the `name` argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned either in its declaration (as with `numberOfSides`) or in the initializer (as with `name`).

- Use `deinit` to create a deinitializer if you need to perform some cleanup before the object is deallocated.

- Subclasses include their superclass name after their class name, separated by a colon. There is no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

- Methods on a subclass that override the superclass's implementation are marked with `override`. Overriding a method by accident, without `override`, is detected by the compiler as an error. The compiler also detects methods with `override` that don't actually override any method in the superclass.

```swift
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() -> Double {
        return sideLength * sideLength
    }
}
```
let test = Square(sideLength: 5.2, name: "my test square")

print(test.area())

print(test.simpleDescription())

// - Experiment:
// Make another subclass of 'NamedShape' called 'Circle' that takes a radius and a name
// as arguments to its initializer. Implement an 'area()' and a 'simpleDescription()' method
// on the 'Circle' class.
// In addition to simple properties that are stored, properties can have a getter and a
// setter.

class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
        get {
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \(sideLength)."
    }
}

var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
triangle.perimeter = 9.9
print(triangle.sideLength)

// Notice that the initializer for the 'EquilateralTriangle' class has three different s
// teps:
// 1. Setting the value of properties that the subclass declares.
// 1. Calling the superclass's initializer.
// 1. Changing the value of properties defined by the superclass. Any additional setup w
// work that uses methods, getters, or setters can also be done at this point.

class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            .....
        }
    }
}
triangle.sideLength = newValue.sideLength
}
}
init(size: Double, name: String) {
    square = Square(sideLength: size, name: name)
    triangle = EquilateralTriangle(sideLength: size, name: name)
}
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
print(triangleAndSquare.square.sideLength)
print(triangleAndSquare.triangle.sideLength)
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)

//: When working with optional values, you can write `?' before operations like methods, properties, and subscripting. If the value before the `?' is `nil`, everything after the `?' is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?' acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.
let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")
let sideLength = optionalSquare?.sideLength

//: [Previous](@previous) | [Next](@next)
## Enumerations and Structures

Use 'enum' to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```swift
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king

    func simpleDescription() -> String {
        switch self {
        case .ace:
            return "ace"
        case .jack:
            return "jack"
        case .queen:
            return "queen"
        case .king:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
```

```swift
define ace = Rank.ace
let aceRawValue = ace.rawValue
```

- **Experiment:**
  - Write a function that compares two 'Rank' values by comparing their raw values.
  - By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values. In the example above, 'Ace' is explicitly given a raw value of '1', and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the 'rawValue' property to access the raw value of an enumeration case.
  - Use the 'init?(rawValue:)' initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or 'nil' if there is no matching 'Rank'.

```swift
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

- **Experiment:**
  - Add a 'color()' method to 'Suit' that returns 'black' for spades and clubs, and returns 'red' for hearts and diamonds.

```swift
enum Suit {
    case spades, hearts, diamonds, clubs

    func simpleDescription() -> String {
        switch self {
        case .spades:
            return "spades"
        case .hearts:
            return "hearts"
        case .diamonds:
            return "diamonds"
        case .clubs:
            return "clubs"
        }
    }
}
```

```swift
define hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
```

- **Experiment:**
  - Add a 'color()' method to 'Suit' that returns \black\200\234\200\235 for spades and clubs, and returns \red\200\234\200\235 for hearts and diamonds.
//: Notice the two ways that the 'hearts' case of the enumeration is referred to above: When assigning a value to the 'hearts' constant, the enumeration case 'Suit.hearts' is referred to by its full name because the constant doesn't have an explicit type specified. Inside the switch, the enumeration case is referred to by the abbreviated form '.hearts' because the value of 'self' is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

//: If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value. Another choice for enumeration cases is to have values associated with the case—these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

//: enum ServerResponse {
    case result(String, String)
    case failure(String)
}

let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

switch success {
    case let .result(sunrise, sunset):
        print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
    case let .failure(message):
        print("Failure... \(message)")
}

//: - Experiment:
//: Add a third case to 'ServerResponse' and to the switch.

//: Notice how the sunrise and sunset times are extracted from the 'ServerResponse' value as part of matching the value against the switch cases.

//: Use 'struct' to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they are passed around in your code, but classes are passed by reference.

//: struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}

let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
## Error Handling

You represent errors using any type that adopts the 'Error' protocol.

```swift
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

Use 'throw' to throw an error and 'throws' to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```swift
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

There are several ways to handle errors. One way is to use 'do'-'catch'. Inside the 'do' block, you mark code that can throw an error by writing 'try' in front of it. Inside the 'catch' block, the error is automatically given the name 'error' unless you give it a different name.

```swift
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch {
    print(error)
}
```

There are several ways to handle errors. One way is to use 'do'-'catch'. Inside the 'do' block, you mark code that can throw an error by writing 'try' in front of it. Inside the 'catch' block, the error is automatically given the name 'error' unless you give it a different name.

```swift
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
```

Another way to handle errors is to use 'try?' to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is 'nil'. Otherwise, the result is an optional containing the value that the function returned.

```swift
let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")
```

Use 'defer' to write a block of code that is executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use 'defer' to write setup and cleanup code next to each other, even though they need to be executed at different times.

```swift
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]
```
func fridgeContains(_ food: String) -> Bool {
    fridgeIsOpen = true
    defer {
        fridgeIsOpen = false
    }
    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
print(fridgeIsOpen)
//: # A Swift Tour

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

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

//: If you have written code in C or Objective-C, this syntax looks familiar to you. In Swift, this line of code is a complete program. You don’t need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don’t need a `main()` function. You also don’t need to write semicolons at the end of every statement.

//: This tour gives you enough information to start writing code in Swift by showing you how to accomplish a variety of programming tasks. Don’t worry if you don’t understand something—everything introduced in this tour is explained in detail in the rest of this book.

//: ## Simple Values

//: Use `let` to make a constant and `var` to make a variable. The value of a constant doesn’t need to be known at compile time, but you must assign it a value exactly once. This means you can use constants to name a value that you determine once but use in many places.

```swift
var myVariable = 42
myVariable = 50
let myConstant = 42
```

//: A constant or variable must have the same type as the value you want to assign to it. However, you don’t always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that `myVariable` is an integer because its initial value is an integer.

//: If the initial value doesn’t provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

```swift
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

//: - Experiment:

//: Create a constant with an explicit type of `Float` and a value of `4`.

//: Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

```swift
let label = "The width is 
let width = 94
let widthLabel = label + String(width)
```

//: - Experiment:

//: Try removing the conversion to `String` from the last line. What error do you get?

//: There’s an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (`\`) before the parentheses. For example:

```swift
let apples = 3
let oranges = 5
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

//: - Experiment:

//: Use `\()` to include a floating-point calculation in a string and to include someone’s name in a greeting.

//: Use three double quotes (""") for strings that take up multiple lines. Indentation
at the start of each quoted line is removed, as long as it matches the indentation of the closing quote. For example:

```swift
let quotation = ""
I said "I have \(apples) apples."
And then I said "I have \(apples + oranges) pieces of fruit."
""
```

// Create arrays and dictionaries using brackets ('[]'), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

```swift
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"

var occupations = {
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
}
occupations["Jayne"] = "Public Relations"
```

// To create an empty array or dictionary, use the initializer syntax.

```swift
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

// If type information can be inferred, you can write an empty array as '[]' and an empty dictionary as '[:]' for example, when you set a new value for a variable or pass an argument to a function.

```swift
shoppingList = []
occupations = [:]
```

// See [License](License) for this sample’s licensing information.

//: [Next](@next)
//: ## Protocols and Extensions

//: Use 'protocol' to declare a protocol.

```swift
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

//: Classes, enumerations, and structs can all adopt protocols.

```swift
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += " Now 100% adjusted."
    }
}
```

```swift
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription
```

```swift
struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
```

```swift
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

//: - Experiment:

//: Write an enumeration that conforms to this protocol.

```swift
//: Notice the use of the 'mutating' keyword in the declaration of 'SimpleStructure' to mark a method that modifies the structure. The declaration of 'SimpleClass' doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

//: Use 'extension' to add functionality to an existing type, such as new methods and computed properties. You can use an extension to add protocol conformance to a type that is declared elsewhere, or even to a type that you imported from a library or framework.

extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
```

```swift
print(7.simpleDescription)
```

//: - Experiment:

//: Write an extension for the 'Double' type that adds an 'absoluteValue' property.

```swift
//: You can use a protocol name just like any other named type for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a protocol type, methods outside the protocol definition are not available.

extension ExampleProtocol {
    var absoluteValue: Double {
        return self
    }
}
```

```swift
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription) // Uncomment to see the error
```

//: Even though the variable 'protocolValue' has a runtime type of 'SimpleClass', the compiler treats it as the given type of 'ExampleProtocol'. This means that you can't accidentally access methods or properties that the class implements in addition to its protocol conformance.
Use 'func' to declare a function. Call a function by following its name with a list of arguments in parentheses. Use '-' to separate the parameter names and types from the function's return type.

```swift
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

- Experiment:
- Remove the 'day' parameter. Add a parameter to include today's lunch special in the greeting.
- By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write '_' to use no argument label.

```swift
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

Use a tuple to make a compound value for example, to return multiple values from a function. The elements of a tuple can be referred to either by name or by number.

```swift
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0
    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }
    return (min, max, sum)
}
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
print(statistics.2)
```

Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

```swift
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

Functions are a first-class type. This means that a function can return another function as its value.

```swift
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
A function can take another function as one of its arguments.

```
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}

func lessThanTen(number: Int) -> Bool {
    return number < 10
}

var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed. You saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces (`{}`). Use `in` to separate the arguments and return type from the body.

```
numbers.map({ (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

- Experiment:
  Rewrite the closure to return zero for all odd numbers.

```
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
```

You can refer to parameters by number instead of by name. This approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
```
# License

IMPORTANT: This Apple software is supplied to you by Apple Inc. ("Apple") in consideration of your agreement to the following terms, and your use, installation, modification or redistribution of this Apple software constitutes acceptance of these terms. If you do not agree with these terms, please do not use, install, modify or redistribute this Apple software.

In consideration of your agreement to abide by the following terms, and subject to these terms, Apple grants you a personal, non-exclusive license, under Apple’s copyrights in this original Apple software (the "Apple Software"), to use, reproduce, modify and redistribute the Apple Software, with or without modifications, in source and/or binary forms; provided that if you redistribute the Apple Software in its entirety and without modifications, you must retain this notice and the following text and disclaimers in all such redistributions of the Apple Software. Neither the name, trademarks, service marks or logos of Apple Inc. may be used to endorse or promote products derived from the Apple Software without specific prior written permission from Apple. Except as expressly stated in this notice, no other rights or licenses, express or implied, are granted by Apple herein, including but not limited to any patent rights that may be infringed by your derivative works or by other works in which the Apple Software may be incorporated.

The Apple Software is provided by Apple on an "AS IS" basis. APPLE MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE APPLE SOFTWARE OR ITS USE AND OPERATION ALONE OR IN COMBINATION WITH YOUR PRODUCTS.

IN NO EVENT SHALL APPLE BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) ARISING IN ANY WAY OUT OF THE USE, REPRODUCTION, MODIFICATION AND/OR DISTRIBUTION OF THE APPLE SOFTWARE, HOWEVER CAUSED AND WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY OR OTHERWISE, EVEN IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (C) 2015 Apple Inc. All Rights Reserved.
## Generics

Write a name inside angle brackets to make a generic function or type.

```swift
func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
makeArray(repeating: "knock", numberOfTimes: 4)
```

You can make generic forms of functions and methods, as well as classes, enumerations, and structures.

Reimplement the Swift standard library’s optional type

```swift
enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
}
```

```swift
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)
```

Use ‘where’ right before the body to specify a list of requirements, for example, to require the type to implement a protocol, to require two types to be the same, or to require a class to have a particular superclass.

```swift
func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
    return false
}
```

```swift
anyCommonElements([1, 2, 3], [3])
```

Experiment:

Modify the `anyCommonElements(_:_:)` function to make a function that returns an array of the elements that any two sequences have in common.

```swift
func anyCommonElements<T: Equatable>(_ lhs: T, _ rhs: T) -> [T]
    var result: [T] = []
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                result.append(lhsItem)
            }
        }
    }
    return result
}
anyCommonElements([1, 2, 3], [3])
```

Writing ‘<T: Equatable>‘ is the same as writing ‘<T> ... where T: Equatable’.

[Previous](@previous) | [Next](@next)
let names = ["Wally", "Abba", "Prairie", "Wally", "Radish", "Springer"]

func printIndex(name: String) {
    printIndex(name: "Wally")
    printIndex(name: "Coconut")
}

func printIndex2(name: String) {
    printIndex2(name: "Wally")
    printIndex2(name: "Coconut")
}
## Control Flow

Use `if` and `switch` to make conditionals, and use `for`-'in', `while`, and `repeat`-'while' to make loops. Parentheses around the condition or loop variable are optional. Braces around the body are required.

```swift
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
```

In an `if` statement, the conditional must be a Boolean expression: this means that code such as `if score { ... }` is an error, not an implicit comparison to zero.

You can use `if` and `let` together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains `nil` to indicate that a value is missing. Write a question mark (`?`) after the type of a value to mark the value as optional.

```swift
var optionalString: String? = "Hello"
prompt(optionalString == nil)
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

Change `optionalName` to `nil`. What greeting do you get? Add an `else` clause that sets a different greeting if `optionalName` is `nil`.

If the optional value is `nil`, the conditional is `false` and the code in braces is skipped. Otherwise, the optional value is unwrapped and assigned to the constant after `let`, which makes the unwrapped value available inside the block of code.

Another way to handle optional values is to provide a default value using the `??` operator. If the optional value is missing, the default value is used instead.

```swift
let nickName: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Hi \(nickName ?? fullName)"
```

Switches support any kind of data and a wide variety of comparison operations, not limited to integers and tests for equality.

```swift
let vegetable = "red pepper"
switch vegetable {
    case "celery":
        print("Add some raisins and make ants on a log.")
    case "cucumber", "watercress":
        print("That would make a good tea sandwich.")
    case let x where x.hasSuffix("pepper"):
        print("Is it a spicy \(x)?)")
    default:
        print("Everything tastes good in soup.")
}
```

Try removing the default case. What error do you get?

Notice how `let` can be used in a pattern to assign the value that matched the pattern to a constant.
After executing the code inside the switch case that matched, the program exits from the switch statement. Execution doesn’t continue to the next case, so there is no need to explicitly break out of the switch at the end of each case’s code.

You use ‘for’-‘in’ to iterate over items in a dictionary by providing a pair of names to use for each key-value pair. Dictionaries are an unordered collection, so their keys and values are iterated over in an arbitrary order.

```swift
let interestingNumbers = {
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
}
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
```

- Experiment:

Add another variable to keep track of which kind of number was the largest, as well as what that largest number was.

Use ‘while’ to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```swift
var n = 2
while n < 100 {
    n *= 2
}
print(n)
var m = 2
repeat {
    m *= 2
} while m < 100
print(m)
```

You can keep an index in a loop by using ‘..<’ to make a range of indexes.

```swift
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
```

Use ‘..<’ to make a range that omits its upper value, and use ‘...’ to make a range that includes both values.

```swift
[Previous](@previous) | [Next](@next)
```