Structures

• All structures have an automatically-generated memberwise initializer

```ocaml
let vga = Resolution(width: 640, height: 480)
```

• Structures are value types
  
  — Any structure instance is always copied when it is passed around the code

```ocaml
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd //this is a copy: cinema and hd are two different instances

cinema.width = 2048
print("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
print("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```
Classes

• Classes are reference types — they are not copied, but a reference to the same instance is made

```javascript
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0

print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// prints "The frameRate property of tenEighty is now 30.0"

• Identical to (===), not identical to (!==), and equal (==)

if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
```
Stored properties

```swift
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

• If you create an instance of a structure, assign that instance to a constant and change it, there is an error:

```swift
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

• A lazy stored property is a property whose initial value is not calculated until the first time it is used

```swift
class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
}
```
Computed properties

```swift
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
                   size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// prints "square.origin is now at (10.0, 10.0)"
```
Read-only computed properties

- A computed property with a getter but no setter is known as a read-only computed property
  - We can remove the get keyword and its braces

```swift
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}

let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// prints "the volume of fourByFiveByTwo is 40.0"
Property Observers

- They observe and respond to changes in property values
  - They are called every time the property’s setter is called
  - cannot be added to lazy properties
- One can implement
  - willSet - called just before the value is stored
  - didSet - called immediately after the new value is stored

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

let stepCounter = StepCounter()
stepCounter.tSteps = 200 // About to set tSteps to 200
// Added 200 steps
stepCounter.tSteps = 360 // About to set tSteps to 360
// Added 160 steps
stepCounter.tSteps = 896 // About to set tSteps to 896
// Added 536 steps
```
Type Properties (static)

• They belong to the type and not to a specific instance!
• For computed type properties for class types, we can use the class keyword instead to allow subclasses to override the superclass's implementation

```swift
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}

class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
Methods

- Methods are functions that are associated with a particular type
  - Classes, structures, and enumerations can all define methods
  - Support for instance methods and type methods

```swift
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func incrementBy(_ amount: Int) {
        self.count += amount
    }
    func reset() {
        count = 0
    }
}

let counter = Counter()
counter.increment()  
counter.incrementBy(5)  
counter.reset()
```
Modifying Value Types from Within Instance Methods

- Structures and enumerations are value types
- By default, the properties of a value type cannot be modified from within its instance methods
  - Unless the method is identified as mutating
  - The method assigns a completely new instance to its implicit self property

```swift
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(x: 2.0, y: 3.0)
print("The point is now at ((somePoint.x), (somePoint.y))")
// prints "The point is now at (3.0, 4.0)"
```
Type Methods

• We indicate type methods for classes by writing keyword `class` before `func`

• We indicate type methods for structures by writing keyword `static` `func`

• Self becomes a reference to the type itself, and not to a particular instance

• Calls to these methods are made directly to the type and not to an instance
Subscripts

- Subscripts are shortcuts for accessing the member elements of a collection, list or sequence
  - Access through indexes and not specific getters and setters
  - Multiple subscripts for a type can coexist, and the correct subscript is chosen based on the type of index value passed
  - Subscripts can have multiple dimensions (not just a single index parameter)
- They can be used to query an instance
  - Introduced by keyword subscript

```swift
subscript(index: Int) -> Int {
    get {
        // return an appropriate value
    }
    set(newValue) {
        // perform a setting action here
    }
}

struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}

let threeTTable = TimesTable(multiplier: 3)
print("six x three is \(threeTTable[6])")
// prints "six x three is 18"
```
Example

```swift
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}

var matrix = Matrix(rows: 2, columns: 2)

matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```
Inheritance

- Swift classes do NOT inherit from a universal base class
  - Every class that does not inherit from another class becomes a base class

```swift
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing — an arbitrary vehicle doesn't necessarily make a noise
    }
}

class Bicycle: Vehicle {
    var hasBasket = false
}

class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```
Overriding

- Overriding
  - The method should be introduced by keyword `override`
  - can be prevented using keyword `final`
- Accessing the superclass is done using keyword `super`
  - An overridden method named `someMethod` can call the superclass version of `someMethod` by calling `super.someMethod()` within the overriding method implementation
  - An overridden property called `someProperty` can access the superclass version of `someProperty` as `super.someProperty` within the overriding getter or setter implementation
  - An overridden subscript for `someIndex` can access the superclass version of the same subscript as `super[someIndex]` from within the overriding subscript implementation
Examples

class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}

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

- Initialization involves setting an initial value for each stored property on the instance and performing any other setup or initialization that is required before the new instance is ready for use
  - Swift initializers do not return a value
- Instances of class types can also implement a deinitializer, which performs any custom cleanup just before an instance of that class is deallocated

```swift
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// prints "The default temperature is 32.0° Fahrenheit"
```
Inizialization

• Optional properties are automatically set to nil during initialization
• Constant properties can be modified at any point during initialization
  – As long as they are set to a definite value by the time initialization finishes
• Default initializers are provided when no other initializers are provided by the programmer
  – Only for base classes
  – The default initializer simply creates a new instance with all of its properties set to their default values
• Structures automatically receive memberwise initializers, if they do not provide their own
  – Initial values for the properties of the new instance can be passed to the memberwise initializer by name
Example

```swift
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```
Initializer Delegation

- Initializers can call other initializers
  - To avoid duplicate code

- The rules are different for value types and class types
  - Value types (structures) do not allow inheritance
    - They can only delegate to initializers that they provide themselves
    - If we define a custom initializer for a value type, we will no longer have access to the default
  - Classes must ensure that all the stored properties they inherit are assigned a suitable value
    - This is achieved through designated initializers and convenience initializers

- Swift subclasses do not inherit their superclass initializers by default
Initializers

- A designated, primary, initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain
  - Every class must have at least one designated initializer, and often it is only one

- Convenience initializers are secondary, supporting initializers for a class
  - Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent

```swift
convenience init(parameters) {
    statements
}
```
Rules for Delegation Calls

- **Rule 1**: A designated initializer must call a designated initializer from its immediate superclass.
- **Rule 2**: A convenience initializer must call another initializer from the same class.
- **Rule 3**: A convenience initializer must ultimately call a designated initializer.

Designated initializers must always delegate up.

Convenience initializers must always delegate across.
Two-phase Initialization

• Each stored property is assigned an initial value by the class that introduced it
• Each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use

• This is to prevent property values
  — from being accessed before they are initialized
  — from being set to a different value by another initializer unexpectedly
Compiler performs 4 safety checks

• A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.
• A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property.
  – If it doesn’t, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.
• A convenience initializer must delegate to another initializer before assigning a value to any property (including properties defined by the same class).
  – If it doesn’t, the new value the convenience initializer assigns will be overwritten by its own class’s designated initializer.
• An initializer cannot call any instance methods, read the values of any instance properties, or refer to self as a value until after the first phase of initialization is complete.
Example

class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}

let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)

class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}

let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// Bicycle: 2 wheel(s)
Automatic Initializer Inheritance

- If we provide default values for any new properties introduced in a subclass
  - **Rule 1**: If the subclass doesn’t define any designated initializers, it automatically inherits all of its superclass designated initializers
  - **Rule 2**: If the subclass provides an implementation of all of its superclass designated initializers, then it automatically inherits all of the superclass convenience initializers
- These rules apply even if your subclass adds further convenience initializers
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]"
    }
}

let namedMeat = Food(name: "Bacon") // namedMeat's name is "Bacon"

let mysteryMeat = Food() // mysteryMeat's name is "[Unnamed]"

class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}

let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
Failable Initializers

- Initialization may fail because of invalid initialization parameter values, absence of a required external resource, or other reasons
  - We write a failable initializer by placing a question mark after the init keyword (init?)
  - We write return nil to indicate a point at which initialization failure can be triggered

```swift
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}

let anonymousCreature = Animal(species:"")
// anonymousCreature is of type Animal?, not Animal

if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}
```
Required Initializers

• The required modifier before the definition of a class initializer indicates that every subclass of the class must implement that initializer
  — The required modifier before every subclass implementation of a required initializer indicates that the initializer requirement applies to further subclasses in the chain
  • We do not use the override modifier when overriding a required designated initializer

```swift
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}

class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required initializer goes here
    }
}
```
Deinitialization

• Called immediately before a class instance is deallocated
  — use the deinit keyword
  — at most 1 deinitializer per class
  — no params and written without parenthesis

• Deinitializers are called automatically
  — Superclass deinitializers are inherited
  — Superclass deinitializers are automatically called after deinitialization of the subclass

• Swift automatically deallocates instances when they are no longer needed
  — achieved through Automatic Reference Counting (ARC)
Extensions

• Extensions add new functionality to an existing class, structure, or enumeration type
• This includes the ability to extend types for which you do not have access to the original source code
  – Add computed instance properties and computed type properties
  – Define instance methods and type methods
  – Provide new initializers
  – Define subscripts
  – Define and use new nested types
  – Make an existing type conform to a protocol
• Extensions can add new functionality to a type, but they cannot override existing functionality
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// prints "One inch is 0.0254 meters"

let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// prints "Three feet is 0.914399970739201 meters"

let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// prints "A marathon is 42195.0 meters long"
Protocols

- A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality
  - Any type that satisfies the requirements of a protocol is said to conform to that protocol
  - The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements
- Protocols can be composed

```swift
protocol SomeProtocol {
    // protocol definition goes here
}

struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
Property Requirements

- A protocol can require any conforming type to provide an instance property or type property with a particular name and type.
- The protocol does NOT specify whether the property should be a stored property or a computed property.
  - It only specifies the required property name and type.
  - The protocol also specifies whether each property must be gettable or gettable and settable.
  - Property requirements are always declared as variable properties, prefixed with the `var` keyword.

```swift
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}

protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```
Example

```swift
protocol FullyNamed {
    var fullName: String { get }
}

struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"

class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```
protocol SomeProtocol {
    static func someTypeMethod()
}

protocol RandomNumberGenerator {
    func random() -> Double
}

class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}

let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.37464991998171"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283"
Initializer Requirements

```swift
protocol SomeProtocol {
    init()
}

class SomeClass: SomeProtocol {
    required init() {
        // initializer implementation goes here
    }
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```
Protocols and Extensions

```swift
protocol TextRepresentable {
    var textualDescription: String {
        get
    }
}

class Dice {
    let sides: Int
    let generator: RandomNumberGenerator

    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }

    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}

extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}

var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
punkt(d6.textualDescription)  // prints "A 6-sided dice"
```
Automatic Reference Counting (ARC)

- Swift tracks and manages memory usage
  - In most cases, this means that memory management “just works” in Swift
  - ARC automatically frees up the memory used by class instances when those instances are no longer needed
  - In a few cases ARC requires more information about the relationships between parts of our code to make sure that instances don’t disappear while they are still needed
- ARC tracks how many properties, constants, and variables are currently referring to each class instance
  - ARC will not deallocate an instance as long as at least one active reference to that instance still exists
  - The reference is called a “strong” reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}

var reference1: Person?
var reference2: Person?
var reference3: Person?

reference1 = Person(name: "John Appleseed")
// prints "John Appleseed is being initialized"

reference2 = reference1
reference3 = reference1

reference1 = nil
reference2 = nil
reference3 = nil
// prints "John Appleseed is being deinitialized"
• Is it possible to never reach a point that we have zero references?
  – It can happen if two class instances hold a strong reference to each other
  – We resolve strong reference cycles by defining some of the relationships between classes as weak references
Weak References

- Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime
  - It is appropriate for an apartment to be able to have “no tenant” at some point in its lifetime
  - Because weak references are allowed to have “no value”, every weak reference is an optional type
  - ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated

```swift
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```
Unowned References

• Conversely, use an unowned reference when you know that the reference will never be nil once it has been set during initialization
  – An unowned reference is assumed to always have a value
  – It is always defined as a non-optional type
  – We don’t need to unwrap the unowned reference each time it is used
Access Control

- Swift’s access control model is based on the concept of modules and source files
  - A module is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift’s import keyword
  - A source file is a single Swift source code file within a module (in effect, a single file within an app or framework)
- Swift supports three access levels
  - Public access enables entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module
  - Internal access enables entities to be used within any source file from their defining module, but not in any source file outside of that module
  - Private access restricts the use of an entity to its own defining source file
Guiding Principals

• No entity can be defined in terms of another entity that has a lower (more restrictive) access level
  – For example, a function cannot have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are not available to the surrounding code

• All entities in your code have a default access level of internal if you do not specify an explicit access level yourself
  – Define the access level for an entity by placing one of the public, internal, or private modifiers before the entity

• Some rules
  – The access level for a tuple type is the most restrictive access level of all types used in that tuple
  – The access level for a function type is calculated as the most restrictive access level of the function’s parameter types and return type
  – A subclass cannot have a higher access level than its superclass, but an override can make an inherited class member more accessible than its superclass version
Grand Central Dispatch (GCD)

- A queue is actually a block of code that can be executed synchronously or asynchronously, either on the main or on a background thread.
- Once a queue is created, the operating system is the one that manages:
  - Queues are following the FIFO pattern (First In, First Out)
  - A queue can be either serial or concurrent.
- A work item is a block of code that is either written along with the queue creation, or it gets assigned to a queue and it can be used more than once (reused):
  - The execution of work items in a queue also follows the FIFO pattern.
  - This execution can be synchronous or asynchronous.
let queue = DispatchQueue(label: "com.app.queue")

queue.async {
    print("Dispatched to background queue")
}

DispatchQueue.main.async {
    print("main thread dispatch")
}

DispatchQueue.global(qos: .userInitiated).async {
    print("user initiated task")
}

let block = DispatchWorkItem {
    print("do something")
}

DispatchQueue.main.async(execute: block)