Properties

Lazy stored properties

```
class DataImport {
    var fileName = "data.txt" }
class DataMan {
// Must have a default initializer
    @lazy var importer = DataImport()
    var data = String[]() }
let manager = DataMan()
manager.data += "Some data"
// Data import still not loaded
println(manager.importer.fileName)
// Now data import loaded.
```

Computed properties

```
struct Point { var x = 0.0, y = 0.0 }
struct Size  { var w = 0.0, h = 0.0 }
struct rect {
    var origin = Point()
    var size   = Size()
    var center: Point {
    get {
        let cX = origin.x + (size.w / 2)
        let cY = origin.y + (size.h / 2)
        return Point(x: cX, y: cY) }
    set(newC) {
        origin.x = newC.x - (size.w / 2)
        origin.y = newC.y - (size.h / 2) } }
    var area: Double {
        return size.w * size.h } }
```

Type Properties

Similar to static types or class variables. Stored type properties are only available in Structs and Enums, computed type properties available in Classes, Structs and Enums. Stored type properties must have an initializer (because the type itself is never initialized)

```
struct Something {
    static var storedTP = 123
    static var computedTP: Int {
        return 1234 } }
class SomethingElse {
    class var newSomething: SomethingElse {
        return SomethingElse() } }
SomethingElse.newSomething
Something.storedTP
```

Optionals

An optional models the idea that a variable can take on a specific value, or none at all. For example, to convert a string to an int may yield an integer, or have no meaning at all. Several optionals can be chained into one expression.

```
let noValue: Int? = nil
let text      = "123"
let num: Int? = text.toInt()
if num { println("Value: \(num!)") }
else   { println("Failed") }

if let actualNum = text.toInt() {
    println("Value: \(actualNum)") }
else { println("Failed") }

let iplString: String! = "implicitly unwrapped"
println(iplString) // No '!' required
```

Functions

```
// The numbers parameter is internal/external
func mean(#numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers { total += number }
    return total / Double(numbers.count) }
// inout parameters change the passed-in vars
var int1 = 1, int2 = 2
func swapInts(inout a: Int, inout b: Int) {
    let tempA = a; a = b; b = tempA }
swap(&int1, &int2)
// Function types can be return values
var stepFunc: (Int) -> Int
func getStepFunc(back: Bool) -> (Int) -> Int {
    // Functions can be nested and returned
    func stepFwd (i: Int) -> Int {return i+1}
    func stepBack(i: Int) -> Int {return i-1}
    return back ? stepBack : stepFwd }
stepFunc = getStepFunc(false)
```

Closures (reference)

```
sort(names, {
    (s1: String, s2: String) -> Bool in
    return s1 > s2 })
sort(names) { s1, s2 in return s1 > s2 }
// Shorthand argument names & implicit return
sort(names) { $0 > $1 }
// The > operator matches the signature
sort(names, >)
```

Extensions

Extensions can add functionality to Classes, Structs and Enums. They can add computed (may be static) properties, Instance and type methods, convenience initializers, subscripts, nested types and protocol conformance. Changes to types are available on all instances, even those created before the extension was defined. The syntax for things in an extension is virtually identical to the syntax used elsewhere.

```
extension SomeType: ProtocolOne, ProtocolTwo {
    var mass: Double { return self + 42.0 }
    init(novelInit: String) {
        println("This init is a convenience")
        self.init() }
    mutating func square() { mass *= mass } }
```

ARC

Automatic Reference Counting is used to maintain class lifetime. Provided that strong reference cycles don't occur it will work without having to think about it. These cycles can be avoided with the *weak* or *unowned* keywords. Use the weak keyword when a property can be *nil* at any point and must be an *optional*. ARC will automatically set *weak* references to *nil* when the referred object is deallocated. Use *unowned* when the value will remain set for its lifetime. It's possible to create cycles using closures, see the language guide for details.

Strings (value)

```
let string = "Some string: \"Hello.\""
let dollar = "\x24"
let heart1 = "\u2665"
let heart2 = "\U0001F496"
var empty  = String() // or ""
let dogcow = "🐶🐮"
let yen: Character = "¥"
let moof   = "\(dogcow) says \"Moof!\""
```

Use .countElements to get the character count (unicode characters != bytes); this may be expensive.

Arrays (see structs)

Substantially similar to NSArray. Constant arrays are constant only in size. Arrays are technically pass-by-value, but only copied when the size changes. To explicitly copy use '*unshare*'. Use *instance equality* to test sharing.

```
var array1: Array<Int> = [0, 1, 2]
var array2: Double[]   = [0, 1, 2]
array1 += array2.map({ return Int($0)})
array2.append(4)
// These can be different sizes...
array2[0..array2.count] = [0.0, 0.1, 0.2]
array1.insert(5, atIndex: 1)
array1.removeAtIndex(0)
for (index, value) in enumerate(array2) {
    println("index \(index) is \(value)") }
for value in array1 { println("\(value)") }
var a3 = Int[](count: 10, repeatedValue: 0)
```

Dictionaries (see structs)

Assigning or passing a dict will copy all the keys and values. Value types are copied and references to reference types are copied.

```
var airports = Dictionary<String, String>()
airports = ["TYO": "Tokyo","DUB": "Dublin"]
airports["LHR"] = "London"
if let old = airports.updateValue("Heathrow",
    forKey: "LHR") { println("LHR was \
(old)") }
airports["TYO"] = nil
for (code, name) in airports {
    println("\(code) is \(name)")
}
```

Type Casting

```
var things = Any[]()
things += 0
things += 0.0
things += "hello"
things += (3.0, 5.0)
for thing in things {
    switch thing {
    case 0 as Int: println("0 as int")
    case 0 as Double: println("0 as fp")
    case is Double: println("A double")
    case let (x, y) as (Double, Double):
        println("Double tuple")
    case let str as String:
        println("A string: \(str)")
}
```

Classes (reference)

```
class SuperClass {
    var instanceVariable: Int
    let constant = 123
    init() {
        constant = 1234
        instanceVariable = constant }
    @final func cantChange() -> String {
        return "Un-overridable!" } }
class ClassName: SuperClass, protocolOne {
    var otherVariable: Double {
    willSet {
        instanceVariable = Int(newValue) }}
    override var instanceVariable: Int {
    didSet {
        otherVariable = Double(oldValue) }}
    init(doubleValue: Double) {
        otherVariable = doubleValue
        super.init() // before inherited vars
        instanceVariable = Int(doubleValue) }
    class func printHello() {
        println("Hello") } }
```

Struct (value)

Structs are mostly similar to classes, however, structs are always copied and do not participate in reference counting. Structs can't be inherited or type-cast.

```
struct Resolution {
    var width: Int; var height: Int
    func pixels() -> Int { return
width*height }
    mutating func standard(std:String) {
        switch std {
        case "HD": width = 1920; height = 1080
        case "SD": width = 640; height = 480
        default: return
        } } }
var hd = Resolution(width: 1920, height: 1080)
var cinema = hd; cinema.width = 2048
// Cinema is now 2048x1080, hd unchanged.
hd.standard("SD")
```

Arrays and Dictionaries are implemented as structs.

Enum (value)

```
enum Barcode {
    case UPCA(Int, Int, Int)
    case QRCode(String)
    func print() -> () {
        switch bc {
        case .UPCA (let sys, let id, let crc):
            println("\(sys) \(id) \(crc)")
        case .QRCode (let code):
            println("QRCode: \(code)") } } }

var bc = Barcode.UPCA(8, 85909_51226, 3)
bc = Barcode.QRCode("ABCDEFGHIJKLMNOPQ")
bc.print()

enum Planet: Int {
    case mercury = 1, Venus, Earth, Mars,
    Jupiter, Saturn, Uranus, Neptune
    mutating func goHome() {
        self = .Earth } }
var mars! = Planet.fromRaw(4)
println("Earth is \(Planet.Earth.toRaw())th")
println("The 4th planet is \(mars)")
mars.goHome()
```

Operators

Unary operators

+	— Returns the value with no change
-	— Returns the negated value
~	— Bitwise negation
++/--	— [in/de]crements the value
[var]	— Subscript (index or key)
var()	— Function call/dereference
as	— Forced downcast
as?	— Optional downcast

Binary operators

+,-,/,*	— Add, Sub, Mult. and Divide
%	— Remainder (r), a = (b×x) + r works correctly with floating point
=	— Assigns the lhs with the value May composite with most binary ops
&&	— Logical AND operator
&	— Bitwise AND (others same as C)
||	— Logical OR operator
is	— Type introspection

Comparison operators

==	— Equal (Equatable types)
!=	— Not equal (Equatable)
<=	— Less-than or equal (Comparable)
>=	— Greater-than or equal (Comp.)
<	— Less-than (Comparable)
>	— Greater-than (Comparable)
===	— References to the same instance
!==	— Not the same instance

Ternary comparison operator

```
let result = (a < b ? trueVal : falseVal)
```

Range operators

Return an array of numeric values interpolated between the provided values. The closed range includes both the first and last value, half-closed range includes only the first.

0...10	— Closed range operator
0..10	— Half-closed range operator

Operator functions (custom operators)

```
// Define these in the global scope
// See language guide for assoc. and precedence
struct Vec2 { var x = 0.0, y = 0.0 }
@infix func + (l: Vec2, r: Vec2) -> Vec2 {
    return Vec2(x: l.x+r.x, y: l.y + r.y)}
@prefix func - (v: Vec2) -> Vec2 {
    return Vec2(x: -v.x, y: -v.y) }
@assignment func += (inout l: Vec2, r: Vec2) {
    l = l + r }
@prefix @assignment func ++ (inout v: Vec2)
                            -> Vec2 {
    v += Vec2(x: 1.0, y: 1.0)
    return v }
@infix func == (l: Vec2, r: Vec2) -> Bool {
    return (l.x == r.x) && (l.y == r.y) }
```

Types

Built-in types

Int	— Signed integer, word sized
UInt8, UInt16, UInt32, UInt64	— Unsigned ints
Int8, Int16, Int32, Int64	— Signed integers
Float	— Single-precision (32-bit)
Double	— Double-precision (64-bit)
Bool	— Boolean value (NON INTEGER!)
String	— String of unicode characters
Any	— Any non-function type instance
AnyObject	— Any class type

Values in Swift are not permitted to overflow their containing type unless the overflow operator *&* is used (as in a &+ b).

Floating point initialization

fp = 1.25e2	— 125.0	— 1.25×10^{2}
fp = 1.25e-2	— 0.0125	— 1.25×10^{-2}
fp = 0xFp2	— 60.0	— 15×2^{2}
fp = 0xFp-2	— 3.75	— 15×2^{-2}
fp = 0xC.3p0	— 12.1875	— 12.1875×2^{0}
fp = 1_000	— 1,000	— 1×10^{3}

Math across types

Swift will not automatically change types for mathematical or assignment operations.

```
let twoK:    UInt16 = 2_000
let one:     Int8   = 1
let twoKOne: Float = Float(twoK + UInt16(one))
```

Floating point numbers are always truncated toward zero to make integer values, 4.75 becomes 4 and -3.9 becomes -3. Numeric literals do not have a type, and can be used in any assignment or operation.

Type Aliases

Type aliases are substantially similar to typedefs:

```
typealias AudioSample = UInt16
```

Tuples

Tuples are individual values grouped together. The components can be any type and have any number of elements. Tuples are useful when returning many elements from a function.

```
let http404Error = (404, "Not Found")
let (code, msg)  = http404Error
println("Status code is: \(code)")
println("Status message: \(msg)")
let (justTheCode, _) = http404Error
println("Status message: \(http404Error.1)")
let http200Status = (code: 200, desc: "OK")
```

Nesting types

It is possible to define any type within another type. To access nested types, chain them.

Methods

Within methods, the *self* keyword can be used to access the instance for the method call, see the enum section.

See the Struct section for methods that change the parameters in a value type. The *mutating* keyword indicates this intention. Constant instances don't allow these methods.

Type methods are possible with Classes, Structs and Enums. In a type method, the *self* keyword refers to the type itself not an instance.

Parameter Names

Names for the parameters to methods can have external and internal names. The first parameter is local only by default, subsequently, they are both. Use the '#' symbol to indicate internal/external, see the functions section. You can provide different names for internal and external use as below:

```
func function(external internal: Int)
```

Subscripts

To implement a function that implements the subscript operator use the *subscript* keyword.

Multiple implementations can be provided, they are selected based on the type signature of the index. Subscripts can contain a number of inputs for the index, including a *variadic*.

Subscripts can be read-write or read-only, this is in reference to the value being selected. If it's read-only it can be simplified as in the area computed property in properties.

```
subscript(index: Int) -> Int {
    get { /* return value selected */ }
    set { /*  perform set action   */ }
}
```

Initialization (init)

It is possible to provide several *init*s that take a different inputs parameters. Designated *init*s must call or override *super.init*. Convenience *init*s must call a designated *init* of the same class. A subclass' *init* must *init* local properties first, call *super.init*, then may change the value of inherited properties. Convenience *init*s must call the designated *init* before changing any instance properties.

Deinitialization (deinit)

Cleanup after the class before removal. Generally not for memory. Writing to file and closing network connections, etc.

Protocols

A protocol describes a minimum interface to a Class, Struct or Enum. See the Class section for example syntax for adopting a protocol. Protocol compliance is checked at compile time. Protocols are types and used as parameter type, type of a variable constant or property, type used in a collection type. The *mutating* keyword is only appropriate with Structs and Enums.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var mayNotBeSettable: Double { get}
    class var typeProp: Int { get set }
    class func typeMethod(name: String)
    @optional func random() -> Double
    mutating func toggle()
}

protocol inheriting: PrtclOne, PrtclTwo {
    // If additional requirements are
desired...
}
```

In the case where compliance with more than one protocol is required, they can be composited with *protocol<>*.

```
var compositing: protocol<PrtclOne, PrtclTwo>
```

Use the type-casting operators to check for protocol compliance, but only if prefixed with @objc, @optional is only available in this mode.

Generics

Allows general code to be written that can work with "type placeholders." The <T> syntax means that the stack type is stored in T.

```
struct Stack<T> {
    var items = T[]()
    mutating func push(item: T) {
        items.append(item) }
    mutating func pop() -> T {
        return items.removeLast() } }
```

It may be necessary to constrain the types used. Include the constraints after *where* in the angle braces:

```
func anyCommonElements
       <T, U where
       T: Sequence, U: Sequence,
       T.GeneratorType.Element: Equatable,
       T.GeneratorType.Element ==
          U.GeneratorType.Element>
       (lhs: T, rhs: U) ->
         Array<T.GeneratorType.Element> {
    var ret = Array<T.GeneratorType.Element>()
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                ret.append(lhsItem) } } }
    return ret }
anyCommonElements([1, 2, 3], [3])
```

Associated type are omitted