Hello World

`print()`

The `print()` function outputs one or more values to the terminal.

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

// Prints: Hello, world!
```

Program Structure

The program runs line by line, from top to bottom:

- Line 1 prints `Hola`
- Line 2 prints `Buenos días`

```
print("Hola")
print("Buenos días")
```

Single-line Comments

Single-line comments are created using two consecutive forward slashes. The compiler ignores any text after `//` on the same line.

```
// This line denotes a comment in
Swift.
```

Multiline Comments

Multiline comments are created using `/*` to begin the comment, and `*/` to end the comment. The compiler ignores any text in between.

```
/*
This is all commented out.
None of it is going to run!
*/
```

Print Share

Variables

Variables

A variable refers to a storage location in the computer's memory that one can set aside to save, retrieve, and manipulate data.

```
var score = 0
```

Constants

Constants refer to fixed values that a program may not alter during its execution. One can be declared by using the `let` keyword.

```
let pi = 3.14
```

Arithmetic Operators

Swift supports arithmetic operators for:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` remainder

```
var x = 0

x = 4 + 2  // x is now 6
x = 4 - 2  // x is now 2
x = 4 * 2  // x is now 8
x = 4 / 2  // x is now 2
x = 4 % 2  // x is now 0
```

Types

Type annotation can be used during declaration. The basic data types are:

- `Int` : integer numbers
- `Double` : floating-point numbers
- `String` : a sequence of characters
- `Bool` : truth values

```
var age: Int = 28

var price: Double = 8.99

var message: String = "good nite"

var lateToWork: Bool = true
```

String Interpolation

String interpolation can be used to construct a `String` from a mix of variables, constants, and others by including their values inside a string literal.

```
var apples = 6

print("I have \(apples) apples!")

// Prints: I have 6 apples!
```

Compound Assignment Operators

Compound assignment operators provide a shorthand method for updating the value of a variable:

- `+=` add and assign the sum
- `-=` subtract and assign the difference
- `*=` multiply and assign the product
- `/=` divide and assign the quotient
- `%=` divide and assign the remainder

```
var numberOfDogs = 100
numberOfDogs += 1

print("There are \(numberOfDogs) dalmations!")

// Prints: There are 101 dalmations!
```

codecademy

Conditionals & Logic

`if` Statement

An `if` statement executes a code block when its condition evaluates to `true`. If the condition is `false`, the code block does not execute.

```
var halloween = true

if halloween {
  print("Trick or treat!")
}

// Prints: Trick or treat!
```

`else` Statement

An `else` statement is a partner to an `if` statement. When the condition for the `if` statement evaluates to `false`, the code within the body of the `else` will execute.

```
var turbulence = false

if turbulence {
  print("Please stay seated.")
} else {
  print("You may freely move around.")
}

// Prints: You may freely move around.
```

`else if` Statement

An `else if` statement provides additional conditions to check for within a standard `if` / `else` statement. `else if` statements can be chained and exist only after an `if` statement and before an `else`.

```
var weather = "rainy"

if weather == "sunny" {
  print("Grab some sunscreen")
} else if weather == "rainy" {
  print("Grab an umbrella")
} else if weather == "snowing" {
  print("Wear your snow boots")
} else {
  print("Invalid weather")
}

// Prints: Grab an umbrella
```

Comparison Operators

Comparison operators compare the values of two operands and return a Boolean result:

- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal to
- `==` equal to
- `!=` not equal to

```
5 > 1       // true
6 < 10      // true
2 >= 3      // false
3 <= 5      // true
"A" == "a"  // false
"B" != "b"  // true
```

Ternary Conditional Operator

The ternary conditional operator, denoted by a `?`, creates a shorter alternative to a standard `if` / `else` statement. It evaluates a single condition and if `true`, executes the code before the `:`. If the condition is `false`, the code following the `:` is executed.

```
var driverLicense = true

driverLicense ? print("Driver's Seat")
: print("Passenger's Seat")

// Prints: Driver's Seat
```

`switch` Statement

The `switch` statement is a type of conditional used to check the value of an expression against multiple cases. A `case` executes when it matches the value of the expression. When there are no matches between the `case` statements and the expression, the `default` statement executes.

```
var secondaryColor = "green"

switch secondaryColor {
  case "orange":
    print("Mix of red and yellow")
  case "green":
    print("Mix of blue and yellow")
  case "purple":
    print("Mix of red and blue")
  default:
    print("This might not be a
secondary color.")
}

// Prints: Mix of blue and yellow
```

`switch` Statement: Interval Matching

Intervals within a `switch` statement's `case` provide a range of values that are checked against an expression.

```
let year = 1905
var artPeriod: String

switch year {
  case 1860...1885:
    artPeriod = "Impressionism"
  case 1886...1910:
    artPeriod = "Post Impressionism"
  case 1912...1935:
    artPeriod = "Expressionism"
  default:
    artPeriod = "Unknown"
}

// Prints: Post Impressionism
```

`switch` Statement: Compound Cases

A compound case within a `switch` statement is a single `case` that contains multiple values. These values are all checked against the `switch` statement's expression and are separated by commas.

```
let service = "Seamless"

switch service {
  case "Uber", "Lyft":
    print("Travel")
  case "DoorDash", "Seamless",
"GrubHub":
    print("Restaurant delivery")
  case "Instacart", "FreshDirect":
    print("Grocery delivery")
  default:
    print("Unknown service")
}

// Prints: Restaurant delivery
```

`switch` Statement: `where` Clause

Within a `switch` statement, a `where` clause is used to test additional conditions against an expression.

```
let num = 7

switch num {
  case let x where x % 2 == 0:
    print("\(num) is even")
  case let x where x % 2 == 1:
    print("\(num) is odd")
  default:
    print("\(num) is invalid")
}

// Prints: 7 is odd
```

Logical Operator `!`

The logical NOT operator, denoted by a `!`, is a prefix operator that negates the value on which it is prepended. It returns `false` when the original value is `true` and returns `true` when the original value is `false`.

```
!true          // false
!false         // true
```

Logical Operator `&&`

The logical AND operator, denoted by an `&&`, evaluates two operands and returns a Boolean result. It returns `true` when both operands are `true` and returns `false` when at least one operand is `false`.

```
true && true    // true
true && false   // false
false && true   // false
false && false  // false
```

Logical Operator `||`

The logical OR operator, denoted by `||`, evaluates two operands and returns a Boolean result. It returns `false` when both operands are `false` and returns `true` when at least one operand is `true`.

```
true || true    // true
true || false   // true
false || true   // true
false || false  // false
```

Combining Logical Operators

Logical operators can be chained in order to create more complex logical expressions. When logical operators are chained, it's important to note that the `&&` operator has a higher precedence over the `||` operator and will get evaluated first.

```
!false && true || false  // true

/*
!false && true evaluates first and
returns true. Then, the expression,
true || false evaluates and returns the
final result, true.
*/

false || true && false   // false

/*
true && false evaluates first which
returns false. Then, the expression,
false || false evaluates and returns
the final result, false.
*/
```

Controlling Order of Execution

Within a Swift logical expression, parentheses, `()`, can be used to organize and control the flow of operations. The usage of parentheses within a logical expression overrides operator precedence rules and improves code readability.

```
// Without parentheses:

true || true && false || false      //
true

// With parentheses:

(true || true) && (false || false)  //
false
```

codecademy

Loops

Ranges

Ranges created by the `...` operator will include the numbers from the lower bound to (and includes) the upper bound.

```
let zeroToThree = 0...3

// zeroToThree: 0, 1, 2, 3
```

`stride()` Function

Calling `stride()` with the 3 necessary arguments creates a collection of numbers; the arguments decide the starting number to, the (excluded) ending number, and how to increment/decrement from the start to the end.

```
for oddNum in stride(from: 1, to: 5,
by: 2) {
  print(oddNum)
}

// Prints: 1
// Prints: 3
```

`for - in` Loop

The `for - in` loop is used to iterate over collections, including strings and ranges.

```
for char in "hehe" {
  print(char)
}

// Prints: h
// Prints: e
// Prints: h
// Prints: e
```

`continue` Keyword

The `continue` keyword will force the loop to move on to the next iteration.

```
for num in 0...5 {
  if num % 2 == 0 {
    continue
  }
  print(num)
}

// Prints: 1
// Prints: 3
// Prints: 5
```

`break` Keyword

To terminate a loop before its completion, use the `break` keyword.

```
for char in
"supercalifragilisticexpialidocious" {
  if char == "c" {
    break
  }
  print(char)
}

// Prints: s
// Prints: u
// Prints: p
// Prints: e
// Prints: r
```

Using Underscore

Use `_` instead of a placeholder variable if the variable is not referenced in the `for - in` loop body.

```
for _ in 1...3 {
  print("Olé")
}

// Prints: Olé
// Prints: Olé
// Prints: Olé
```

`while` Loop

A `while` loop accepts a condition and continually executes its body's code for as long as the provided condition is `true`.
If the condition is never `false` then the loop continues to run and the program is stuck in an infinite loop.

```
var counter = 1
var stopNum = Int.random(in: 1...10)

while counter < stopNum {
  print(counter)
  counter += 1
}

// The loop prints until the stop
condition is met
```

codecademy

Arrays & Sets

Array

An array stores an ordered collection of values of the same data type.
Use the initializer syntax, `[Type]()`, to create an empty array of a certain type.

```
var scores = [Int]()

// The array is empty: []
```

Initialize with Array Literal

An array can be initialized with an array literal, which is a short-hand method of writing one or more values as an array collection.
An array literal is written as a list of values, separated by commas, and surrounded by a pair of square brackets.

```
// Using type inference:
var snowfall = [2.4, 3.6, 3.4, 1.8,
0.0]

// Being explicit with the type:
var temp: [Int] = [33, 31, 30, 38, 44]
```

Index

An index refers to an item's position within an ordered list. Use the subscript syntax, `array[index]`, to retrieve an individual element from an array.
Note: Swift arrays are zero-indexed, meaning the first element has index 0.

```
var vowels = ["a", "e", "i", "o", "u"]

print(vowels[0])  // Prints: a
print(vowels[1])  // Prints: e
print(vowels[2])  // Prints: i
print(vowels[3])  // Prints: o
print(vowels[4])  // Prints: u
```

`.count` Property

The `.count` property returns the number of elements in an array.

```
var grocery = ["🥓", "🥞", "🍪", "🥛", "🍊"]

print(grocery.count)

// Prints: 5
```

`.append()` Method and `+=` Operator

The `.append()` method can be called on an array to add an item to the end of the array.
The `+=` addition assignment operator can be used to add the elements of another array to the existing array.

```
var gymBadges = ["Boulder", "Cascade"]

gymBadges.append("Thunder")
gymBadges += ["Rainbow", "Soul"]

// ["Boulder", "Cascade", "Thunder", "Rainbow", "Soul"]
```

`.insert()` and `.remove()` Methods

The `.insert()` method can be called on an array to add an element at a specified index. It takes two arguments: `value` and `at: index`.
The `.remove()` method can be called on an array to remove an element at a specified index. It takes one argument: `at: index`.

```
var moon = ["🌕", "🌖", "🌗", "🌑"]

moon.insert("🌕", at: 0)

// ["🌕", "🌕", "🌖", "🌗", "🌑"]

moon.remove(at: 4)

// ["🌕", "🌕", "🌖", "🌗"]
```

Iterating Over an Array

In Swift, a `for - in` loop can be used to iterate through the items of an array.
This is a powerful tool for working with and manipulating a large amount of data.

```
var employees = ["Michael", "Dwight", "Jim", "Pam", "Andy"]

for person in employees {
  print(person)
}

// Prints: Michael
// Prints: Dwight
// Prints: Jim
// Prints: Pam
// Prints: Andy
```

Swift Sets

We can use a set to store unique elements of the same data type.

```
var paintingsInMOMA: Set = ["The Dream", "The Starry Night", "The False Mirror"]
```

Empty Sets

An empty set is a set that contains no values inside of it.

```
var team = Set<String>()

print(team)
// Prints: []
```

Populated Sets

To create a set populated with values, use the `Set` keyword before the assignment operator. The values of the set must be contained within brackets `[]` and separated with commas `,`.

```
var vowels: Set = ["a", "e", "i", "o", "u"]
```

.insert()

To insert a single value into a set, append `.insert()` to a set and place the new value inside the parentheses `()`.

```
var cookieJar: Set = ["Chocolate Chip", "Oatmeal Raisin"]

// Add a new element
cookieJar.insert("Peanut Butter Chip")
```

.remove() and .removeAll() Methods

To remove a single value from a set, append `.remove()` to a set with the value to be removed placed inside the parentheses `()`. To remove every single value from a set at once, append `.removeAll()` to a set.

```
var oddNumbers: Set = [1, 2, 3, 5]

// Remove an existing element
oddNumbers.remove(2)

// Remove all elements
oddNumbers.removeAll()
```

.contains()

Appending `.contains()` to an existing set with an item in the parentheses `()` will return a `true` or `false` value that states whether the item exists within the set.

```
var names: Set = ["Rosa", "Doug", "Waldo"]

print(names.contains("Lola")) // Prints: false

if names.contains("Waldo"){
  print("There's Waldo!")
} else {
  print("Where's Waldo?")
}
// Prints: There's Waldo!
```

Iterating Over a Set

A `for - in` loop can be used to iterate over each item in a set.

```
var recipe: Set = ["Chocolate chips", "Eggs", "Flour", "Sugar"]

for ingredient in recipe {
  print ("Include \(ingredient) in the recipe.")
}
```

.isEmpty Property

Use the built-in property `.isEmpty` to check if a set has no values contained in it.

```
var emptySet = Set<String>()

print(emptySet.isEmpty)  // Prints: true

var populatedSet: Set = [1, 2, 3]

print(populatedSet.isEmpty) // Prints: false
```

.count Property

The property `.count` returns the number of elements contained within a set.

```
var band: Set = ["Guitar", "Bass", "Drums", "Vocals"]

print("There are \(band.count) players in the band.")
// Prints: There are 4 players in the band.
```

.intersection() Operation

The `.intersection()` operation populates a new set of elements with the overlapping elements of two sets.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.intersection(setB)
print(setC)  // Prints: ["D", "C"]
```

.union() Operation

The `.union()` operation populates a new set by taking all the values from two sets and combining them.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.union(setB)
print(setC)
// Prints: ["B", "A", "D", "F", "C",
"E"]
```

.symmetricDifference() Operation

The `.symmetricDifference()` operation creates a new set with all the non-overlapping values between two sets.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC =
setA.symmetricDifference(setB)
print(setC)
// Prints: ["B", "E", "F", "A"]
```

.subtracting() Operation

The `.subtracting()` operation removes the values of one second set from another set and stores the remaining values in a new set.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D"]

var setC = setA.subtracting(setB)
print(setC)
// Prints: ["B", "A"]
```

Dictionaries

Assigning a Value to a Variable

To assign the value of a key-value pair to a variable, set the value of a variable to `dictionaryName[keyValue]`.

Note: Assigning the value of a key-value pair to a variable will return an optional value. To extract the value, use optional unwrapping.

```
var primaryHex = [
  "red": "#ff0000",
  "yellow": "#ffff00",
  "blue": "#0000ff",
]

print("The hex code for blue is \
(primaryHex["blue"])")
// Prints: The hex code for blue is
Optional("#0000ff")

if let redHex = primaryHex["red"] {
  print("The hex code for red is \
(redHex)")
}
// Prints: The hex code for red is
#ff0000
```

Dictionary

A dictionary is an unordered collection of paired data, or key-value pairs.

```
var dictionaryName = [
  "Key1": "Value1",
  "Key2": "Value2",
  "Key3": "Value3"
]
```

Keys

Every key in a dictionary is unique.
Keys can be be used to access, remove, add, or modify its associated value.

```
// Each key is unique even if they all contain the same value

var fruitStand = [
  "Coconuts": 12,
  "Pineapples": 12,
  "Papaya": 12
]
```

Type Consistency

In a dictionary, the data type of the keys and the values must remain consistent.

```
// Contains only String keys and Int values

var numberOfSides = [
  "triangle": 3,
  "square": 4,
  "rectangle": 4
]
```

Initialize a Populated Dictionary

Dictionary literals contain lists of key-value pairs that are separated by commas; this syntax can be used to create dictionaries that are populated with values.

```
var employeeID = [
  "Hamlet": 1367,
  "Horatio": 8261,
  "Ophelia": 9318
]
```

Initialize an Empty Dictionary

An empty dictionary is a dictionary that contains no key-value pairs.
There is more than one way to initialize an empty dictionary; the method chosen is purely up to preference and makes no impact on the dictionary.

```
// Initializer syntax:
var yearlyFishPopulation = [Int: Int]()

// Empty dictionary literal syntax:
var yearlyBirdPopulation: [Int: Int] =
[:]
```

Adding to a Dictionary

To add a new key-value to a dictionary, use subscript syntax by adding a new key contained within brackets `[ ]` after the name of a dictionary and a new value after the assignment operator (=).

```
var pronunciation = [
  "library": "lai·breh·ree",
  "apple": "a·pl"
]

// New key: "programming", New value:
"prow·gra·muhng"
pronunciation["programming"] =
"prow·gra·muhng"
```

Removing Key-Value Pairs

To remove a key-value pair from a dictionary, set the value of a key to `nil` with subscript syntax or use the `.removeValue()` method.
To remove all the values in a dictionary, append `.removeAll()` to a dictionary.

```
var bookShelf = [
  "Goodnight Moon": "Margaret Wise Brown",
  "The BFG": "Roald Dahl",
  "Falling Up": "Shel Silverstein",
  "No, David!": "David Shannon"
]

// Remove value by setting key to nil
bookShelf["The BFG"] = nil

// Remove value using .removeValue()
bookShelf.removeValue(forKey: "Goodnight Moon")

// Remove all values
bookShelf.removeAll()
```

Modifying Key-Value Pairs

To change the value of a key-value pair, use the `.updateValue()` method or subscript syntax by appending brackets `[ ]` with an existing key inside them to a dictionary's name and then adding an assignment operator (=) followed by the modified value.

```
var change = [
  "Quarter": 0.29,
  "Dime": 0.15,
  "Nickel": 0.05,
  "Penny": 0.01
]

// Change value using subscript syntax
change["Quarter"] = .25

// Change value using .updateValue()
change.updateValue(.10, forKey: "Dime")
```

`.isEmpty` Property

The `.isEmpty` property will return a `true` value if there are no key-value pairs in a dictionary and `false` if the dictionary does contain key-value pairs.

```
var bakery = [String:Int]()

// Check if dictionary is empty
print(bakery.isEmpty)  // Prints true

bakery["Cupcakes"] = 12

// Check if dictionary is empty
print(bakery.isEmpty)  // Prints false
```

`.count` Property

The `.count` property returns an integer that represents how many key-value pairs are in a dictionary.

```
var fruitStand = [
  "Apples": 12,
  "Bananas": 20,
  "Oranges", 17
]

print(fruitStand.count)  // Prints: 3
```

Iterating Over a Dictionary

A `for - in` loop can be used to iterate through the keys and values of a dictionary.

```
var emojiMeaning = [
  "🤔": "Thinking Face",
  "😪": "Sleepy Face",
  "😵": "Dizzy Face"
]

// Iterate through both keys and values
for (emoji, meaning) in emojiMeaning {
  print("\(emoji) is known as the '\
(meaning) Emoji'")
}

// Iterate only through keys
for emoji in emojiMeaning.keys {
  print(emoji)
}

// Iterate only through values
for meaning in emojiMeaning.values {
  print(meaning)
}
```

Functions

What is a Function?

A function is a named, reusable block of code responsible for a certain task. It consists of a definition that includes the `func` keyword, name, optional parameters, and return type as well as a body that contains the code block needed to execute its task.

```
func washCar() -> Void {
  print("Soap")
  print("Scrub")
  print("Rinse")
  print("Dry")
}
```

Calling a Function

The process of executing the code inside the body of a function is known as calling it. A function is called by typing its name followed by a set of parentheses, `()`, which should hold any necessary arguments.

```
func greetLearner() {
 print("Welcome to Codecademy!")
}

// Function call:
greetLearner() // Prints: Welcome to Codecademy!
```

Returning a Value

A value can be passed from a function with a `return` statement. If a function returns a value, the return type must be specified in the function definition.

```
let birthYear = 1994
var currentYear = 2020

func findAge() -> Int {
  return currentYear - birthYear
}

print(findAge()) // Prints: 26
```

Multiple Parameters

A function can accept multiple parameters in its definition. All parameters must be separated by commas and assigned arguments during the function call.

```
func convertFracToDec(numerator:
Double, denominator: Double) -> Double
{
  return numerator / denominator
}

let decimal =
convertFracToDec(numerator: 1.0,
denominator: 2.0)
print(decimal) // Prints:  0.5
```

Returning Multiple Values

A function can return multiple values in the form of a tuple. When a tuple is returned, each value within its parentheses needs to be labeled and assigned a type in the function definition.

```
func smartphoneModel() -> (name:
String, version: String, yearReleased:
Int) {
  return ("iPhone", "8 Plus", 2017)
}

let phone = smartphoneModel()

print(phone.name) // Prints: iPhone
print(phone.version) // Prints: 8 Plus
print(phone.yearReleased) // Prints:
2017
```

Omitting Argument Labels

An argument label can be omitted in the function call when an `_` is prepended to a parameter in the function definition.

```
func findDifference(_ a: Int, b: Int) -
> Int {
  return a - b
}

print(findDifference(6, b: 4)) //
Prints: 2
```

Parameters and Arguments

Parameters are optional input values that exist between the `()` in a function definition. For each defined parameter, a real value must be passed in as an argument during the function call.

```
func findSquarePerimeter(side: Int) ->
Int {
  return side * 4
}

let perimeter =
findSquarePerimeter(side: 5)
print(perimeter) // Prints: 20

// Parameter: side
// Argument:  5
```

Implicit Return

Implicit returns were introduced in Swift version 5.1 to reduce the amount of code within a function body. An implicit return allows for an omitted `return` keyword from a function that returns a single value or expression.

```
func nextTotalSolarEclipse() -> String
{
  "April 8th, 2024 🌎"
}

print(nextTotalSolarEclipse()) //
Prints: April 8th, 2024 🌎
```

Default Parameters

A default parameter has a real value assigned to a parameter in the function's definition. When a function with a default parameter is called, an argument for that parameter is not required. If the argument is included, that value will overwrite the default value and be used in the function body.

```
func timeToFinishBook(numWords: Double, wordsPerMin: Double = 200) -> Double {
  let totalMinutes = numWords / wordsPerMin
  return totalMinutes / 60
}

print("\(timeToFinishBook(numWords: 93000)) hours")
// Prints: 7.75 hours
```

Variadic Parameters

A variadic parameter is a parameter that accepts zero or more values of a certain type. It is denoted by three consecutive dots, `...`, following a parameter's data type in the function definition.

```
func totalStudents(students: String...) -> Int {
  let numStudents = students.count
  return numStudents
}

print(totalStudents(students: "Jamie", "Michael", "Rose", "Idris")) // Prints: 4
```

In-Out Parameters

An in-out parameter allows a function to reassign the value of a variable passed in as an argument. An in-out parameter is denoted by `inout` in the function definition, and when the function is called, its variable argument must be prepended with an `&`.

```
var currentSeason = "Winter"

func determineSeason(monthNum: Int,
season: inout String) {

switch monthNum {
  case 1...2:
    season = "Winter ☃️"
  case 3...6:
    season = "Spring 🌱"
  case 7...9:
    season = "Summer ⛱"
  case 10...11:
    season = "Autumn 🍂"
  default:
    season = "Unknown"
  }
}

determineSeason(monthNum: 4, season:
&currentSeason)

print(currentSeason) // Spring 🌱
```

Print Share

codecademy

Structures

Structure Creation

Structures, or structs, are used to programmatically represent a real-life object in code. Structures are created with the `struct` keyword followed by its name and then body containing its properties and methods.

```
struct Building {
  var address: String
  var floors: Int

  init(address: String, floors: Int,
color: String) {
    self.address = address
    self.floors = floors
  }
}
```

Default Property Values

A structure's properties can have preassigned default values to avoid assigning values during initialization. Optionally, these property's values can still be assigned a value during initialization.

```
struct Car {
  var numOfWheels = 4
  var topSpeed = 80
}

var reliantRobin = Car(numOfWheels: 3)

print(reliantRobin.numOfWheels) //
Prints: 3
print(reliantRobin.topSpeed)    //
Prints: 80
```

Structure Instance Creation

A new instance of a structure is created by using the name of the structure with parentheses `()` and any necessary arguments.

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

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

// Instance of Person:
var morty = Person(name: "Morty", age:
14)
```

Checking Type

The built-in function `type(of:)` accepts an argument and returns the type of the argument passed.

```
print(type(of: "abc")) // Prints:
String
print(type(of: 123))   // Prints: 123
```

`init()` Method

Structures can have an `init()` method to initialize values to an instance's properties. Unlike other methods, The `init()` method does not need the `func` keyword. In its body, the `self` keyword is used to reference the actual instance of the structure.

```
struct TV {
  var screenSize: Int
  var displayType: String

  init(screenSize: Int, displayType:
String) {
    self.screenSize = screenSize
    self.displayType = displayType
  }
}

var newTV = TV(screenSize: 65,
displayType: "LED")
```

Structure Methods

Methods are like functions that are specifically called on an instance. To call the method, an instance is appended with the method name using dot notation followed by parentheses that include any necessary arguments.

```
struct Dog {
  func bark() {
    print("Woof")
  }
}

let fido = Dog()
fido.bark() // Prints: Woof
```

Mutating Methods

Structure methods declared with the `mutating` keyword allow the method to affect an instance's own properties.

```
struct Menu {
  var menuItems = ["Fries", "Burgers"]

  mutating func addToMenu(dish: String)
{
    self.menuItems.append(dish)
  }
}

var dinerMenu = Menu()

dinerMenu.addToMenu(dish: "Toast")
print(dinerMenu.menuItems)
// Prints: ["Fries", "Burgers",
"Toast"]
```

codecademy

Classes

Swift Class

A class is used to programmatically represent a real-life object in code. Classes are defined by the keyword `class` followed by the class name and curly braces that store the class's properties and methods.

```
// Using data types:
class Student {
  var name: String
  var year: Int
  var gpa: Double
  var honors: Bool
}


// Using default property values:
class Student {
  var name = ""
  var year = 0
  var gpa = 0.0
  var honors = false
}
```

Instance of a Class

Creating a new instance of a class is done by calling a defined class name with parentheses `()` and any necessary arguments.

```
class Person {
  var name = ""
  var age = 0
}

var sonny = Person()

// sonny is now an instance of Person
```

Class Properties

Class properties are accessed using dot syntax, i.e. `.property`.

```
var ferris = Student()

ferris.name = "Ferris Bueller"
ferris.year = 12
ferris.gpa = 3.81
ferris.honors = false
```

`init()` Method

Classes can be initialized with an `init()` method and corresponding initialized properties. In the `init()` method, the `self` keyword is used to reference the actual instance of the class assign property values.

```
class Fruit {
  var hasSeeds = true
  var color: String

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

let apple = Fruit(color: "red")
```

Inheritance

A class can inherit, or take on, another class's properties and methods:

- The new inheriting class is known as a subclass.
- The class that the subclass inherits from is known as its superclass.

```
// Suppose we have a BankAccount class:

class BankAccount {
  var balance = 0.0

  func deposit(amount: Double) {
    balance += amount
  }

  func withdraw(amount: Double) {
    balance -= amount
  }
}


// And we want a new SavingsAccount
class that inherits from BankAccount:

class SavingsAccount: BankAccount {
  var interest = 0.0

  func addInterest() {
    let interest = balance * 0.005
    self.deposit(amount: interest)
  }
}

// Here, the new SavingsAccount class
(subclass) automatically gains all of
the characteristics of BankAccount
class (superclass). In addition, the
SavingsAccount class defines a
.interest property and a .addInterest()
method.
```

Overriding

A subclass can provide its own custom implementation of a property or method that is inherited from a superclass. This is known as overriding.

```
// Suppose we have a BankAccount class:

class BankAccount {
  var balance = 0.0

  func deposit(amount: Double) {
    balance += amount
  }

  func withdraw(amount: Double) {
    balance -= amount
  }
}


// Suppose we want a new SavingsAccount
class and we want to override the
.withdraw() method from its superclass
BankAccount:

class SavingsAccount: BankAccount {
  var interest = 0.0
  var numWithdraw = 0

  func addInterest() {
    let interest = balance * 0.01
    self.deposit(amount: interest)
  }

  override func withdraw(amount:
Double) {
    balance -= amount
    numWithdraw += 1
  }
}
```

Reference Types

Classes are reference types, while structures are value types.
Unlike value types, reference types are not copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used.

Properties and Access Control

Access Control

Access control specifies which methods and properties can be accessed from outside that scope of a structure, file, or module. Access control makes it possible to hide implementation details and protect properties from being changed at unexpected times.

Access Levels

Swift provides several different levels of access. From least restrictive to most restrictive they are:

- open / public
- internal
- fileprivate
- private

```
// public structures can be accessed in other modules
public struct User {
  // internal is the default level of access control
        let name: String
  // fileprivate methods can only be accessed inside of the file they're declared in
  fileprivate func incrementVisitCount() {
    visitCount += 1
  }
  // private properties can only be accessed inside their structure's definition
  private let visitCount = 0
}
```

Private Properties and Methods

Mark methods and properties as `private` to prevent them from being accessed outside of the structure, class, or enumeration's definition.

```
struct User {
  let name: String
  init(name: String) {
    self.name = name
    uploadNewUser()
  }
  private func uploadNewUser() {
    print("Uploading the new user...")
  }
}
```

Read-only Computed Properties

Read-only computed properties can be accessed, but not assigned to a new value. To define a read-only computed property, either use the `get` keyword without a `set` keyword, or omit keywords entirely.

```
struct Room {
  let width: Double
  let height: Double
  var squareFeet: Double {
    return width * height
  }
  var description: String {
    get {
      return "This room is \(width) x \
(height)"
    }
  }
}
```

Property Observers

Property observers execute code whenever a property is changed. The `willSet` property observer is triggered right before the property is changed and creates a `newValue` variable within the block's scope. The `didSet` property observer is triggered right after the property is changed and creates an `oldValue` within the block's scope.

```
struct Employee {
  var hourlyWage = 15 {
    willSet {
      print("The hourly wage is about to be changed from \(hourlyWage) to \(newValue)")
    }
    didSet {
      print("The hourly wage has been changed from \(oldValue) to \(hourlyWage)")
    }
  }
}

var codey = Employee()
codey.hourlyWage = 20

// Prints:
// The hourly wage is about to be changed from 15 to 20
// The hourly wage has been changed from 15 to 20
```

Private Setters

Properties marked as `private(set)` can be accessed from outside the scope of its structure, but only assigned within it. This allows the setter to be more restrictive than the getter.

```
struct User {
        private(set) var name: String
  mutating func updateName(to newName:
String) {
        if newName != "" {
        name = newName
    }
  }
}

var currentUser = User(name: "codey")
currentUser.updateName(to: "Codey")
print(currentUser.name)
// currentUser.name = "Bob" // This
line doesn't compile because the 'name'
setter is inaccessible
```

Static Properties and Methods

The `static` keyword is used to declare type methods and properties. These are accessed from the type itself rather than an instance.

```
struct User {
  static var allUsers = [User]()
  let id: Int
  init(id: Int) {
    self.id = id
    User.allUsers.append(self)
  }
}

let userOne = User(id: 1)
let userTwo = User(id: 2)
let userThree = User(id: 3)

print(User.allUsers) // Prints:
[User(id: 1), User(id: 2), User(id: 3)]
```

Extensions

The `extension` keyword is used to continue defining an existing class, structure, or enumeration from anywhere in a codebase. Extensions can have new methods, internal types, and computed properties, but can't contain new stored properties.

```
struct User {
  let name: String
}

extension User {
  var description: String {
    return "This is a user named \
(name)"
  }
}
```