Agenda

- Control Flow and Functions
- Classes and Structs

Control Flow and Functions

- What are conditional statements and how to use them
- What are loops and how to use them
- What are control transfer statements and how to use them
- How to create and use functions in Swift

Curly Brackets

```
if (x > y)
        x = 0


if (x > y) {
        x = 0
}
```

Parenthesis

```
if x > y {
        x = 0
}
```

```
if (x > y) {
        x = 0
}
```

Control Flow

- Conditional statements
- For loop
- While loop
- Switch statements
- Case and for/where
- Control transfer statements

Conditional Statements

- if
- if … else

If Statements

```
if condition {
        block of code
}
```

If Statements

```
if condition {
    block of code
}
```

```
let teamOneScore = 7
let teamTwoScore = 6
if teamOneScore > teamTwoScore {
        print("Team One Won")
}
```

If…Else Statements

```
if condition {
        block of code if true
} else {
        block of code if not true
}
```

If…Else Statements

```
var teamOneScore = 7
var teamTwoScore = 6


if teamOneScore > teamTwoScore {
        print("Team One Won")
} else {
        print("Team Two Won")
}
```

If…Else Statements

```
var teamOneScore = 7
var teamTwoScore = 6


if teamOneScore > teamTwoScore {
        print("Team One Won")
} else if teamTwoScore > teamOneScore {
        print("Team Two Won")
} else {
        print("We have a tie")
}
```

For Statements

- for…in Loop

For Statements

```
for variable in Collection/Range {
        block of code
}
```

- Variable
- Collection/Range

For Statements

```
for index in 1...5 {
        print(index)
}
```

For Statements

```
for index in 1..<5 {
        print(index)
}
```

For Statements

```
var countries = ["USA","UK", "IN"]
for item in countries {
        print(item)
}
```

For Statements

```
var dic = ["USA": "United States", "UK": "United Kingdom", "IN":"India"]


for (abbr, name) in dic {
        print("\(abbr) -- \(name)")
}
```

While Statements

- While
- Repeat…while

While Statements

```
while condition {
        block of code
}
```

While Statements

```
var  ran  =  0
while ran < 4 {
        ran  =  Int(arc4random_uniform(100) % (5))
}
```

Repeat … While Statements

```
repeat {
        block of code
} while condition
```

Repeat … While Statements

```
var ran: Int
repeat {
        ran = Int(arc4random_uniform(100) % (5))
} while ran < 4
```

Switch Statement

```
switch value {
        case match1 :
                block of code
        case match2 :
                block of code
        ..... as many cases as needed
        default :
                block of code

}
```

Switch Statement

```
var speed = 300000000


switch speed {
        case 300000000:
                print("Speed of light")
        case 340:
                print("Speed of sound")
        default: print("Unknown speed")
}
```

Switch Statement

```
var num = 5

switch num {
        case 1 :
                print("number is one")
        case 2 :
                print("Number is two")
        case 3 :
                print("Number is three")
}
```

Switch Statement

```
var char : Character = "e"

switch char {
        case "a", "e", "i", "o", "u":
                print("letter is a vowel")
        case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
                print("letter is a consonant")
        default:
                print("unknown letter")
}
```

Switch Statement

```
var grade = 93

switch grade {
        case 90...100:
                print("Grade is an A")
        case 80...89:
                print("Grade is a B")
        case 70...79:
                print("Grade is an C")
        case 60...69:
                print("Grade is a D")
        case 0...59:
                print("Grade is a F")
        default:
                print("Unknown Grade")
}
```

Switch Statement

```
var studentId = 4
var grade = 57

switch grade {
        case 90...100:
                print("Grade is an A")
        case 80...89:
                print("Grade is a B")
        case 70...79:
                print("Grade is an C")
        case 55...69 where studentId == 4
                print("Grade is a D for Student 4")
        case 60...69:
                print("Grade is an D”)
        case 0...59:
                print("Grade is a F")
        default:
                print("Unknown Grade")
}
```

Switch Statement

```
var studentId = 4
var grade = 57

switch grade {
        case 90...100:
                print("Grade is an A")
        case 80...89:
                print("Grade is a B")
        case 70...79:
                print("Grade is an C")
        case 60...69:
                print("Grade is an D”)
        case 55...69 where studentId == 4:
                print("Grade is a D for Student 4")
        case 0...59:
                print("Grade is a F")
        default:
                print("Unknown Grade")
}
```

Switch Statement

```
enum Product {
        case Book(String, Double, Int)
        case Puzzle(String, Double)
}

var order = Product.Book("Mastering Swift 2", 49.99, 2015)

switch order {
        case .Book(let name, let price, let year):
                print("You ordered the book \(name) for \(price)")
        case .Puzzle(let name, let price):
                print("You ordered the Puzzle \(name) for \(price)")
}
```

Using case and where statements with conditional statements

- Filtering with the **where** statement
- Filtering with the **for…case** statement
- Using the **if…case** statement

Filtering with the where statement

```
for number in 1...30 {
        if number % 2 == 0 {
                print(number)
        }
}
```

Filtering with the where statement

```
for number in 1...30 {
        if number % 2 == 0 {
                print(number)
        }
}

for number in 1...30 where number % 2 == 0 {
        print(number)
}
```

Filtering with the case statement

```
var worldSeriesWinners = [
        ("Red Sox", 2004),
        ("White Sox", 2005),
        ("Cardinals", 2006),
        ("Red Sox", 2007),
        ("Phillies", 2008),
        ("Yankees", 2009),
        ("Giants", 2010),
        ("Cardinals", 2011),
        ("Giants", 2012),
        ("Red Sox", 2013),
        ("Giants", 2014),
        ("Royals", 2015) ]

for case let ("Red Sox", year) in worldSeriesWinners {
        print(year)
}
```

Filtering with the case statement

```
let myNumbers: [Int?] = [1, 2, nil, 4, 5, nil, 6]

for case let .some(num) in myNumbers {
        print(num)
}
```

Filtering with the case statement

```
let myNumbers: [Int?] = [1, 2, nil, 4, 5, nil, 6]

for case let .some(num) in myNumbers {
        print(num)
}
```

Using the if … case statement

```
enum Identifier {
        case Name(String)
        case Number(Int)
        case NoIdentifier
}

var playerIdentifier = Identifier.Number(42)

if case let .Number(num) = playerIdentifier {
        print("Player's number is \(num)")
}
```

Using the if … case statement

```
var playerIdentifier = Identifier.Number(2)

if case let .Number(num) = playerIdentifier, num == 2 {
        print("Player is either Xander Bogarts or Derek Jeter")
}
```

Control Transfer Statements

- Continue
- Break
- Fallthrough
- Guard

Continue

```
for i in 1...10 {
        if i % 2 == 0 {
                continue
        }
        print("\(i) is odd")
}
```

Break

```
for i in 1...10 {
        if i % 2 == 0 {
                break
        }
        print("\(i) is odd")
}
```

Fallthrough statement

```
var name = "Jon"
var sport = "Baseball"
switch sport {
        case "Baseball":
                print("\(name) plays Baseball")
                fallthrough
        case "Basketball":
                print("\(name) plays Basketball")
                fallthrough
        default:
                print("Unknown sport")
}
```

Guard statement

```
var x = 9
if x > 10 {
        // Functional code here
} else {
        // Do error condition
}
```

Guard statement

```
var x = 9
guard x > 10 else {
        // Do error condition
        return
}
// Functional code here
```

Functions

- Single parameter
- Multi-parameter
- Default values
- Return values
- External parameters
- Variadic parameters
- Inout parameters
- Nesting Functions

Single Parameter

```
func sayHello(name: String) -> Void {
        let retString = "Hello " + name print( retString)
}


sayHello(name:"Jon")
```

Single Parameter

```
func sayHello2(name: String) ->String {
        let retString = "Hello " + name
        return retString
}


var message = sayHello2(name:"Jon")

print(message)
```

Single Parameter

```
sayHello2(name:"Jon")

var message = sayHello2(name: "Jon")
```

Single Parameter

```
sayHello2(name:"Jon")

var message = sayHello2(name: "Jon")

_ = sayHello2(name:"Jon")
```

Multi Parameter

```
func sayHello(name: String, greeting: String) {
        print("\(greeting) \(name)")
}
```

Multi Parameter

```
func sayHello(name: String, greeting: String) {
        print("\(greeting) \(name)")
}

sayHello(name:"Jon", greeting:"Bonjour")
```

Defining a parameter's default values

```
func sayHello(name: String, greeting: String = "Bonjour") {
        print("\(greeting) \(name)")
}
```

Defining a parameter's default values

```
func sayHello(name: String, greeting: String = "Bonjour") {
        print("\(greeting) \(name)")
}


sayHello(name:"Jon")

sayHello(name:"Jon", greeting: "Hello")
```

Defining a parameter's default values

```
func sayHello4(name: String, name2: String = "Kim", greeting: String = "Bonjour")
{
        print"\(greeting) \(name) and \(name2)")
}

sayHello4(name:"Jon", greeting: "Hello")
```

Defining a parameter's default values

```
func sayHello4(name: String, name2: String = "Kim", greeting: String = "Bonjour")
{
        print"\(greeting) \(name) and \(name2)")
}
```

```
sayHello4(name:"Jon", greeting: "Hello")
```

```
Hello Jon and Kim
```

Returning multiple values from a function

```
func getNames() -> [String] {
        let retArray = ["Jon", "Kim", "Kailey", "Kara"]
        return retArray
}

var names = getNames()
```

Returning multiple values from a function

```
func getTeam() -> (team:String, wins:Int, percent:Double) {
        let retTuple = ("Red Sox", 99, 0.611)
        return retTuple
}

var t = getTeam()

print("\(t.team) had \(t.wins) wins")
```

Returning multiple values from a function

```
func getTeam() -> (team:String, wins:Int, percent:Double) {
        let retTuple = ("Red Sox", 99, 0.611)
        return retTuple
}

var t = getTeam()

print("\(t.team) had \(t.wins) wins")
```

Returning multiple values from a function

```
func getTeam() -> (team:String, wins:Int, percent:Double) {
        let retTuple = ("Red Sox", 99, 0.611)
        return retTuple
}

var t = getTeam()

print("\(t.team) had \(t.wins) wins")
```

Red Sox had 99 wins

Returning optional values

```
func getName() ->String {
        return nil
}
```

expression does not conform to type 'NilLiteralConvertible'

Returning optional values

```
func getName() ->String? {
        return nil
}
```

Returning optional values

```
func getTeam2(id: Int) -> (team:String, wins:Int, percent:Double)? {
        if id == 1 {
                return ("Red Sox", 99, 0.611)
        }
        return nil
}
```

Returning optional values

```
func getTeam2(id: Int) -> (team:String, wins:Int, percent:Double?) {
        if id == 1 {
                return ("Red Sox", 99, nil)
        }
        return nil
}
```

Adding external parameter names

```
func winPercentage(team: String, wins: Int, loses: Int) -> Double {
        return Double(wins) / Double(wins + loses)
}

var per = winPercentage(team: "Red Sox", wins: 99, loses: 63)
```

Adding external parameter names

```
func winPercentage(BaseballTeam team: String, withWins wins: Int, andLoses losses: Int) -> Double {
        return Double(wins) / Double(wins + losses)
}
```

Adding external parameter names

```
func winPercentage(BaseballTeam team: String, withWins wins: Int, andLoses losses: Int) -> Double {
        return Double(wins) / Double(wins + losses)
}

var per = winPercentage(BaseballTeam:"Red Sox", withWins:99, andLoses:63)
```

Using variadic parameters

```
func sayHello(greeting: String, names: String...) {
        for name in names {
                print("\(greeting) \(name)")
        }
}
```

Using variadic parameters

```
func sayHello(greeting: String, names: String...) {
        for name in names {
                print("\(greeting) \(name)")
        }
}

sayHello(greeting:"Hello", names: "Jon", "Kim")
```

Using variadic parameters

```
func sayHello(greeting: String, names: String...) {
        for name in names {
                print("\(greeting) \(name)")
        }
}

sayHello(greeting:"Hello", names: "Jon", "Kim")
```

```
Hello Jon
Hello Jim
```

Inout parameters

```
func reverse( first: inout String, second: inout String) {
        let tmp = first
        first = second
        second = tmp
}
```

Inout parameters

```
func reverse( first: inout String, second: inout String) {
        let tmp = first
        first = second
        second = tmp
}

var one = "One"
var two = "Two"
reverse(first: &one, second: &two)
print("one: \(one) two: \(two)")
```

Inout parameters

```
func reverse( first: inout String, second: inout String) {
        let tmp = first
        first = second
        second = tmp
}

var one = "One"
var two = "Two"
reverse(first: &one, second: &two)
print("one: \(one) two: \(two)")
```

one:two two:one

Nesting functions

```
func sort( numbers: inout [Int]) {
    func reverse( first: inout Int, second: inout Int) {
        let tmp = first
        first = second
        second = tmp
    }

    var count = numbers.count

    while count > 0 {
        for var i in 1..<count {
            if numbers[i] < numbers[i-1] {
                reverse(first: &numbers[i], second: &numbers[i-1])
            }
        }
        count -= 1
    }
}
```

Nesting functions

```
var nums: [Int] = [6,2,5,3,1]
sort(numbers: &nums)

for value in nums {
        print("--\(value)")
}
```

Classes and Structures

- Creating and using classes and structures
- Adding properties and property observers to classes and structures
- Adding methods to classes and structures
- Adding initializers to classes and structures
- Using access controls
- Creating a class hierarchy
- Extending a class
- Understanding memory management and ARC

Similarities between classes and structures

- **Properties**: These are used to store information in our classes and structures
- **Methods**: These provide functionality for our classes and structures
- **Initializers**: These are used when initializing instances of our classes and structures
- **Subscripts**: These provide access to values using the subscript syntax
- **Extensions**: These help in extending both classes and structures

Differences between classes and structures

- **Type**: A structure is a value type while a class is a reference type
- **Inheritance**: A structure cannot inherit from other types while a class can
- **Deinitializers**: Structures cannot have custom deinitializers while a class can

Creating a Class or Structure

```
class MyClass {
        // MyClass definition
}

struct MyStruct {
        // MyStruct definition
}
```

Properties

- **Stored properties**: They store variable or constant values as part of an instance of a class or structure. Stored properties can also have property observers, which can monitor the property for changes and respond with custom actions when the value of the property changes.
- **Computed properties**: They do not store a value by themselves, but retrieve and possibly set other properties. The value returned by a computed property can also be calculated when it is requested.

Stored Properties

```
struct MyStruct {
        let c = 5
        var v = ""
}

class MyClass {
        let c = 5
        var v = ""
}

var myStruct = MyStruct()
var myClass = MyClass()
```

Stored Properties

```
struct MyStruct {
        let c = 5
        var v = ""
}

var myStruct = MyStruct(v: "Hello")
```

Stored Properties

```
struct MyStruct {
        let c: Int
        var v = ""
}

var myStruct = MyStruct(c: 10, v: "Hello")
```

Stored Properties

```
struct EmployeeStruct {
        var firstName = ""
        var lastName = ""
        var salaryYear = 0.0
}

public class EmployeeClass {
        var firstName = ""
        var lastName = ""
        var salaryYear = 0.0
}
```

Computed Properties

```
var salaryWeek: Double {
        get{
                return self.salaryYear/52
        }
}
```

Computed Properties

```
var salaryWeek: Double {
                return self.salaryYear/52
}
```

Computed Properties

```
var salaryWeek: Double {
        get {
                return self.salaryYear/52
        }
        set (newSalaryWeek){
                self.salaryYear = newSalaryWeek*52
        }
}
```

Stored Properties

```
struct FixedLengthRange {
var firstValue: Int
let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Computed Properties

```
struct EmployeeStruct {
        var firstName = ""
        var lastName = ""
        var salaryYear = 0.0
        var salaryWeek: Double {
                get{
                        return self.salaryYear/52
                } set (newSalaryWeek){
                        self.salaryYear = newSalaryWeek*52
                }
        }
}
```

```
struct EmployeeStruct {
        var firstName = ""
        var lastName = ""
        var salaryYear = 0.0
        var salaryWeek: Double {
                get{
                        return self.salaryYear/52
                } set (newSalaryWeek){
                        self.salaryYear = newSalaryWeek*52
                }
        }
}
var f = EmployeeStruct(firstName: "Jon", lastName: "Hoffman", salaryYear: 39000)

print(f.salaryWeek)          //prints 750.00 to the console

f.salaryWeek = 1000

print(f.salaryWeek)          //prints 1000.00 to the console

print(f.salaryYear)          //prints 52000.00 to the console
```

Property observers

```
var salaryYear: Double = 0.0 {
        willSet(newSalary) {
                print("About to set salaryYear to \(newSalary)")
        } didSet {
                if salaryWeek > oldValue {
                        print("\(firstName) got a raise")
                } else {
                        print("\(firstName) did not get a raise")
                }
        }
}
```

```
struct EmployeeStruct {
        var firstName = ""
        var lastName = ""
        // var salaryYear = 0.0
        var salaryYear: Double = 0.0 {
                willSet(newSalary) {
                        print("About to set salaryYear to \(newSalary)")
                } didSet {
                        if salaryYear > oldValue {
                                print("\(firstName) got a raise")
                        } else {
                                print("\(firstName) did not get a raise")
                        }
                }
        }
        var salaryWeek: Double {
                get{
                        return self.salaryYear/52
                } set (newSalaryWeek){
                        self.salaryYear = newSalaryWeek*52
                }
        }
}
```

```
var f = EmployeeStruct(firstName: "Jon", lastName: "Hoffman", salaryYear: 39000)
print(f.salaryWeek)
f.salaryWeek = 1000
print(f.salaryWeek)
print(f.salaryYear)
```

750.0

About to set salaryYear to 52000.0

Jon got a raise

1000.0

52000.0

Methods

```
func getFullName() -> String {
        return firstName + " " + lastName
}
```

```
public class EmployeeClass {
        var firstName = ""
        var lastName = ""
        var salaryYear = 0.0
        var salaryWeek: Double {
                get{
                        return self.salaryYear/52
                } set (newSalaryWeek) {
                        self.salaryYear = newSalaryWeek*52
                }
        }

        func getFullName() -> String {
                return firstName + " " + lastName
        }
}
```

Methods

```
var e = EmployeeClass()

e.firstName = "Jon"
e.lastName = "Hoffman"
e.salaryYear = 50000.00


print(e.getFullName()) //Jon Hoffman is printed to the console
```

Custom Initializers

```
init() {
        //Perform initialization here
}
```

Custom Initializers

```
init() {
        self.firstName = ""
        self.lastName = ""
        self.salaryYear = 0.0
}

init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
        self.salaryYear = 0.0
}

init(firstName: String, lastName: String, salaryYear: Double) {
        self.firstName = firstName
        self.lastName = lastName
        self.salaryYear = salaryYear
}
```

Custom Initializers

```
var g = EmployeeClass()

var h = EmployeeStruct(firstName: "Me", lastName: "Moe")

var i = EmployeeClass(firstName: "Me", lastName: "Moe", salaryYear: 45000)
```

Internal and external parameter names

```
init(employeeWithFirstName firstName: String, lastName: String,
andSalary salaryYear: Double) {
        self.firstName = firstName
        self.lastName = lastName
        self.salaryYear = salaryYear
}

var i = EmployeeClass(employeeWithFirstName: "Me", lastName:
"Moe", andSalary: 45000)
```

Failable Intializers

```
init?(firstName: String, lastName: String, salaryYear: Double) {
        self.firstName = firstName
        self.lastName = lastName
        self.salaryYear = salaryYear
        if self.salaryYear < 20000 {
                return nil
        }
}
```

Failable Intializers

```
if let f = EmployeeClass(firstName: "Jon", lastName: "Hoffman",
salaryYear: 19000) {
        print(f.getFullName())
} else {
        print("Failed to initialize")
}
```

Access Levels

- Public
- Internal
- Private
- Fileprivate

Access Levels

```
private struct EmployeeStruct {}
public class EmployeeClass {}
internal class EmployeeClass2 {}
public var firstName = "Jon"
internal var lastName = "Hoffman"
private var salaryYear = 0.0
public func getFullName() -> String {}
private func giveRaise(amount: Double) {}
```

Class Inheritance

- Derive class from another class
- Single inheritance only
- Classes can be derived from a parent or superclass, but a structure cannot.

Class Inheritance

```
class Plant {
        var height = 0.0
        var age = 0
        func growHeight(inches: Double) {
                self.height += inches;
        }
}
```

Class Inheritance

```
class Tree: Plant {
        private var limbs = 0

        func limbGrow() {
                self.limbs += 1
        }

        func limbFall() {
                self.limbs -= 1
        }
}
```

Class Inheritance

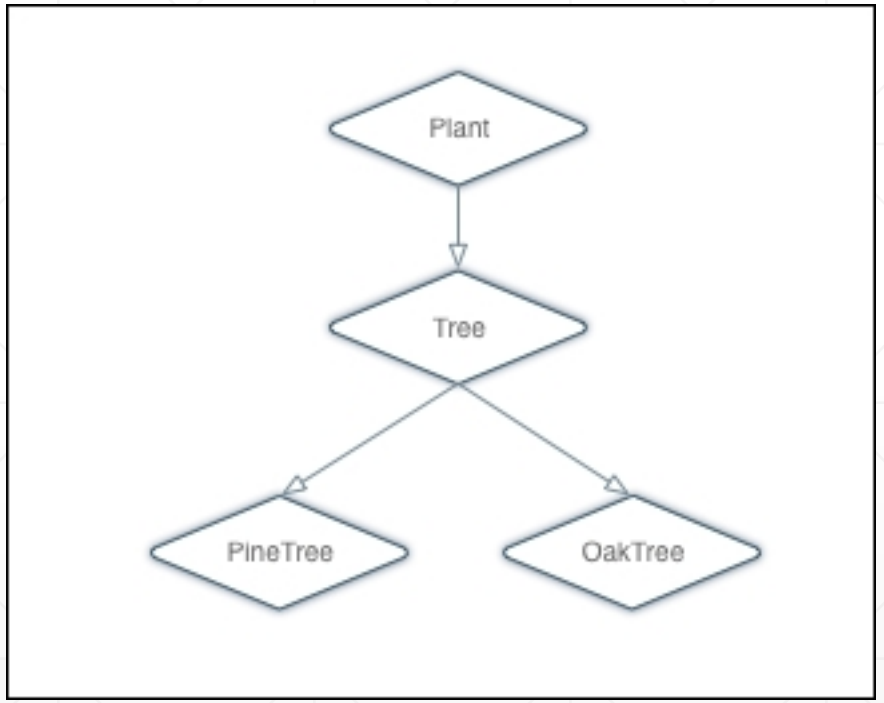
```
var tree = Tree()
tree.age = 5
tree.height = 4
tree.limbGrow()
tree.limbGrow()
```

Class Inheritance

```
class PineTree: Tree {
        var needles = 0
}

class OakTree: Tree {
        var leaves = 0
}
```

Class Inheritance



Class Inheritance

- Overriding methods
- Overriding properties
- Preventing overrides

Overriding methods

```
class Plant {
  var height = 0.0
  var age = 0

  func growHeight(inches: Double) {
    self.height +=  inches;
  }

  func getDetails() -> String {
    return "Plant Details"
  }
}
```

Overriding methods

```
class Tree: Plant {
  private var limbs = 0

  func limbGrow() {
    self.limbs += 1
  }
  func limbFall() {
    self.limbs -= 1
  }

  override func getDetails() -> String {
    return "Tree Details"
  }
}
```

Overriding Methods

```
var plant = Plant()
var tree = Tree()
print("Plant: \(plant.getDetails())")
print("Tree: \(tree.getDetails())")
```

Overriding Methods

```
var plant = Plant()
var tree = Tree()
print("Plant: \(plant.getDetails())")
print("Tree: \(tree.getDetails())")
```

Plant: Plant Details
Tree: Tree Details

```
class Plant {
    var height: Double = 0.0
    var  age  =  0
    func growHeight(inches: Double) {
        self.height += inches
     }
    func getDetails() -> String {
        return "Height: \(self.height)  age:  \(self.age)"
     }
}
```

```
class Tree: Plant {
    private var limbs = 0
    func limbGrow() {
            self.limbs  +=  1
      }
    func limbFall() {
            self.limbs  -=  1
      }
    override func getDetails() -> String {
       return " \(super.getDetails()) Limbs:  \(self.limbs)"
      }
}
```

Overriding Methods

```
var tree = OakTree()
tree.age = 5
tree.height = 4
tree.leaves = 50
tree.limbGrow()
tree.limbGrow()
print(tree.getDetails())
```

Overriding Methods

```
var tree = OakTree()
tree.age = 5
tree.height = 4
tree.leaves = 50
tree.limbGrow()
tree.limbGrow()
print(tree.getDetails())
```

Height: 4.0 age: 5 limbs: 2 Leaves: 50

Overriding Properties

```
var description: String {
        get {
                return "Base class is Plant."
        }
}
```

Overriding Properties

```
override var description: String {
        return "\(super.description) I am a Tree class."
}
```

Overriding Properties

```
override var description: String {
        return "\(super.description) I am a Tree class."
}
```

Base class is Plant. I am a Tree class.

Preventing Overrides

- Use final keyword

```
final func growHeight(inches: Double) {
        self.height += inches
}
```