# การเขียนโปรแกรมโดยใช้ภาษา SWIFT

**โดย**

**ผศ.ดร. ปิยะ เตชะธีราวัฒน์**

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยธรรมศาสตร์

พ.ศ. 2560

# คำนำ

ตำราฉบับนี้ได้จัดทำขึ้นเพื่อใช้ในการเรียนการสอนรายวิชา Mobile Application, Software Engineering และวิชาที่เกี่ยวข้องภายใต้หลักสูตร Computer Engineering และ Software Engineering รวมถึงสำหรับผู้ที่ต้องการต่อยอดในการเขียน Mobile Application ผ่านภาษา SWIFT ที่เน้นการใช้งานในระบบ iOS และมีจุดประสงค์ในการให้แอปพลิเคชันใช้งานในผลิตภัณฑ์ของ Apple เช่น iPhone และ iPad

จุดมุ่งหมายสำหรับผู้ใช้ที่มีพื้นฐานด้านการเขียนโปรแกรมเบื้องต้น และต้องการต่อยอดสำหรับภาษา SWIFT เพื่อประยุกต์ใช้กับการพัฒนา Mobile Application หรือ การวางโครงสร้างของโปรแกรมด้วยแนวคิดแบบภาษาอื่นๆ เพื่อใช้ในการเรียนการสอนให้มีประสิทธิภาพต่อไป

ทั้งนี้จุดมุ่งหมายในการถ่ายทอดความรู้ผ่านตำราฉบับนี้เน้นหลักการให้สามารถไปประยุกต์กับโจทย์และแบบฝึกหัดดังที่แสดงไว้ในส่วนท้ายของเอกสารฉบับนี้ อย่างไรก็ตามการพัฒนาของภาษาจะมีการพัฒนาตามเทคโนโลยี โดยเอกสารฉบับนี้พัฒนาขึ้นในปี 2558 และได้ปรับปรุงใน ปี พ.ศ. 2560 หากต้องการตรวจสอบกับเวอร์ชันล่าสุดขอให้ตรวจสอบกับผู้พัฒนาต่อไป

# สารบัญ

# The Basic

**เครื่องมือในการพัฒนา**

ในการเตรียมเบื้องต้น เตรียมเครื่องคอมพิวเตอร์ในระบบ OSX และอุปกรณ์ที่ใช้ในการทดสอบแอพพลิเคชั่นแล้ว ต้องทำการดาวน์โหลด Xcode ซึ่งเป็นกลุ่มของซอฟต์แวร์ที่ออกแบบมาสำหรับการพัฒนาแอพพลิเคชั่นโดยเฉพาะจาก App Store เมื่อต้องการเริ่มสร้างโปรแกรมด้วยเครื่องแมคอินทอชให้ทำตามขั้นตอน [1-3] ดังต่อไปนี้

1. คลิกที่ไอคอน Launchpad
2. คลิกที่ Xcode
3. จะพบหน้าต่าง Welcome Screen สามารถทดสอบโค้ดด้วย playground, สร้างโปรเจคใหม่ และเปิดไฟล์เก่ามาแก้ไขใหม่ได้จากหน้าต่างนี้
4. คลิกที่ Get started with a playground แล้วเลือกประเภทอุปกรณ์ที่ต้องการพัฒนาแอพพลิเคชั่น
5. เลือกเทมเพลทเป็น Blank ตั้งชื่อไฟล์และเลือกที่เก็บบันทึก
6. จะพบตัวอย่างโค้ดคำสั่งที่ฝั่งซ้ายมือ และแสดงผลลัพธ์ที่ฝั่งขวามือ ให้ทดลองเปลี่ยนโค้ดในบรรทัดที่ 3 จาก var str = “Hello, playground” เป็น var str = “Hello, Swift” ก็จะพบว่าที่ด้านขวามือมีการแสดงผลอัตโนมัติ
7. หากต้องการทดสอบโค้ดด้วยเครื่องพีซี สามารถ Run แอพพลิเคชั่นออนไลน์ในเว็บไซต์ได้ ในหลายไซต์ เช่น https://iswift.org/playground สามารถเลือกเวอร์ชั่นของ Swift และพิมพ์ทดสอบโค้ดในด้านซ้าย แล้วกด Run เพื่อแสดงผลในด้านขวา ได้

**หลักการในการเขียนโปรแกรม**

Swift เป็นภาษาที่ใช้ในการเขียนโปรแกรมสำหรับ iOS และ OSX มีความคล้ายคลึงกับภาษา C และ Objective C ซึ่งชนิดของข้อมูลใน Swift [4-7] ได้แก่

- *Int*
- *Double*
- *Float*
- *Bool*
- *String*
- *Collection Type*
- *Array*
- *Dictionary*

ส่วนที่เพิ่มเข้ามาใน Swift คือ

Truples คือ Group ค่าที่มีชนิดข้อมูลต่างกันได้

Optionaltype คือ คล้ายกับ pointer

**Constant และ Variable**

- *Constant คือ ตัวแปรค่าคงที่ ไม่สามารถเปลี่ยนแปลงได้*
- *Variables คือ ตัวแปรทั่วไป เปลี่ยนแปลงค่าได้*

Let ใช้ประกาศตัวแปร Constant คือ let ชื่อตัวแปร = ค่า

Var ใช้ประกาศตัวแปร variable คือ var ชื่อตัวแปร = ค่า

## ประเภทตัวแปร (Type Annotation)

Annotation คือ การระบุว่าตัวแปรนั้นๆเก็บข้อมูลชนิดไหน โดยระบุหลังเครื่องหมาย : (colon)

เช่น var welcome : String

หมายความว่า ตัวแปร welcome เก็บข้อมูลชนิด String

## การตั้งชื่อ Constant และ Variable (Naming Constants and Variables)

สามารถประกอบไปด้วยอักษร รวมถึงอักขระใน Unicode Characters ได้โดยยกเว้น

- *เครื่องหมายทางคณิตศาสตร์*
- *ลูกศร*
- *จุด*
- *เส้น หรือรูปวาดที่อยู่ในตัวอักษร*
- *การขึ้นต้นด้วยตัวเลข*

หากมีการประกาศชื่อตัวแปรนั้นไปแล้ว จะไม่อนุญาตให้ประกาศซ้ำกันอีกและไม่อนุญาติให้เปลี่ยนแปลงชนิดข้อมูลได้ ดังนั้นการเปลี่ยนแปลง Constant -> Variable หรือ variable -> Constant จึงไม่สามารถทำได้

## การพิมพ์ผลลัพธ์ Constant และ Variable (Printing Constants and Variables)

ใช้ print แล้วตามด้วย (ชื่อตัวแปร) ซึ่งจะพิมพ์พร้อมขึ้นบรรทัดใหม่ให้ ถ้าไม่ต้องการขึ้นบรรทัดใหม่ให้ใช้คำสั่ง print(“ข้อความ”, terminator:””)

การสั่ง print String ให้ใช้ “ ” ร่วมด้วย

```
print(“สวัสดี”)
```

```
Var welcome: String = “สวัสดี”
print(“คำทักทายในภาษาไทยคือ \(welcome)”)
```

**Comment**

Comment คือข้อความอธิบายโค้ด ซึ่งจะไม่ส่งผลกระทบกับ code

*// ->สำหรับ 1 บรรทัด*

*/*...*/  -> สำหรับหลายบรรทัด (... ใส่ข้อความที่ต้องการ comment )*

**Semicolon (;)**

ไม่ได้มีไว้เพื่อจบท้ายคำสั่ง แต่มีไว้เพื่อคั่นคำสั่งหลายๆคำสั่งใน 1 บรรทัด

```
let lemon = “melon” ; print(lemon)
```

**Integers**

มี 2 แบบ ได้แก่

1. Signed เป็นบวก ศูนย์ หรือลบก็ได้
2. Unsigned เป็นบวก ศูนย์ ไม่มีค่าลบ

เราสามารถกำหนดขนาดของ integer ให้มีขนาด 8,16,32 และ 64 bit ได้

*8-bit Signed Integer คือ Int8*

*32-bit Unsigned Integer คือ UInt32*

**Integer Bound**

หมายถึง ขอบเขตของค่าในแต่ละขนาด ใช้ .min และ .max ในการกำหนด

```
let minvalue = UInt8.min     // minvalue = 0
let maxvalue = UInt8.max     // maxvalue = 255
```

หากไม่ต้องการกำหนดขอบเขตของตัวแปร Integer ก็สามารถกำหนดเป็น Int ได้เนื่องจากใน 32-bit platform Int นั้น สามารถเก็บค่าได้ในช่วง -2,147,483,648 ถึง 2,147,483,647 ซึ่งมากพอที่จะเก็บค่า Integer

คือตัวเลขที่มีจุดทศนิยม เช่น 0.1, 38.26 หรือ -2.7

- *Double มีขนาด 64 bit*
- *Float มีขนาด 32 bit*

*การใช้ Float หรือ Double ขึ้นอยู่กับขนาดของข้อมูลที่ต้องการใช้งานและความถูกต้องในการแสดงทศนิยมกี่ตำแหน่งด้วย*

## Numerical Literals

- **Integer literals**

  *เลขฐาน 10 ไม่มี prefix*

  *เลขฐาน 2 มี 0b เป็น prefix*

  *เลขฐาน 8 มี 0o เป็น prefix*

  *เลขฐาน 16 มี 0x เป็น prefix*

```
let dInt =17
let bInt = 0b10001
let oInt = 0o21
let hInt = 0x11
```

เลขทั้งหมดที่เขียนมาทั้งหมดมีค่า เท่ากับ 17

- **Floating-Point literals**

- Decimal หลัง e คือ เลขยกกำลัง 10exp

1.25e2 คือ $1.25 \times 10^{2}$ หรือ 125.0

1.25e-2 คือ $1.25 \times 10^{-2}$ หรือ0.0125

- Hexadecimal หลัง p คือ เลขยกกำลัง 2exp

0xFp2 คือ $15 \times 2^{2}$ หรือ 60.0

0xFp-2 คือ $15 \times 2^{-2}$ หรือ 3.75

- **Numeric literals**

สามารถใส่ 0 ไว้ข้างหน้า และ เติม underscores(_) เพื่อให้ง่ายต่อการอ่านไม่ส่งผลกระทบต่อคำ

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
```

## Type Safety และ Type Inference

การเขียนโปรแกรมในแบบ Type Safety นั้น หากได้กำหนดประเภทของตัวแปรไปแล้ว จะไม่สามารถนำข้อมูลแบบอื่นใส่ลงไปได้ และ Type Interference คือการตรวจสอบตัวแปรว่ามีการกำหนดข้อมูลแบบใดในครั้งแรก ซึ่งวิธีการทั้งสองจะช่วยลดข้อผิดพลาดในการเขียนโปรแกรมได้มาก

```
var SLocation = “Bangkok”
var SBranch = 6
SLocation = “Krabi”
SBranch = 8.57 // Error เนื่องจากในครั้งแรกเป็น Int ไม่ใช่ Double
```

## การเปลี่ยนแปลงประเภทตัวเลข (Numeric Type Conversion)

- **Integer Conversion**

```
let a: UInt16 = 2_000
let b: UInt8 = 1
let c = a + b  //จะ error เพราะ a กับ b เป็นข้อมูลคนละชนิด
```

แก้ไขได้โดยใส่ UInt16 ไปไว้ข้างหน้า b จะสามารถทำให้ a บวกกับ b ได้

```
let c = a + UInt16(b)
```

และ c จะมีข้อมูลเป็น UInt16

- **Integer และ Floating-Point Conversion**

ทำให้ integer และ float บวกกันได้

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
```

*pi จะมีชนิดข้อมูลเป็น Double*

| ชนิดข้อมูล (ชื่อตัวแปร) |
|---|

## Type Aliases

การประกาศ Type เพื่ออ้างถึง Type ที่มีอยู่แล้ว ใช้ typealias ในการประกาศ

```
typealias sample = UInt16
var a = sample.min
        //a = 0
var b = sample.max
        //b = 255
```

**Booleans**

true หรือ false

การประกาศตัวแปร Booleans

*let rosesAreRose = true*

*let flowersAreFragrant = false*

ตัวอย่าง

```
if flowersAreFragrant {
print("Flowers smell good!")
} else {
print("Flowers are smelly.")
}
// ผลที่ได้จะพิมพ์ Flowers are smelly.
let i = 1
if i {
print("Flowers smell good!") // error เพราะ i ไม่ได้เป็นตัวแปร boolean
}
ควรแก้เป็น  if i ==1
```

```
let http404Error = (404, "Not Found")
        404 → Int
        Not Found → Sting
        http404Error จะมี type เป็น (Int, String)
let (a , b) = http404Error
let (c , _ ) = http404Error // c=404
```

- เราสามารถเข้าถึงค่าใน tuple โดยใช้ index โดยตัวแรกจะมี index เป็น 0

```
let http404Error = ( 404 , “Not Found” )
                print( “a is \(http404Error.0)” )
                print( “b is \( http404Error.1)”)
```

- เราสามารถกำหนดชื่อของแต่ละชนิด element ใน tuple ได้

```
let http200Status = (statusCode: 200, description: "OK")
                print("The status code is \(http200Status.statusCode)")
                        // ผลที่ได้จะพิมพ์ The status code is 200
```

**Optionals**

ใช้ Optionals ในเหตุการณ์ที่ค่าของตัวแปรอาจจะมีหรือไม่มีอยู่ กำหนด Optional ด้วยการใช้ ? ตามหลัง Type และสามารถนำไปตรวจสอบเงื่อนไขซึ่งมีสิ่งสำคัญอีกก็คือ การนำ Optional ที่มีค่าแน่นอนมาใช้ต้องใส่เครื่องหมาย ! หลัง optional นั้นเสมอ

```
var myPhone = "66123456789"
var myInt: Int? = Int(myPhone)
if myInt != nil {
        print("หมายเลขโทรศัพท์ +\(myInt!)")
}else {
        print("ไม่พบหมายเลขโทรศัพท์")
```

ซึ่งผลลัพธ์ที่ได้ จะแสดง “หมายเลขโทรศัพท์ +66123456789”

**Optional Binding**

```
        if let constantName = someOptional {
        statements
        }
```

หากสามารถ conversion ได้สำเร็จ constantName จะมีค่าเท่ากับค่าที่ถูกเปลี่ยน และได้นิพจน์เป็น true

**Nil**

Set ค่าให้ว่าง

```
var serverResponseCode: Int? = 404
serverResponseCode = nil      //ไม่มีค่าเก็บอยู่ในตัวแปรนี้แล้ว
```

**Implicitly Unwrapped Optionals**

เราสามารถเขียน Implicitly Unwrapped Optionals ได้ โดยใช้ (String!) หรือ (String?) ตามด้วย type ที่ต้องการทำ optional

```
let possibleString: String? = "An optional string."
print(possibleString!)  // เพื่อให้เข้าถึงค่า

let assumedString: String! = "An implicitly unwrapped optional string."
print(assumedString)      //  ไม่ต้องมี ! ก็เข้าถึงค่าได้
```

## Basic Operators

Operator เป็นสัญลักษณ์พิเศษหรือชุดคำสั่งที่เราใช้เพื่อตรวจสอบ, เปลี่ยน หรือ รวมค่า เช่น Add Operator (+) สำหรับรวมตัวเลข2ชุดเข้าด้วยกัน ตัวอย่าง let I =1+2

And Operator (&&) สำหรับตรวจสอบค่าความจริง ซึ่งจะให้ผลเป็นจริงเมื่อค่าทั้งหมดเป็นจริง

Increment Operator (++) สำหรับแก้ไขค่าโดยเพิ่มค่าเข้าไป 1 ตัวอย่าง ++i = เพิ่มค่า I ไป 1

Swift รองรับ operators มาตรฐานของภาษา C และพัฒนาให้ดีขึ้น โดยสามารถจัดการกับ code ที่ errors ได้ การใช้เครื่องหมาย (=) จะไม่ส่งค่ากลับและใช้เพื่อกำหนดค่าเท่านั้น หากต้องการตรวจสอบค่าจะใช้เครื่องหมาย (==) แทน การใช้เครื่องหมายทางคณิตศาสตร์(+,-,*,/,%) รวมกับ overflow เพื่อป้องกันผลลัพธ์ที่อาจจะเกิด overflow ขึ้น สามารถใช้ overflow operators ของ swift ได้ [8-10]

Swift สามารถใช้(%) กับ float ได้สามารถใช้ two range operators เช่น (a..<b และ a…b) ที่ไม่มีในภาษา C ได้

### Terminology

- Unary operators ที่มีเพียง 1 ตัวแปรเช่น (-a) , !b
- Binary operators ที่มี 2 ตัวแปร เช่น (2+3)
- Ternary operators ที่มี 3 ตัวแปร มีเพียง 1 operator คือ (a ? b :c )

### Assignment Operator

Operator สำหรับกำหนดค่า เช่น

```
let b=10
var a=5
a=b
```

```
let (x,y) = (1,2)
// x = 1, y = 2
```

ไม่เหมือนใน C assignment operator ไม่ return ค่ากลับ

```
if x=y {
// error ไม่สามารถใช้ในการตรวจสอบเงื่อนไขได้เนื่องจากไม่มีการ return ค่ากลับ
}
```

เอาไว้ป้องกันเวลา (=) บังเอิญถูกใช้แทน (==)

**Arithmetric Operators**

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

```
1+2 //equals 3
5-3 //equals 2
2*3 // equals 6
10.0/2.5 //equals 4.0
```

ซึ่ง Arithmetric Operators ใน swift ไม่เหมือนกับ C arithmetric คือไม่อนุญาตให้ใช้ค่า overflows (เกินขอบเขตของ type) แต่สามารถใช้ overflow operators ได้

เช่น (a &+ b) เพื่อทำการอนุญาตให้ overflow ได้

```
Note “hello , “+”world” // equals “hello , world”
```

**Remainder Operator**

Remainder operator (a % b) ใช้บอกว่า b ต้องคูณกี่ครั้งถึงจะใกล้เคียงหรือเท่ากับ a และ return ส่วนที่เหลือออกมา

เช่น

*9%4 //remainder มีค่าเท่ากับ 1*

เพื่อกำหนดค่า a%b , เราสามารถคำนวณได้จาก

*A = (b*ค่าที่นำไปคูณ) + remainder*

เมื่อค่าที่นำไปคูณเป็นเลขที่มากที่สุดที่คูณกับ b แล้วใกล้เคียง a เช่น 9 = (4x2) +1

Method เดียวกันที่สามารถใช้ได้เหมือน % แต่ได้ค่าลบคือ

*-9 % 4 // remainder มีค่าเท่ากับ -1*

ใส่ค่า -9 และ 4 เข้าไปในสมการ -9 = (4x2) + -1 จะได้ค่าเศษ = -1

ถ้าใส่ – หรือ + ที่ค่าของ b จะมีค่าเหมือนกัน คือ a%b = a%-b

**Floating – Point Remainder Calculations**

ใน swift สามารถใช้ float กับ % (remainder) ได้

*8 % 2.5 // remainder มีค่าเท่ากับ 0.5*

ในตัวอย่างนี้ถ้า 8 หาร 2.5 จะได้เท่ากับ 3 และเหลือเศษ 0.5 ดังนั้น remainder operators return ค่า 0.5 ที่เป็น Double

**Unary Minus Operator**

เป็นสัญลักษณ์ แทนค่าลบของค่านั้น เช่น

*Let three = 3*

*Let minusThree = -three //minusThree เท่ากับ -3*

*Let plusThree = -minusThree //plusThree เท่ากับ 3*

**Unary Plus Operator**

สัญลักษณ์ (+) ใส่หน้าค่าไว้บอกว่าตัวเลขนั้นเป็นค่าบวก

**Compound Assignment Operator**

Swift สามารถใช้ (+=) เป็น operator เหมือนใน C ได้ เช่น

```
var a=1
a += 2
// a เท่ากับ 3
```

คำสั่ง a+=2 หมายถึง a = a+2

***Note***

swift สามารถใช้ identity operators (=== และ !== ) ที่เอาไว้ทดสอบว่าสอง object references ที่ชี้ไปที่ object เดียวกันหรือเปล่าได้

สำหรับ Comparison operators จะ return ค่าเป็น Bool(Boolean) Comparison operators ใช้บ่อยในคำสั่งประเภทเงื่อนไข เช่น if

```
Let name = "มานี"
  If name == "มานี" {
     Print ("ยินดีต้อนรับมานี")
} else {
     Print ("ไม่พบชื่อสมาชิก \(name) ในระบบ")
}
// พิมพ์ "ยินดีต้อนรับมานี" เพราะ name เท่ากับ "มานี"
```

### Ternary Conditional Operator

เป็น operator พิเศษมี 3 ส่วน เป็นแบบ question ? answer1 : answer2 เป็นวิธีการเขียนแบบลัดถ้า question เป็น true จะไปทำ answer1 และคืนค่าหรือ answer2 จะทำและคืนค่า Ternary Conditional Operator เป็นวิธีลัดในการเขียน

```
If question {
    Answer1
} else {
    Answer2
}
```

### Nil Coalescing Operator

(a ?? b ) nil coalescing operator แปลได้ว่าถ้า a มีค่า จะส่งค่าของ a แต่ถ้าไม่มี ( a เป็น nil ) จะส่งค่าที่เป็น default คือ b เช่น

```
a != nil ? a! = b
```

Code ข้างต้นใช้ Ternary Conditional Operator

***Note***

*ถ้า a เป็น non-nil ค่าของ b จะไม่ถูกกระทำเรียกว่า short-circuit-evaluation*

```
Let defaultColorName = "red"
Var userDefinedColorName : String ? //defaults to nil

 Var colorNameToUse = userDefinedColorName ?? default ColorName
//userDefinedColorName is nil, so ColorNameToUse is set to the default of "red"
```

```
colorNameToUse = userDefinedColorName ?? default ColorName
// userDefinedColorName is not-nil, so ColorNameToUse is set to "green"
```

**Range Operators**

Swift เพิ่มสอง range operators ที่ใช้เป็นทางลัดสำหรับบอกช่วงของค่า

**Closed Range Operator**

(a…b) บอกว่าค่าจะเริ่มตั้งแต่ a ไปถึง b รวมถึงค่าของ a และ b ด้วยและค่าของ a ต้องน้อยกว่า b

Closed range operator มีประโยชน์สำหรับใช้ใน for-in loop

```
For index in 1...5 {
   Print ("\(index) คูณ 5 เท่ากับ \(index*5)")
}
// 1 คูณ 5 เท่ากับ 5
// 2 คูณ 5 เท่ากับ 10
// 3 คูณ 5 เท่ากับ 15
// 4 คูณ 5 เท่ากับ 20
```

**Half – Open Range Operator**

(a..<b) บอกว่าเริ่มตั้งแต่ a ไปถึง b แต่ไม่รวม b และค่าของ a ต้องน้อยกว่า b

Half-open ranges เหมาะสำหรับใช้ใน arrays เพราะเริ่มต้นที่ 0 นับขึ้นไปเรื่อย ๆ แต่จะไม่รวมค่าที่เป็นขนาดของ arrays

**Logical Operator**

ตัวดำเนินการทางตรรกะ (Logical Operators) หรือ ค่าตัวแปรประเภท Boolean ที่มีค่าความจริงเป็น true และ false

ภาษา Swift สนับสนุนตัวดำเนินการทางตรรกะ ที่พบในภาษา C คือ

*Logical NOT (!a)*

*Logical AND (a && b)*

*Logical OR (a || b)*

**Logical NOT Operator**

ตัวดำเนินการทางตรรกะ NOT (!a) อ่านว่า “not a” เปลี่ยน Boolean จากค่า true เป็น false และเปลี่ยนค่าจาก false เป็น true ยกตัวอย่างเช่น

```
let passedExam = false
   if !passedExam {
                print("Sorry, You Failed!!")
               }
// พิมพ์" Sorry, You Failed!!"
```

ในที่นี้ if !passedExam สามารถอ่านได้ว่า “if not passed exam” ซึ่งมีค่าความจริงเป็น true เนื่องจาก passedExam มีค่าความจริงเป็น false

```
passedExam = false
!passedExam = true
```

จากตัวอย่างข้างต้นควรระมัดระวัง Boolean constant และชื่อตัวแปร(variable names) ควรตั้งชื่อให้สอดคล้องและเข้าใจได้ง่ายต่อการทำความเข้าใจโค้ด และควรหลีกเลี่ยงตรรกะที่จะทำให้สับสน

**Logical AND Operator**

logical AND operator (a && b) นิพจน์ a&&b จะมีค่าความจริงเป็นจริงก็ต่อเมื่อ ทั้ง a และ b มีค่าความจริงเป็น true แต่ ถ้า a หรือ b ตัวใดตัวหนึ่งเป็น false นิพจน์นั้นก็จะมีค่าความจริง เป็น false ในความเป็นจริงถ้านิพจน์ ตัวแรกมีค่าความจริงเป็น false นิพจน์ตัวที่สองจะไม่ได้รับการดำเนินการเนื่องจากว่า ไม่สามารถทำให้เกิดค่าความจริงที่เป็น true ได้ ดังตัวอย่างต่อไปนี้

```
let passedMidExam = true
let passedFinalExam = false
if passedMidExam && passedFinalExam {
   print("Congratulation, You have Passed the Exam!!")
} else {
   print("Sorry, You Failed!!")
}
// พิมพ์ " Sorry, You Failed!!"
```

### Logical OR Operator

logical OR operator (a || b) นิพจน์ a || b มีค่าความจริงเป็น true เมื่อ a หรือ b ตัวใดตัวหนึ่ง มีค่าความจริงเป็น true ดังตัวอย่างต่อไปนี้

```
let passedFinalExam = false
let passedResitExam = true
if  passedFinalExam || passedResitExam {
   print("Congratulation, You have Passed the Exam!!")
} else {
   print("Sorry, You Failed!!")
}
// พิมพ์" Congratulation, You have Passed the Exam!!"
```

### Combining Logical Operators

เราสามารถ นำตัวดำเนินการทางตรรกะ ไม่ว่าจะเป็น NOT AND หรือ OR มารวมกันเพื่อสร้างเป็นนิพจน์ได้ ดังตัวอย่างต่อไปนี้

```
if passedMidExam && passedFinalExam || hasExtraScore || passedResitExam {
   print("Congratulation, You have Passed the Exam!!")
} else {
   print("Sorry, You Failed!!")
}
// พิมพ์ " Congratulation, You have Passed the Exam!!"
```

ในตัวอย่างนี้ใช้ตัวดำเนินการทางตรรกะ (&&) AND และ (||) OR ในการสร้างนิพจน์ พิจารณานิพจน์จากตัวอย่างข้างบน ถึงแม้ว่า passedMidExam, passedFinalExam, hasExtraScore จะมีค่าความจริงเป็น false อย่างไรก็ตาม ถ้า passedResitExam มีค่าความจริงเป็น true ก็จะทำให้ ทั้งนิพจน์ มีค่าความจริงเป็น true

**Explicit Parentheses**

ในบางครั้ง วงเล็บ จะเป็นประโยชน์ เมื่อเรามี นิพจน์ที่ซับซ้อน เพราะ วงเล็บทำให้ง่ายต่อการอ่าน โดยรวมแล้วค่าความจริงไม่ได้เปลี่ยนแต่ การใส่วงเล็บทำให้มีความชัดเจนต่อผู้อ่าน :

```
i f (passedMidExam && passedFinalExam) || hasExtraScore || passedResitExam {
   print("Congratulation, You have Passed the Exam!!")
} else {
   print("Sorry, You Failed!!")
}
// พิมพ์ " Congratulation, You have Passed the Exam!!"
```

String เป็นลำดับกลุ่มของตัวอักษร เช่น “hello ,world” หรือ “albrtross” Swift string นั้นเหมือนกับ string ทั่วไป

ไวยากรณ์สำหรับการสร้าง string สามารถอ่านได้ด้วยไวยากรณ์ที่คล้ายกับภาษา C

**String Literals**

จะอยู่ภายใต้เครื่องหมาย double quote(“ ”) โดยที่ let จะใช้กับการประกาศค่าคงที่ ส่วน var ใช้กับตัวแปรทั่วไป เช่น

```
let someString = "นี่คือตัวแปร Constant"
var someString = "นี่คือตัวแปร Varaible"
```

**Initializing an Empty String**

```
var emptyString = ""
var anotherEmptyString = String()
```

ทั้ง 2 แบบนี้ empty และเท่ากัน เราสามารถใช้ Boolean isEmpty เพื่อนตรวจสอบว่า string นี้ empty หรือเปล่า *เช่น*

```
if emptyString.isEmpty {
print("Nothing to see here")}
//พิมพ์ "Nothing to see here"
```

String สามารถแก้ไขหรือเปลี่ยนเปลงค่าได้ เช่น

```
var variableString = "ครู"
variableString += " และ นักเรียน"
variableString ตอนนี้จะมีค่าเป็น "ครู และ นักเรียน"
let constantString = "ประธานนักเรียน"
constantString += " และ สมาชิกสภา"
//Error constantString จะไม่สามารถเปลี่ยนแปลงค่าได้เพราะว่าเป็นค่าคงที่
```

ได้โดยการใช้

for-loop

```
for Character in "Big!".characters {
print(Character)
}
```

จะได้ผลลัพธ์ออกมาเป็น

B

i

g

!

มีอีกฟังก์ชั่นที่เราใช่บ่อยๆนั่นคือการนับจำนวนตัวอักษรที่อยู่ข้อความนั้นๆ เราสามารถเรียกใช้ฟังก์ชั่น count มาช่วยได้ ตัวอย่างเช่น

```
let myStudents = "Mali, Nirin, Nada, Pimai"
print("ชื่อนักเรียนทั้งหมดมี \(myStudents.characters.count) ตัวอักษร")
```

**การเชื่อม Strings และ Characters**

เราสามารถทำได้โดยการใช้เครื่องหมาย (+) ตัวอย่างเช่น

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"
let stringPlusCharacter = string1 + character1      จะมีค่าเท่ากับ "hello!"
let stringPlusString = string1 + string2            จะมีค่าเท่ากับ "hello there"
let characterPlusString = character1 + string1      จะมีค่าเท่ากับ "!hello"
let characterPlusCharacter = character1 + character2  จะมีค่าเท่ากับ "!?"
```

เราสามารถกำหนดค่าใหม่ไปให้ตัวแปรได้เลย โดยใช้เครื่องหมาย(+) เปลี่ยนค่า เช่น

```
var instruction = "look over"
instruction += string2     ตอนนี้ instruction มีค่าเท่ากับ  "look over there"
var welcome = "good morning"
welcome += character1 ตอนนี้  welcome มีค่าเท่ากับ "good morning!"
```

**String  Interpolation**

เราสามารถสร้าง string ขึ้นมาใหม่จากการผสมผสาน  constants, variables, literals, and expressions และใส่ลงในเครื่องหมาย (“ “) ตัวอย่างเช่น

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
ตอนนี้ message มีค่าเป็น  "3 times 2.5 is 7.5"
```

**Comparing Strings**

**String Equality** เป็นการเปรียนเทียบ string ที่มีทั้งตัวอักษรและตำแหน่งที่เหมือนกัน เช่น

```
let firstText = "Pen Pineapple Apple Pen"
let secondText = "Pen Pineapple Apple Pen"
firstText == secondText  จะจริงเพราะ 2 ตัวแปรนี้มีค่าเท่ากัน
```

**Prefix and Suffix Equality**

เป็นการเปรียบเทียบไม่ทั้งหมด อาจจะเป็นกลุ่มคำข้างหน้าหรือหลัง ตัวอย่างเช่น

```
let romeoAndJuliet = [
"Act 1 Scene 1: Verona, A public place",
"Act 1 Scene 2: Capulet's mansion",
"Act 1 Scene 3: A room in Capulet's mansion",
"Act 1 Scene 4: A street outside Capulet's mansion",
"Act 1 Scene 5: The Great Hall in Capulet's mansion",
"Act 2 Scene 1: Outside Capulet's mansion",
"Act 2 Scene 2: Capulet's orchard",
"Act 2 Scene 3: Outside Friar Lawrence's cell",
"Act 2 Scene 4: A street in Verona",
"Act 2 Scene 5: Capulet's mansion",
"Act 2 Scene 6: Friar Lawrence's cell"]
```

เราสามารถใช้ hasPrefix method เพื่อนนับจำนวนได้

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        act1SceneCount += 1
    }
}
print("There are \(act1SceneCount) scenes in Act 1")
```

จะได้ผลลัพธ์ออกมาว่า "There are 5 scenes in Act 1"

**ถ้า** เราใช้ hasSuffix method จะทำได้แบบนี้ เช่น

```
var mansionCount = 0
var cellCount = 0
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
        mansionCount += 1
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        cellCount += 1
    }
}
print("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
// Prints "6 mansion scenes; 2 cell scenes"
```

จะได้ผลลัพธ์ว่า "6 mansion scenes; 2 cell scenes"

## Uppercase and Lowercase Strings

```
let normal = "Could you help me, please?"
let shouty = normal.uppercased()
```

จะได้ว่า *shouty* มีค่าเป็น *"COULD YOU HELP ME, PLEASE?"*

```
let whispered = normal.lowercased()
```

จะได้ว่า *whispered* มีค่าเป็น *"could you help me, please?"*

**Unicode**

คือ มาตรฐานสากลที่ใช้กำหนดรหัส เพื่อใช้แทนตัวเลข ตัวอักษร หรืออักขระต่างๆ String และ Character ในภาษา Swift สนับสนุน Unicode แบบเต็มรูปแบบ ดังนั้นเราจึงสามารถกำหนดค่า Constance หรือ Variable ด้วยภาษาใดก็ได้

- UTF-8

  แต่ละชุดจะเป็นตัวเลข ตั้งแต่ 0 จนถึง 255

- UTF-16

  จะมีค่าอยู่ระหว่าง 0 จนถึง 65535

- Unicode Scalar

  ตัวอักษรทุกตัวใน Unicode จะแสดงด้วยค่า Unicode scalar ที่เป็น 21 บิต ถ้าต้องการรู้ที่อยู่ของ Unicode scalar ก็จะต้องดูจากค่า value property ของ Type Unicode Scalar อีกที

# Collection Types

## Control Flow

Swift จะใช้หลักการทำงานของ control flow เหมือน ภาษา c ซึ่งประกอบด้วย for และ while เพื่อทำงานเป็น loop หลายๆครั้ง และ if switch เพื่อทำงาน แบบทางเลือก(condition) และ break หรือ continue เพื่อเปลี่ยนสถานะของ flow การทำงานไปที่จุดต่างๆ ใน code

Swift ยังมีการใช้ Loop ที่ชื่อว่า for-in เพื่อทำซ้ำในตัวแปร เช่น arrays, dictionaries, range, strings และ sequences

Switch ในภาษา swift จะไม่เกิด “fall through” คือเมื่อในแต่ละ case ไม่ได้ใส่ break ก็จะไม่ทำ case ข้างล่าง, case ต่างๆยังสามารถจับกับรูปแบบต่างๆได้ รวมทั้ง range, tuple และ type เฉพาะอย่าง match value ใน switch case นั้นยังสามารถเก็บตัวแปรต่างๆได้ชั่วคราวหรือส่งมาเป็น parameter ได้และยังมีเงื่อนไขเพื่อทำการ match ได้ด้วยโดยการใส่คำว่า where ในแต่ละ case

### For Loops

Swift มี loop อยู่ 2 ประเภทดังนี้

**For-in** เป็น loop สำหรับเซทของการทำซ้ำแต่ละค่าใน range, sequence, collection, progression

**For loop** จะทำซ้ำจะกว่าจะเจอเงื่อนไขที่เป็นเท็จจนจบ loop

2

**For-in**

ตัวอย่างการใช้ for-in ดังนี้

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

ในที่นี้จะได้ผลลัพธ์เป็น 11

หรือนำมาใช้กับ range เช่น

```
For index in 1...5 {
                print("\(index) ")
        } // พิมพ์ 1 2 3 4 5
```

เราสามารถ loop วน dictionary เพื่อเข้าถึงทั้งค่า key และ value ในแต่ละคู่ ดังตัวอย่างต่อไปนี้

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat":4]
        for (animalName,legCount) in numberOfLegs {
           print("\(animalName)s have \(legCount) legs")
        }
        // ants have 6 legs
        // cats have 4 legs
        // spiders have 8 legs
```

หรือเราสามารถ loop ค่าที่เป็น string ออกมาเป็น character ได้ดังนี้

```
for Characters in “Hello”.character {
        print(“\(Characters) ” ,terminator: “”)
}// พิมพ์ “H e l l o”
```

**For**

**for loop** มี syntax หรือรูปแบบการใช้ดังนี้

*For initialization ; condition ; increment {*

*statement*

*}*

**While Loops**

**While**

จะมี syntax ดังนี้

*while condition {*

*statement*

*}*

**Do – while**

จะมี syntax ดังนี้

*do {*

*statement*

*} while condition*

**ข้อสังเกตุ การทำ loop ทั้งหมดจะออกจาก loop ได้เมื่อ condition เป็น false

## Condition Statement

### If

If สามารถทำ nested if ได้เหมือนกับภาษา java โดยมี syntax ดังนี้

```
if condition  {
                    statement
                  }
else if condition {
                        statement
                        }
                 else {
                        statement
                        }
```

### Switch

Switch มี syntax ดังนี้

```
Switch some value to consider {
                        Case value 1: response to value 1
                        Case value 2, value 3 : response to value 2 or  3
                        Default: otherwise do something else
                }
```

**ข้อสังเกตุ สามารถใส่ value ที่ case เป็น character หรือ range หรือ tuple ได้

นอกจากนี้ยังสามารถใช้ value binding ได้ คือ case สามารถรับค่าตัวแปร มาได้เหมือน argument เช่น

```
let aPoint = (2,0)
                switch aPoint {
                        case (let  x,0) : print("this is \(x) value")
                        case (0,let y) : print("this is \(y) value") }
```

สิ่งสำคัญคือการใช้งาน switch จะต้องได้ผลลัพธ์ในทางใดทางหนึ่ง หากไม่แน่ใจว่าจะตรงกับเงื่อนไขใดให้ทำการกำหนด default เสมอเพื่อป้องกันการเกิด error เช่น

```
let aPoint = (1,-1)
        switch aPoint {
            case let (x,y) where x == -1  : print(" x equal  y")
            case let(x,y) where x == 1 : print("x is negative of y")
            default: print("x=1 y=-1")}
```

เราสามารถ execute code ใน สวนตางๆ ของ control flow โดยใช้ control transfer statement ดงน

- *continue*
- *break*
- *fallthrough*
- *return*

**Continue**

เมื่อเจอคำสั่ง continue จำทำรอบใหม่เมื่อจบคำสั่ง

**Break**

จะออกจาก loop หรือ control flow โดยทันที โดยไม่สนใจ statements ที่เหลือ

**Fallthrough**

ปกติแล้วภายใน switch เมื่อไม่มี break จะไม่ไหลลงไปใน case ต่อไปเหมือนกับภาษาจาวา แต่ถ้ามีการกำหนด fallthrough จะทำการเช็ค case ที่เหลือต่อไป

**Return**

เป็นการ return ค่ากลับให้ผู้ที่เรียกใช้

**Lable Statement**

เป็นตัวระบุชื่อ Control Flow เพื่อสามารถใช้เรียก หรือระบุส่วนของโค้ดนั้นๆ มี syntax ดังนี้

*Label name : while condition{*

*Statements*

*}*

ตัวอย่างเช่น

```
var myString = "Swift tutorial"
myString: for char in myString.characters {
   switch char{
   case "a","e","i","o","u":
      break
   default:
      continue myString
   }
   print("\(char) ",terminator:"")
}
```

## Functions

ฟังก์ชั่นคือกลุ่มก้อนของโค๊ดที่มีหน้าที่ทำงานเฉพาะอย่าง สามารถตั้งชื่อให้ฟังก์ชั่นเพื่อบ่งบอกถึงเป้าหมายของแต่ละฟังก์ชั่นได้ และเมื่อต้องการใช้งานก็จะเรียกผ่านชื่อฟังก์ชั่นเนี่ยละ

ทุก ๆ ฟังก์ชั่นในสวิฟท์จะมีประเภท ประกอบไปด้วยประเภทของข้อมูลที่รับเข้าและประเภทของข้อมูลที่ส่งกลับ ซึ่งจะมีหลายประเภทของข้อมูลมากมายให้เลือกใช้ได้ และที่พิเศษเลยคือสามารถส่งฟังก์ชั่นไปเป็นพารามิเตอร์ให้กับฟังก์ชั่นอื่น ๆ หรือรีเทิร์นค่าฟังก์ชั่นออกมาจากฟังก์ชั่นหนึ่ง ๆ ได้

**Defining and Calling Functions**

เมื่อคุณสร้างฟังก์ชั่นขึ้นมา คุณสามารถปรับแต่งได้หลาย ๆ อย่าง เช่น ชื่อ ประเภทของอินพุท ประเภทของเอาท์พุท ทุก ๆ ฟังก์ชั่นจะมีชื่อเป็นของตัวเอง ที่จะบ่งบอกรายละเอียดว่าฟังก์ชั่นนั้น ๆ มีหน้าที่ทำอะไร เวลาที่คุณต้องการเรียกใช้ฟังก์ชั่นก็แค่เรียกผ่านชื่อของฟังก์ชั่นแล้วตามด้วยค่าที่ต้องการส่งไปให้เป็นอินพุทของฟังก์ชั่นนั้น ๆ (หรือเรียกว่าอาร์กิวเมนท์) โดยจะต้องตรงกันกับประเภทของพารามิเตอร์ในฟังก์ชั่นด้วย โดยปกติแล้วอาร์กิวเมนท์ที่จะส่งให้ฟังก์ชั่นนั้นจะมีลำดับการเรียงเหมือนฟังก์ชั่นพารามิเตอร์

ตัวอย่างฟังก์ชั่น sayHello

```
func sayHello(personName: String) -> String {
    let greeting = "Hello, " + personName + "!"
    return greeting
}
```

จะเห็นได้ว่าชื่อฟังก์ชั่นคือ sayHello ซึ่งมีตัวแปรชื่อ personName คอยรับข้อมูลอินพุทเป็น String และรีเทิร์นค่าเอาท์พุทออกมาเป็นประเภท String เช่นกัน (ตรงส่วนของการรีเทิร์นจะอยู่หลังสัญลักษณ์ -> หรือ Arrow) โดยรายละเอียดการทำงานของฟังก์ชั่นก็จะขึ้นอยู่กับโค๊ดที่เขียน

```
print(sayHello("Harry"))
// prints "Hello, Harry!"
print(sayHello("Ron"))
// prints "Hello, Ron!"
```

ตัวอย่างการเรียกใช้งาน ซึ่งอย่างที่กล่าวไปคือเรียกใช้ผ่านชื่อฟังก์ชั่นแล้วตามด้วยค่าที่ต้องการจะส่งไปให้ในฟังก์ชั่น เช่น sayHello(“Harry”) จะเห็นได้ว่า “Harry” คือค่าที่เราส่งไป ซึ่งจะไปสัมพันธ์กับตัวแปรรับค่า personName ที่รับเป็น String เช่นกัน หลังจากนั้น ตัวฟังก์ชั่นก็จะไปทำงานของมันแล้วรีเทิร์นค่าส่งกลับมาให้ตรงที่ฟังก์ชั่นถูกเรียกใช้นั้นเอง

คุณสามารถเรียกใช้ฟังก์ชั่น sayHello ได้หลาย ๆ ครั้ง โดยที่แต่ละครั้งก็สามารถส่งค่าที่แตกต่างกันออกไปได้ เพียงแต่ให้เป็นชนิดเดียวกันก็พอ

เราสามารถลดความซับซ้อนของโค๊ดลงได้โดยการเขียนแบบย่อ ๆ แต่ให้ผลแบบเดียวกันได เช่น

```
func sayHelloAgain(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
print(sayHelloAgain("Harry"))
// prints "Hello again, Harry!"
```

**Function Parameters and Return Values**

สำหรับตัวพารามิเตอร์และค่าที่ถูกรีเทิร์นนั้น ในภาษาสวิฟท์สามารถดัดแปลงได้ง่าย เขียนได้ง่ายมาก คุณสามารถนิยามฟังก์ชั่นด้วยพารามิเตอร์ที่ไม่มีชื่อก็ได้ ไปถึงขั้นฟังก์ชั่นที่มีความซับซ้อนพร้อมกับพารามิเตอร์หลาย ๆ ตัวที่มีชื่อแตกต่างกันและหน้าที่แตกต่างกัน

**Multiple Input Parameters**

ฟังก์ชั่นสามารถมีพารามิเตอร์ได้หลายค่า ซึ่งปกติถ้าเราเขียนพารามิเตอร์เดียวก็อย่างที่ได้กล่าวไปตอนต้น ถ้าต้องการเพิ่มก็เพียงแค่ใส่ comma แล้วตามด้วยพารามิเตอร์ตัวต่อไปได้เลย เช่น

```
func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}
print(halfOpenRangeLength(start: 1,end: 100))
// พิมพ์ "99"
```

**Functions Without Parameters**

สำหรับกรณีนี้ เราจะไม่นิยามพารามิเตอร์ให้มัน ซึ่งวิธีการก็ง่าย ๆ คือไม่ต้องไปเขียนในส่วนพารามิเตอรนั้นเอง เช่

```
func HelloSwift() -> String {
    return "hello, swift"
}
print(HelloSwift())
// พิมพ์ "hello, swift"
```

จะเห็นได้ว่าหลัง HelloSwift ซึ่งเป็นชื่อฟังก์ชั่นจะตามด้วย ( ) เปล่า ๆ ไม่มีอะไรข้างในนั้นเอง

**Functions Without Return Values**

สำหรับฟังก์ชั่นประเภทนี้ คือเมื่อมันทำการทำงานของมันเสร็จสิ้น จะไม่มีการรีเทิร์นค่าใด ๆ กลับมาทั้งสิ้น ซึ่งจะสามารถเขียนได้ โดยลบโค๊ดในส่วนของการบ่งบอกฟังก์ชั่นว่าจะมีการรีเทิร์นค่าแค่นั้นเอง เช่น

```
func Goodbye(personName: String) {
    print("Goodbye, \(personName)!")
}
Goodbye(personName: "Hermione")
// พิมพ์ "Goodbye, Hermione!"
```

จะเห็นได้ว่า Goodbye(personName: String) จะไม่มีการตามต่อด้วยส่วนของการรีเทิร์นค่าข้อมูล (สัญลักษณ์ ->) ซึ่งจะสัมพันธ์กับในโค๊ดคือไม่มีคำสั่ง return เช่นกัน

*NOTE*

*สำหรับฟังก์ชั่นที่ไม่มีการรีเทิร์นค่านั้น จริง ๆ แล้วเมื่อเสร็จการทำงานมันจะรีเทิร์นค่าอยู่ ซึ่งเราจะเรียกมันว่า void เป็นประเภทของข้อมูลชนิดหนึ่งซึ่งหมายถึงความว่างเปล่า สามารถเขียนแทนได้ด้วย ()*

การรีเทิร์นค่าของข้อมูลในฟังก์ชั่นสามารถถูกละเว้นได้ ดังตัวอย่าง
ต่อไปนี้

```
func printAndCount(stringToPrint: String) -> Int {
    print(stringToPrint)
    return countElements(stringToPrint)
}
func printWithoutCounting(stringToPrint: String) {
    printAndCount(stringToPrint)
```

```
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
tWithoutCounting("hello, world")
rints "hello, world" but does not return a value
```

จะเห็นได้ว่า ในบรรทัดที่ 8 มีการเรียกใช้ฟังก์ชั่นแรกก่อน ซึ่งค่าที่ออกมาก็เป็นไปตามปกติ ได้ค่ารีเทิร์นมาเป็น 12 แต่เมื่อเรามาดูที่บรรทัด 10 ซึ่งเรียกใช้ฟังก์ชั่นที่สองก่อน (ฟังก์ชั่นที่สองไม่มีการรีเทิร์นค่า) ซึ่งในการทำงานของฟังก์ชั่นที่สองนั้นคือการไปเรียกฟังก์ชั่นแรกอีกที จะเห็นได้ว่าเอาท์พุทออกมาเป็นปกติ แต่จะไม่มีค่ารีเทิร์นออกมา เพราะว่าถึงแม้ฟังก์ชั่นแรกจะรีเทิร์นค่าออกมาก็ตาม แต่ในฟังก์ชั่นที่สองไม่มีตัวแปรใด ๆ มารับ และในตัวของฟังก์ชั่นที่สองเองก็ไม่มีคำสั่ง return อีกด้วย ดังนั้นตัวอย่างนี้แสดงให้เห็นถึงการละเว้นคำสั่ง return (แต่โดยปกติแล้ว จะไม่ผ่านคอมไพล์ด้วยซ้ำ เพราะผิดหลักการ)

**Functions with Multiple Return Values**

คุณสามารถเขียนฟังก์ชั่นแบบได้โดยแก้ตรงส่วนที่บ่งบอกถึงการรีเทิร์นค่าในส่วนหัวของฟังก์ชั่น ซึ่งข้างหลังสัญลักษณ์ -> ถ้าเราต้องการที่จะรีเทิร์นหลาย ๆ ค่า ก็ให้ทำการใส่วงเล็บแล้วตามด้วยชื่อตัวแปรที่มารับค่าก่อนออกจากฟังก์ชั่น และชนิดของตัวแปรนั้น ๆ (สามารถใช้ comma ได้เหมือนกับกรณีพารามิเตอร์) ดังเช่นตัวอย่าง

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

จะเห็นได้ว่าตรงส่วนของการรีเทิร์นค่าจะมี 2 ค่าคือ min , max ซึ่งเราส่งค่า currentMin กับ currentMax ไปให้ตามลำดับ

ในส่วนของการเรียกใช้ฟังก์ชั่นประเภทนี้ ดูได้จากตัวอย่างนี้

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

เราจะเห็นได้ว่าการเรียกใช้นั้นทำได้ตามปกติเลย คือการหาตัวแปรมารองรับหนึ่งตัว แต่วิธีการที่จะเข้าถึงนั้นแตกต่าง เราจะมองเหมือนตัวแปรที่มารับค่านั้นเป็นชื่อตาราง ซึ่งในตารางจะมี tuple เป็นค่าที่ได้รับคืนมาจากฟังก์ชั่น ในกรณีนี้จะมองได้เป็น ตาราง bounds มี tuble อยู่ 2 อัน อันแรกคือ min ซึ่งมีค่าเท่ากับ -6 อันที่สองคือ max ซึ่งมีค่าเท่ากับ 109 ส่วนวิธีการเข้าถึงนั้นก็แค่เรียกผ่านชื่อตารางตามด้วยชื่อ tuble โดยขั้นด้วย '.' นั้นเอง

และที่สำคัญเลยคือเราไม่สามารถเปลี่ยนชื่อ tuple ได้ ถ้าเราต้องการที่จะเปลี่ยนเราต้องเปลี่ยนในส่วนของฟังก์ชั่นเท่านั้น ไม่สามารถเปลี่ยนที่อื่นได้

จะเห็นได้ว่าข้างหลังส่วนรีเทิร์นข้อมูลมีการใส่ ? ตาม สิ่งนั้นหมายถึงการใส่ตัวเลือกเพิ่มเข้าไปนั้นเอง และจะเห็นได้อีกว่าถ้าเราใส่อินพุทเป็นค่าอาเรย์ที่ว่างเปล่า ตัวฟังก์ชั่นจะทำการรีเทิร์น nil ออกมาให้เราได้

*NOTE*

*การใส่ตัวเลือก 2 แบบนี้แตกต่างกัน (int,Int)? กับ (int?,int?) ซึ่งแบบแรกจะครอบคลุมทั้งหมด คือสามารถส่งค่า nil เป็นก้อนใหญ่ได้เลย แต่แบบที่สองคือต้องไปไล่ส่งค่า nil ให้ทีละตัวนั้นเอง*

**Function Parameter Names**

จากที่ได้กล่าวมาทั้งหมดในเรื่องฟังก์ชั่น เราจะเรียกชื่อพารามิเตอร์ที่ใช้ในฟังก์ชั่นว่า Parameter name ซึ่งสามารถนำมาใช้ภายใน Body ของฟังก์ชั่นและสามารถนำไปใช้เป็นชื่อพารามิเตอร์ตอนเรียกใช้งานฟังก์ชั่นได้ด้วย ดังตัวอย่าง

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)}
```

**Agrument Label**

หมายถึงชื่อพารามิเตอร์ที่ใช้ตอนเรียกใช้งานฟังก์ชั่น มีประโยชน์เพื่อให้ user รู้ว่าเวลาส่งค่าต่าง ๆ ไปให้ฟังก์ชั่น แล้วตัวแปรตัวไหนมารับ ซึ่งโดยปกติแล้ว user จะไม่มีทางรู้เลยว่า ใครจะมารับค่าต่าง ๆ ที่ส่งไป ดังนั้น การกำหนด Agrument Label จะช่วยให้ user รู้ได้ว่าตัวแปรตัวไหนมารับไป เพราะ user ต้องเป็นคนสั่งการเองผ่านชื่อตัวแปร ซึ่งการสร้างนั้นไม่ยาก เพียงแค่ใส่ชื่อ Agrument Labelไปก่อน แล้วตามด้วย Parameter name โดยขั้นกลางด้วยเว้นวรรค เช่น

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter
.}
```

*NOTE*

*ถ้าคุณต้องการใช้ Agrument Label คุณจะต้องจำชื่อตัวแปรนั้น ๆ ให้ได้ เพราะเวลาใช้งานฟังก์ชั่นจะต้องส่งค่าอินพุทต่าง ๆ โดยอ้างอิงชื่อ Agrument Label เสมอ*

ตัวอย่างการใช้งาน Agrument Label

```
func join(string s1: String, toString s2: String, withJoiner joiner: String)
    -> String {
        return s1 + joiner + s2
}
```

จะเห็นได้ว่ามี Agrument Label อยู่ 3 ตัวคือ string , toString , withJoiner ซึ่งเวลาเรียกใช้ฟังก์ชั่นก็จะแตกต่างไปอีกแบบ ดังตัวอย่างนี้

```
join(string: "hello", toString: "swift", withJoiner: ", ")
// returns "hello, swift"
```

จะเห็นได้ว่าเวลาส่งค่าไปให้ฟังก์ชั่นนั้น เราจะต้องระบุชื่อ Agrument Label ที่จะต้องการให้มารับค่าเสมอ

*NOTE*

*วิธีนี้อาจจะทำให้ใครหลาย ๆ คนงงได้ เมื่อมาอ่านโค๊ดของคุณ ถ้าคุณเขียนโค๊ดแบบปกติ(ไม่ได้ใช้พารามิเตอร์แบบภายนอก) แล้วคนที่มาอ่านโค๊ดของคุณไม่งง คุณก็ไม่จำเป็นต้องใช้วิธีการนี้หรอก*

**Shorthand External Parameter Names**

เพื่อป้องกันการสับสนหากต้องการให้ Parameter Name และ Agrument Label เป็นตัวเดียวกัน ตอนประกาศฟังก์ชั่นใช้ชื่อพารามิเตอร์แบบไหน ตอนเรียกฟังก์ชั่นให้ใช้ชื่อเดียวกันเท่านั้นเอง

```
func containsCharacter(string: String, characterToFind: Character) -> Bool {
    for character in string {
        if character == characterToFind {
            return true
        }}
    return false}
```

```
let containsAVee = containsCharacter(string: "aardvark", characterToFind: "v")
// containsAVee equals true, because "aardvark" contains a "v"
```

**Default Parameter Values**

คุณสามารถกำหนดค่าเริ่มต้นให้กับพารามิเตอร์ต่าง ๆ ของฟังก์ชั่นได้ ซึ่งถ้าคุณกำหนดแล้ว เวลาเรียกใช้ฟังก์ชั่นคุณสามารถละเว้นการใส่อินพุทให้ฟังก์ชั่นก็ได้ ซึ่งวิธีก็ง่ายมากเพียงแค่คุณต้องการให้มันมีค่าเริ่มต้นคืออะไรก็กำหนดไปที่ข้างหลังพารามิเตอร์ตัวนั้นเลย เช่น

```
func join(string s1: String, toString s2: String,
    withJoiner joiner: String = "-") -> String {
        return s1 + joiner + s2
}
```

พารามิเตอร์ joiner(withJoiner) นั้นเอง ส่วนผลลัพธ์ของการเรียกใช้ จะเป็นดังนี้

```
join(string: "hello", toString: "world", withJoiner: " ")
// returns "hello world"
```

แบบนี้คือการเรียกแบบปกติ มีการส่งค่าอินพุทให้ไปฟังก์ชั่นทุกค่า ซึ่งก็จะไม่กระทบอะไรกับฟังก์ชั่นที่มีการกำหนดค่าเริ่มต้นให้กับพารามิเตอร์ การทำงานจะเหมือนฟังก์ชั่นปกติทุกอย่าง

```
join(string: "hello", toString: "world")
// returns "hello-world"
```

ส่วนถ้าเป็นกรณีนี้คือไม่ได้ส่งค่าไปกับพารามิเตอร์ที่ชื่อ joiner(withJoiner) การทำงานในฟังก์ชั่นก็จะนำค่าเริ่มต้นที่ถูกกำหนดให้แต่แรกมาใช้แทน

**Agrument Label for Parameters with Default Values**

ในการประกาศฟังก์ชั่นเราสามารถกำหนดค่า Default Values (ค่าเริ่มต้น) ให้กับพารามิเตอร์ขณะเรียกใช้งานฟังก์ชั่น พารามิเตอร์ที่ได้กำหนดค่าเริ่มต้นไปแล้วนั้นเราไม่จำเป็นต้องกำหนดค่าให้กับพารามิเตอร์ดังกล่าวก็ได้โดยให้ข้ามพารามิเตอร์นั้นไปได้ หรือหากต้องการเปลี่ยนแปลงก็สามารถทำการกำหนดค่าเริ่มต้นให้ตามที่ต้องการ ดังนี้

```
func join(_ s1: String, _ s2: String, joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

```
let result = join("hello", "world", joiner: "-")
print(result) // returns "hello-world"
```

จะเห็นว่าตอนส่งค่าอินพุทไปนั้น ได้ใช้ตัวแปรที่ชื่อว่า joiner ซึ่งมีชื่อเดียวกันกับ Parameter name (แต่จริง ๆ แล้วเหมือนเราเรียกใช้ Agrument Label)

*NOTE*

*คุณสามารถเลือกได้ว่าจะให้ทางสวิฟท์สร้างตัวแปรพารามิเตอร์แบบภายนอกให้เองหรือไม่ต้องการ ถ้าไม่ต้องการเพียงแค่คุณใส่ _ แทนชื่อตัวแปรพารามิเตอร์แบบภายนอกเป็นอันจบ ซึ่งถ้าทำแบบหลัง เวลาเรียกใช้ฟังก์ชั่นก็จะทำแบบปกติ แต่ถ้าเรียกใช้ฟังก์ชั่นแบบระบุชื่อตัวแปรพารามิเตอร์แบบภายนอกไป จะไม่สามารถทำงานได้*

**Variadic Parameters**

เราจะเรียกพารามิเตอร์ตัวนั้นว่า variadic ซึ่งจะทำการรับค่าได้หลาย ๆ ค่าภายในตัวเดียว ดังนั้นค่าหลาย ๆ ค่าที่รับมานั้นจะอยู่ในรูปแบบของ Array นั้นเอง ตัวอย่าง

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
```

พารามเตอรทชอวา numbers โดยในฟงกชนตวแปร numbers จะกลายเปน Array ไปโดยปรยาย ดงนนในการทำงานของฟังก์ชั่นจะมีลูปที่เกี่ยวกับ Array numbers อยู่ด้วย

*NOTE*

*ในแต่ละฟังก์ชั่นจะมี variadic ได้มากสุดเพียงแค่ 1 ตัว และถ้าเกิดฟังก์ชั่นนั้นมีหลายพารามิเตอร์ variadic จะอยู่ในตัวสุดท้ายของรายชื่อพารามิเตอร์*

**Constant and Variable Parameters**

โดยปกติแล้วเมื่อเราส่งค่าอินพุทให้กับฟังก์ชั่น พารามิเตอร์ที่มารับนั้นจะมองอินพุทที่ส่งมาเป็นค่าคงที่ทันที นั้นหมายความว่าเราจะไม่สามารถเปลี่ยนแปลงค่าในพารามิเตอร์ได้ ดังนั้นทางสวิฟท์จึงมี

```
func myString(string stringInput: String){
    var stringInput = stringInput
    stringInput += "Swift"
    print(stringInput)
}
myString(string:"Hello ")
```

สามารถเปลี่ยนแปลงค่าในตัวของมันเองได้ ถ้างสังเกตการทำงานในฟังก์ชั่นจะเห็นว่ามีการเปลี่ยนแปลงค่า string ซึ่งสามารถทำได้

**In-Out Parameters**

โดยปกติแล้วเวลาเราแก้ไขค่าต่าง ๆ ในฟังก์ชั่น ภายนอกฟังก์ชั่นจะไม่มีการเปลี่ยนแปลง เช่น ถ้าเราส่งค่าของตัวแปรใดตัวแปรหนึ่งมาเป็นอินพุทของฟังก์ชั่น เวลาเราแก้ไขค่านั้นในฟังก์ชั่น ค่าดั้งเดิมที่อยู่กับตัวแปรนั้นจะไม่เปลี่ยนแปลง สิ่งที่เปลี่ยนแปลงจะอยู่ภายในการใช้งานของฟังก์ชั่นเท่านั้น แต่ถ้าเราต้องการให้เวลาแก้ค่าในฟังก์ชั่น แล้วข้างนอกเปลี่ยนไปด้วย เราจำเป็นจะต้องใส่คำว่า inout เข้าไปข้างหน้า Type ของพารามิเตอร์ ดังตัวอย่าง

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

*NOTE*

*inout ไม่สามารถกำหนดค่าเริ่มต้นของพารามิเตอร์ได้ และไม่สามารถกำหนดข้างหน้าให้เป็นแบบ let หรือ var ได้*

จากตัวอย่าง จะเห็นว่าพารามิเตอร์ a เป็นแบบ inout ซึ่งเวลาเราจะส่งค่าให้กับ a นั้นเราจะต้องส่งเป็นแบบ & นำหน้า(กล่าวคือการใส่ & ข้างหน้าคือการส่งแบบ address ของตัวแปรตัวนั้น) ดังตัวอย่าง

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt,&anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// prints "someInt is now 107, and anotherInt is now 3"
```

บรรทัดที่ 5 จะรู้ได้ว่ามันเปลี่ยนแปลงค่าไปจริง ๆ ทั้ง ๆ ที่ไม่มีการแก้ไขค่าที่ main เลย

> *NOTE*
>
> *อย่าสับสนระหว่าง inout กับ return ทั้งสองอย่างนี้แตกต่างกันโดยสิ้นเชิง inout คือการแก้ไขค่าในตัวแปรนั้นจริง ๆ ส่งผลต่อค่าของตัวแปรนี้ในภายนอกฟังก์ชั่น แต่ return คือการส่งคืนค่ากลับมาเท่านั้น*

**Function Types**

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

จากตัวอย่างเราจะเห็นได้ว่าทั้งสองฟังก์ชั่น มี 2 พารามิเตอร์แบบ int และ return แบบ int เช่นเดียวกัน ซึ่งการเรียกทั้งสองตัวนี้(พารามิเตอร์และรีเทิร์น) เราจะเรียกมันว่า function type ซึ่งสามารถเขียนได้เป็น (Int, Int) -> Int ซึ่งสามารถอ่านได้ว่า ฟังก์ชั่นนี้มีพารามิเตอร์ 2 ตัวเป็นแบบ Int ทั้งคู่และรีเทิร์นค่าเป็น Int เช่นกัน

```
func printHelloWorld() {
    print("hello, world")
}
```

ส่วนตัวอย่างนี้จะมี function type คือ () -> () หรือจะอ่านว่า ฟังก์ชั่นนี้ไม่มีพารามิเตอร์และมีค่ารีเทิร์นเป็น void

**Using Function Types**

เราสามารถใช้งาน function types เป็นเหมือนค่าคงที่หรือตัวแปรได้ เช่น

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

ดังรูปด้านบน จะเห็นว่าตัวแปรชื่อ mathFunction มีการกำหนด function type เป็นแบบ 2 พารามิเตอร์ Int และ return Int ซึ่งสุดท้ายแล้วมีการกำหนดไปให้อ้างอิงถึงฟังก์ชั่น addTwoInts ด้วย ดังนั้นเวลาเรียกใช้ฟังก์ชั่น addTwoInts เราจะสามารถเรียกใช้ผ่าน mathFunction ก็ได้เช่นกัน ดังนี้

```
print("Result: \(mathFunction(2, 3))")
// prints "Result: 5"
```

ซึ่งเราสามารถเปลี่ยนแปลงฟังก์ชั่นที่ mathFunction อ้างอิงได้เสมอ โดยการกำหนดค่าให้มันใหม่ได้เลย ดังนี้

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// prints "Result: 6"
```

เวลาเรียกใช้ ก็จะเปลี่ยนไปที่ฟังก์ชั่น multiplyTwoInts ไปโดยปริยาย

แต่จริง ๆ แล้วเราไม่จำเป็นต้องกำหนด function type เองก็ได้ ปล่อยให้เป็นหน้าที่ของสวิฟท์ในการกำหนดได้ เช่น

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

เราแค่ let ตัวแปรใดตัวแปรหนึ่งให้ไปอ้างอิงถึงฟังก์ชั่นที่เราต้องการ ทางสวิฟท์ก็จะจัดการเรื่อง function type ให้เราเองโดยอัตโนมัติ ซึ่งในกรณีนี้ anotherMathFunction จะมี function type คือ (int, int) -> int ตามฟังก์ชั่น addTwoInts

**Function Types as Parameter Types**

เราสามารถใช้ function type ไปเป็นพารามิเตอร์ในฟังก์ชั่นใด ๆ ได้ ซึ่งนั้นหมายถึงว่า พารามิเตอร์ที่ทำตัวเป็น function type จะคอยรับค่าฟังก์ชั่นต่าง ๆ ที่ส่งมานั้นเอง เช่น

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "Result: 8"
```

จะเห็นว่าบรรทัดที่ 4 เราส่ง addToInts ไปให้กับพารามิเตอร์แบบ function type ซึ่งพารามิเตอร์จะไปมองหาว่ามีฟังก์ชั่น addToInts อยู่หรือไม่ ถ้ามีก็จะทำการอ้างอิงไปถึงฟังก์ชั่นนั้น ๆ ให้ผ่านชื่อตัวแปรพารามิเตอร์ ส่วนการทำงานก็เหมือนกับเรื่องด้านบนซึ่งได้อธิบายไว้แล้ว

**Function Types as Return Types**

โดยหลักการแล้วจะเหมือนกับใช้ในแบบ parameter type เพียงแต่เราไปใส่ function type ในส่วนของการรีเทิร์นค่า สมมติว่าเรามี ฟังก์ชั่นดังรูป

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

ซึ่งเป็นฟังก์ชั่นปกติ และเรามีฟังก์ชั่นอีกอันคือ

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
```

จะเห็นว่าฟังก์ชั่นนี้มีการใช้ function type ในส่วนของการรีเทิร์นค่า เพราะฉะนั้นการริเทิร์นค่าหลังจากจบการทำงานในฟังก์ชั่นก็จะรีเทิร์นฟังก์ชั่นใด ๆ ออกมา ซึ่งนั้นจะหมายถึงว่าตัวที่รอรับค่านั้นจะสามารถอ้างอิงถึงฟังก์ชั่นที่ถูกริเทิร์นออกมาได้ เช่น

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backwards: currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

จะเห็นได้ว่าบรรทัดที่สองเราส่งค่า true ไปให้กับฟังก์ชั่น chooseStepFuntion ซึ่งภายในฟังก์ชั่นนั้นจะก็รีเทิร์นค่า stepBackward มาให้ แต่จะเป็นแบบ function ซึ่งนั้นหมายความว่า ตัวแปร moveNearerToZero จะไปอ้างอิงถึงฟังก์ชั่น stepBackward นั้นเอง ส่วนตัวอย่างการเรียกใช้ผ่านตัวแปรนั้น เป็นดังนี้

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```

**Nested Functions**

จากทั้งหมดที่กล่าวไป ฟังก์ชั่นทั้งหลายนั้นเป็นแบบ global functions ซึ่งสำหรับ nested นั้นจะเป็นการนิยามฟังก์ชั่นภายใต้ฟังก์ชั่นอีกที ซึ่งจะเรียกฟังก์ชั่นที่อยู่ภายใต้ฟังก์ชั่นนั้นเป็น nested function ซึ่งคุณสมบัติของ nested function นั้นก็คือ จะไม่สามารถเรียกใช้ได้จากภายนอก จะเรียกใช้ได้แค่ภายในฟังก์ชั่นที่นิยามมันขึ้นเท่านั้น ตัวอย่างดังรูป

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward }
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backwards: currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
print("\(currentValue)... ")
currentValue = moveNearerToZero(currentValue)
} print("zero!")
```

## Closures

**Closures** มีบล็อกของฟังก์ชันของตัวมันเอง Closures ใน Swift มีความคล้ายกับ บล็อกใน ภาษา C และ Objective-C และคล้ายกับในภาษาโปรแกรมอื่น ๆ

Closures สามารถจับ(capture) และเก็บ references ของค่าคงที่และตัวแปรจากการกำหนด ภาษา Swift ทำหน้าที่จัดการหน่วยความจำ

Global และ nested functions ตามที่แนะนำในฟังก์ชัน เป็นกรณีพิเศษของ Closures หนึ่งในสามรูปแบบ:

- *Global functions คือ closures ที่ไม่เก็บค่าใด ๆ*
- *Nested functions คือ closures สามารถเก็บค่าใด ๆ ได้ จาก ฟังก์ชันของตัวเอง*
- *Closure สามารถจับค่าโดยรอบของ Closure เอง*

นิพจน์ของภาษา Swift มีรูปแบบที่ชัดเจนด้วยการเพิ่มประสิทธิภาพ โดยปราศจากความไม่เป็นระเบียบของไวยากรณ์ในสถานการณ์ทั่วไป การเพิ่มประสิทธิภาพเหล่านี้รวมถึง :

- *การส่งกลับค่าพารามิเตอร์*
- *การส่งกลับค่า implicit จากนิพจน์ single-expression*
- *ชื่อ อากิวเมนต์*
- *ไวยากรณ์*

### Closure Expressions

Nested functions เป็นวิธีที่สะดวกของการตั้งชื่อและกำหนดบล็อกของตัวมันเอง เป็นส่วนหนึ่งของฟังก์ชั่นขนาดใหญ่ อย่างไรก็ตามบางครั้งก็เป็นประโยชน์ในการเขียนรูปแบบโครงสร้างแบบสั้นโดยไม่ต้องเขียนแบบเต็ม นี่คือความจริงโดยเฉพาะอย่างยิ่งเมื่อคุณ ทำงานกับฟังก์ชันที่ใช้ฟังก์ชันอื่น ๆ

Closure expressions เป็นวิธีการเขียนไวยากรณ์แบบย่อ จัดเตรียมรูปแบบของไวยากรณ์ไว้หลายรูปแบบในการเขียนโดยไม่สูญเสียความชัดเจนหรือเจตนา โดยในตัวอย่างข้างล่างนี้ แสดงถึงการทำงานแบบเดียวกัน

**The Sorted Function**

ไลบรารีมาตรฐานของ Swift จัดเตรียมฟังก์ชันที่เรียกว่า sort การเรียงลำดับ การจัดเรียงลำดับค่าที่อยู่ในอาร์เรย์ เมื่อการจัดเรียงลำดับเสร็จสิ้น sort function จะส่งค่า ใหม่ ของอาร์เรย์ที่มีชนิด และขนาดเท่ากับอาร์เรย์ตั้งต้น และมีการเรียงลำดับที่ถูกต้อง

ตัวอย่างต่อไปนี้ใช้ฟังก์ชันการเรียงลำดับ sort function จะเรียงลำดับ ค่าอาร์เรย์ของสตริง ค่าตามลำดับตัวอักษรแบบย้อนกลับ นี่คือ อาร์เรย์ที่เริ่มต้นเรียงลำดับ :

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

ตัวอย่างการ จัดเรียงลำดับค่าสตริงของอาร์เรย์ โดยมีฟังก์ชัน (String, String) -> Bool

*func backwards(s1: String, s2: String) -> Bool {*

```
func backwards(_ s1: String, _ s2: String) -> Bool {
return s1 > s2
}
var reversed = names.sorted(by: backwards)
// reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

***Closure Expression Syntax***

```
{ ( parameters ) -> return type in
statements
}
```

```
reversed = names.sorted(by: { (s1: String, s2: String) -> Bool in
   return s1 > s2
})
```

return จะเขียนอยู่ใน ปีกกา { }

จากตัวอย่างข้างต้นสามารถเขียนให้อยู่ในบรรทัดเดียวได้

```
reversed = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

**Inferring Type From Context**

สวิฟท์สามารถสรุปประเภทของพารามิเตอร์และชนิดของค่าที่มันส่งกลับมาจากชนิดของการจัดเรียงที่สองฟังก์ชั่นของพารามิเตอร์ พารามิเตอร์นี้ถูกคาดหวังว่าการทำงานของชนิด (String, String) -> Bool ซึ่งหมายความว่าสตริงสตริงและประเภท Bool ไม่จำเป็นต้องเขียนเป็นส่วนหนึ่งของความหมายของการแสดงออกปิด เพราะทุกประเภทสามารถสรุปลูกศรกลับมา (->)และยังสามารถละเครื่องหมายวงเล็บล้อมรอบชื่อพารามิเตอร์:

```
reversed = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

**Implicit Returns from Single-Expression Closures**

Closures แบบ single-expression (บรรทัดเดียว) สามารถส่งค่าผลลัพธ์กลับได้ด้วยตัวเอง โดยที่ไม่ต้องใช้คำว่า return ใน expression อย่างเช่น

```
reversed = names.sorted(by: { s1, s2 in s1 > s2 } )
```

argument ที่สองของ ฟังก์ชั่น sorted มั่นใจได้ว่าต้องส่งค่ากลับเป็น Bool เพราะ (s1 > s2) ส่งค่ากลับเป็น Bool และสามารถข้ามคำว่า return ไปได้

**Shorthand Argument Names**

Swift จัดการ การเขียน argument อย่างสั้น ให้เองอัตโนมัติ สำหรับข้างใน closure ที่เอาไว้ใช้ อ้างไปยัง ค่าต่าง ๆ ของ argument ใน closure เช่น $0, $1, $2, ถ้าใช้ shorthand argument names เหล่านี้ใน closure ก็จะเป็นการอ้างถึงมันทันที โดยที่ไม่ต้องใช้ keyword “in”

```
reversed = names.sorted(by: { $0 > $1 } )
```

$0 และ $1 อ้างถึง String argument ที่ 1 และ 2

**Operator Functions**

แต่เราสามารถเขียน code ข้างต้นได้ง่ายกว่านี้อีกโดยใช้ Swift's string type เช่น (>) greater-than operator เป็นฟังก์ชั่น ที่เหมาะสำหรับ sort และส่งค่ากลับเป็น Bool คุณสามารถส่งค่า greater-than operator และ swift จะรู้เองว่าต้องการใช้ string-specific implementation เช่น

```
reversed = names.sorted(by: >)
```

**Trailing Closures**

ถ้าคุณต้องการส่งค่าจาก นิพจน์ประเภท closure ไปให้ function ใด เป็นเหมือนกับ final argument ของ functionนั้น และ นิพจน์ประเภท closure นั้นมีความยาว สามารถใช้ trailing closure ได้ โดย trailing closure เป็น closure ที่เขียนข้างนอกวงเล็บ และยังตามหลังวงเล็บ ของ function ที่เรียกใช้มัน

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
// function body goes here
}
// here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure({
// closure's body goes here
})
//here's how you call this function with a trailing closure instead:
someFunctionThatTakesAClosure() {
// trailing closure's body goes here
```

*NOTE*

*ถ้า closure เป็นแบบต้องการเพียง argument เดียวเราไม่จำเป็นต้องใช้วงเล็บ() ตามหลังชื่อ Function*

เมื่อต้องการเรียกใช้ string-sort closure จากหัวข้อ Closure Expression Syntax สามารถเขียนนอกวงเล็บของฟังก์ชั่น sort เช่น

*reversed = sort(names) { $0 > $1 }*

Trailing closure มีประโยชน์ เมื่อ closure ยาวมากและไม่สามารถเขียนให้จบได้ในบรรทัดเดียว ดังตัวอย่าง Array ของ Swift มี method map ที่เอาค่าขอ array มาทีละตัว และส่งค่ากลับเป็นตัวที่ถูก map ออกมา โดยค่าที่ส่งจะเรียกออกมาตามการถูกส่งเข้าไป

สำหรับวิธีการใช้ trailing closure ให้กับ array ที่ใช้ method map เปลี่ยน Int เป็น String เช่นarray

```
[16, 58, 510] => ["OneSix", "FiveEight", "FiveOneZero"]
```

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

trailing closure เป็นดังนี้

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
//strings value คือ ["OneSix", "FiveEight", "FiveOneZero"]
```

map method จะเรียก closure expression ทุกๆค่าของ array โดยที่เราไม่จำเป็นต้องกำหนด อย่างเช่นในตัวอย่าง parameter number ของ closure เป็นเหมือนตัวแปรใน closure ดังนั้นค่าของมันจึงสามารถเปลี่ยนแปลงได้

Closure expression สร้าง string output ทุกครั้งที่ถูกเรียก มันคำนวณตัวเลขสุดท้ายของ Number โดยใช้ number % 10 แล้วเอาค่านี้ไปหาที่ digitNames dictionary String รับค่าจาก digitNames dictionary ถูกใส่ไปที่หน้า output ทำให้ string version เป็นคำแบบย้อนกลับ ตัวแปร number ถูกหาร 10 และถูกลดค่าลงมา เช่น 16 เป็น 1 58 เป็น 5 กระบวนการจะเป็นไปเรื่อยๆ จน number /= 10 มีค่าเท่ากับ 0 ที่จุดนั้น output string จะถูก Return และใส่เพิ่มไปใน output array โดย map method

**Capturing Values**

Closure สามารถเข้าถึง ค่าคงที่หรือตัวแปรจากสภาพแวดล้อมได้หากประกาศแล้ว ซึ่งสามารถอ้างถึง และแก้ไขค่าของตัวแปรภายใน closure ได้

ตัวอย่างที่ง่ายที่สุดของ closure ในภาษา Swift คือ การซ้อนกันของฟังค์ชั่น (nested function) หรือการที่เขียนส่วนของฟังค์ชั่นไว้ในฟังค์ชั่นอื่นๆ ภายในฟังค์ชั่นที่ซ้อนอยู่ สามารถเข้าถึงตัวแปร และแก้ไขค่าของตัวแปรที่อยู่ภายนอกตัวเองแต่อยู่ในฟังค์ชั่นหลักได้

ตัวอย่างในที่นี้มีฟังค์ชั่นชื่อ makeIncrementor ที่มีฟังค์ชั่นซ้อนอยู่ภายในชื่อ incrementor ภายในฟังค์ชั่นที่ซ้อนอยู่มีการเข้าถึงตัวแปร runningTotal และ amount จากสภาพแวดล้อมโดยรอบ และฟังค์ชั่น incrementor ถูกในส่งค่า (return) โดย makeIncrementor โดยนำค่าที่ได้จาก runningTotal มาบวกกับ amount ในแต่ละครั้ง

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

จะส่งออกมาเป็นค่าทั่วไป ฟังค์ชั่นที่ส่งออกมาจะไม่มี parameters และ ค่าที่ได้จากฟังค์ชั่นคือ Int

ฟังค์ชั่น makeIncrementor ประกาศตัวแปร int ที่ชื่อ runningTotal เพื่อเก็บค่าในรอบปัจจุบันหลังจากการบวก และจะถูก return ออกมาโดยให้ค่าเริ่มต้น = 0

ฟังค์ชั่น makeIncrementor มีparameter 1 ตัวเป็นชนิด Int โดยมีชื่อภายนอกคือ forIncrement และชื่อภายในฟังค์ชั่นคือ amount โดยส่งมาเพื่อบอกว่าค่าที่ runningTotal ควรจะเพิ่มเท่าไหรในแต่ละรอบของการเรียกฟังค์ชั่น

ฟังค์ชั่น makeIncrementor มีการประกาศฟังค์ชั่นซ้อนชื่อ incrementor โดยทำการเพิ่มค่า runningTotal ด้วยค่า amount ที่ได้รับเข้ามาและส่งออกเป็นผลลัพท์ของ function

```
func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
}
```

Incrementor ไม่มี parameters เลยและอ้างถึง runningTotal และ amount จากรายละเอียดการทำงานของฟังค์ชั่นโดยรอบและนำมาใช้ในฟังค์ชั่นของตัวเอง

แต่ค่า amount และ runningTotal จะไม่ได้ถูกแก้ไขจริงๆแต่จะถูกสำเนาจากค่าที่เก็บอยู่ในตัวแปร amount และถูกสร้างขึ้นใหม่ภายในฟังค์ชั่น incrementor

อย่างไรก็ตาม ค่าของ runningTotal จะถูกแก้ไขทุกครั้งที่ถูกเรียก incrementor จะจับค่าปัจจุบันของตัวแปร runningTotal และทำให้ค่าของ runningTotal ไม่หายไปเมื่อ makeIncrementor จบลงและยังยืนยันว่าค่า runningTotal จะยังสารถใช้งานได้เมื่อฟังค์ชั่น incrementor ถูกเรียกครั้งถัดไป

*NOTE*

*Swift จะทำการตรวจสอบว่าค่าใดควรเก็บไว้ โดยผู้ใช้ไม่ต้องบอกว่าจ้องการจะใช้ amount และ runningTotal ในฟังค์ชั่นที่ซ้อนกันอยู่ Swiftจ จะจัดการเรื่องหน่วยความจำ และตรวจสอบหากตัวแปรไม่ต้องการการเข้าอีกจากฟังค์ชั่น incrementor*

จากตัวอย่าง makeIncrementor

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

กลุ่มของตัวอย่างนี้เรียกว่า incrementByTen โดยอ้างถึงฟังค์ชั่น incrementor ที่เพิ่มค่า runningTotal ด้วย 10 การเรียกแต่ละครั้งจะได้ผลลัพธ์ดังต่อไปนี้

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

หาทำการสร้างฟังค์ชั่น incrementor ขึ้นมาอีกจะได้รับ runningTotal ที่แยกจากกัน ในตัวอย่างด้านล่าง incrementBySeven จะได้ค่าตัวแปร runningTotal ขึ้นมาใหม่ และไม่ได้ขึ้นอยู่กับการนับของตัว incrementByTen

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()
// returns a value of 7
incrementByTen()
// returns a value of 40
```

## Closures เป็นตัวแปร Reference

ในตัวอย่างก่อน incrementBySeven และ incrementByTen คือค่าคงที่โดยแต่ละค่าอ้างถึงตัวแปร runningTotal ของตัวเอง

เมื่อไหร่ก็ตามที่เราประกาศฟังค์ชั่นเป็นตัวแปร หรือค่าคงที่ เราจะทำการกำหนดค่าอ้างอิงที่ไปถึงฟังค์ชั่นหรือ closure นั้นๆได้

หรือกล่าวได้ว่าหากเราประกาศตัวแปรหรือค่าคงที่สองตัว หากทั้งสองตัวเก็บค่าที่อ้างไปถึง closure ตัวเดียวกันดังตัวอย่าง

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50
```

## Enumerations

enumerations คือ กลุ่มของค่าๆนึงเอาไว้เพื่อใช้ใน code ในกรณีต่างให้ง่ายขึ้น โดยที่เราไม่ต้องกำหนด type ของ value ให้ enum โดย value ที่กำหนดใน enum จะเรียกว่า raw value โดย value นั้นจะสามารถเป็นได้ทั้ง string , character, integer หรือ floating-point

enumerations member สามารถระบุ value เป็น type อะไรก็ได้ใน กลุ่มเดียวกัน

enumerations ใน ภาษา swift เราจะเรียกว่า first-class types ซึ่งจะสามารถปรับ หรือ สนับสนุนกับ คุณสมบัติต่างๆได้เช่น Class ซึ่งก็คือสามารถ คำนวณเพื่อแสดงข้อมูลของ enumeration ปัจจุบันได้ ,method เพื่อเรียก value เป็นต้น

### Enumerations Syntax

เราสามารถ define enumerations ได้โดย keyword ว่า " enum " ดังตัวอย่างนี้

```
enum Shapes {
   case Square
   case Circle
   case Triangle
   case Oval
}
```

value ที่ถูก define ใน enum จะเรียกว่า member value และ " case " keyword เพื่อบอกว่าค่าที่ใส่ของ value ที่ define

เราสามารถประกาศ value ของ enum ภายในบรรทัดเดียวได้ดังนี้

```
enum Weeks
   {case Monday, Tuseday, Wednesday, Thursday, Friday, Saturday, Sunday}
```

ตัวอย่างต่อไปนี้

```
var TrueDimension = Shapes.Square
```

เมื่อเราทำการ Initialize enum ไปแล้ว ตัวแปร ชื่อ TrueDimension ก็จะสื่อถึง enum Shapes เมื่อจะหนดค่าอื่นๆสามารถทำได้สั้นลง ดังนี้

```
TrueDimension = .Circle
```

### Matching Enumeration Values with a Switch Statement

ตัวอย่าง

```
TrueDimension = .Oval
switch TrueDimension {
case .Square: print("สี่เหลี่ยม")
case .Triangle:print("สามเหลี่ยม")
default: print("ไม่ระบุ")
}
// prints "ไม่ระบุ"
```

ข้อสังเกตุ เราควรใส่ default case เพื่อป้องกัน switch ไม่เข้า case ไหนเลย

### Associated Values

enumerations สามารถเก็บ associated value ได้ คือสามารถเก็บ type value ไหนก็ได้ใน enumeretions เรารู้จักรูปแบบนี้ในภาษโปรแกรมมิ่งอื่นๆ เช่น discriminated union, tageed unions, variants

ตัวอย่างเช่น barcode ส่วนใหญ่แบ่งได้ เป็น 2 ประเภทคือ UPC-A barcodes ซึ่งจะระบุด้วยจำนวน integers 4 ตัว และ QR code barcode ซึ่งจะถูก encode ด้วย string ความยาวสูงสุด 2,953 ตัว ดังนั้นจะได้ enum ดังนี้

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

ตัวอย่างการ สร้างตัวแปร barcode จะได้ดังนี้

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

สินค้าประเภทเดียวกันแต่ใช้อีก barcode จะได้ดังนี้

```
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

ตัวอย่างไปใช้ กับ switch (“let” prefix เพื่อรับค่ามาแต่ละ associated value)

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
   print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
   print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

**Raw Values**

ตัวอย่างที่ผ่านมาเรื่อง barcode เป็นตัวอย่างหนึ่งของ Associated Values แสดงให้เห็นว่า enumerations สามารถเก็บ value ที่ type ต่างกันได้ , ค่าเริ่มต้น หรือ default value ของ enumeration หรือเรียกว่า raw values จะถูกกำหนดไว้เป็น type ที่เหมือนกันทั้งหมด

ดังตัวอย่างต่อไปนี้ enum สำหรับเก็บ ASCII

```
enum ASCIIControlCharacter: Character {
   case Tab = “\t”
   case LineFeed = “\n”
   case CarriageReturn = “\r”
}
```

ตัวอย่างข้างต้นคือ enum ชื่อ ASCIIControlCharacter ที่เป็น type Character จำไว้ว่า raw value ไม่เหมือนกับ associated value, สมาชิก ใน rawa value จะมี type เหมือนกันเสมอ

Raw Value สามารถเป็นได้ทั้ง string,characters,interger หรือ floating-point

Raw value มีคุณสมบัติอย่างหนึ่งเรียกว่า auto incrementation ดังตัวอย่างต่อไปนี้

```
enum Weeks: Int {
   case Monday = 1, Tuseday, Wednesday, Thursday, Friday, Saturday, Sunday
}
```

จะเห็นได้ว่ามี value บางตัวไม่ได้ถูกกำหนดค่า ในที่นี้จะได้ Tuseday = 2 , Wednesday = 3 , Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7 จากการ auto incrementation

การเข้าถึง raw value ทำได้ดังนี้

```
let FridayNum = Weeks.Friday.rawValue
// FridayNum = 5
```

**Initializing from a Raw Value**

ถ้าเรา define enumerations เป็น raw value type, enumerations จะกำหนดค่าเริ่มต้น(initializer) คือรับ parameter ชื่อว่า “rawValue” และ ค่าที่ return จะเป็น value ของ enum หรือ nil เราสามารถ initializer enum ที่เป็น raw value ได้เองดังนี้

```
let FreeDay = Weeks(rawValue: 7)
// FreeDay จะเป็น type Weeks มีค่าเท่ากับ Weeks.Sunday
```

ถ้าเราพยายามหา Weeks ด้วยตำแหน่งที่เกิน 7 เกินค่าที่อยู่ใน enum จะได้ ค่า return กลับมา เป็น nil

## Classes and Structures

Swift ไม่ต้องการ การสร้าง interface แยกออกมาก ทำให้ไม่ต้อง implementation class โดยสามารถสร้าง class ในไฟล์เดียวกันได้

### Comparing Classes and Structures

- ประกาศตัวแปร
- สร้าง mothod
- กำหนดค่าให้กับตัวแปร
- กำหนดค่าเริ่มต้นการทำงาน

### Definition Syntax

มีรูปแบบการเขียน class และ structure ดังนี้

```
Struct NameStruct {
   Var name = "Test"
}
Class NameClass {
   Var nameStruct = NameStruct()
   Var sname = "swift"
}
```

จากตัวอย่างเป็นการสร้าง structure ชื่อว่า NameStruct โดยมีการเก็บค่า name และใน class NameClass มีการสร้าง Structure ด้วยค่าเริ่มต้นด้วย

### Class and Structure Instances

การสร้างค่าเริ่มต้น ทำได้โดย

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

**Accessing Properties**

การเข้าถึงค่าตัวแปร instances ทำได้โดยการระบุชื่อ class และจุดโดยไม่ต้องเว้นวรรคแล้วต่อด้วยชื่อของ instances นั้นๆ เช่น

```
someResolution.width
```

และยังสามารถเรียกตัวแปร instances ผ่าน structure ย่อยได้ด้วย เช่น

```
NameClass.nameStruct.name
```

หรือจะกำหนดค่าให้กับ instances นั้นก็ได้ เช่น

```
NameClass.nameStruct.name = “Hello”
```

**Memberwise Initializers for Structure Types**

การกำหนดค่าหลายๆ instances ให้กับ Structure Type ทำได้โดย

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

ซึ่งการทำแบบนี้ ไม่สามารถนำไปใช้กับ class ได้

## การส่งต่อค่า และรูปแบบ

เช่น

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

เมื่อทำแบบนี้ cinema และ hd จะมี width และ height เหมือนกัน โดยทั้ง 2 อิสระต่อกัน เช่น

```
cinema.width = 2048
```

จะทำให้มี width ของ cinema เปลี่ยนเป็น 2048 แต่ของ hd ยังคงเป็น 1920 อยู่

**Classes Are Reference Types**

สามารถส่งผ่านค่าที่อ้างถึงได้โดย การเปลี่ยนแปลงค่าจะส่งผลถึงกัน เช่น

```
Let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

จากการทำดังกล่าว จะทำให้ framerate ของทั้ง tenEighty และ alsoTenEight มีค่าเท่ากับ 30 ทั้งคู่

**Identity Operators**

การเช็คค่าว่า reference เดียวกันหรือไม่ใช่ ให้ใช้ (same class)

*Identical ใช้ (===)*

*Not identical ใช้ (!==)*

## การเลือกใช้ Class และ Structures

Structures เหมาะกับพวก geometric และรูปแบบ type เดียวกัน และข้อสำคัญ Structures ไม่สามารถสืบทอดได้

**Dictionaries Copy**

การ Copy Dictionaries จะเป็นการสร้างขึ้นมาใหม่ ทำให้เมื่อมีการเปลี่ยนแปลงค่าที่ copy จะไม่ส่งผลถึงค่าเดิม เช่น

```
var scores = ["Math": 86, "Science": 89, "English": 75, "Art": 44]
var newScores = scores
newScores["Art"] = 51
```

จากการทำดังกล่าว print scores[“Art”] จะยังคงได้ 51

**Arrays Copy**

Array ไม่สามารถส่งผ่านค่าถึงกันได้ (unshare) เช่น

```
var a = [1, 2, 3]
var b = a
var c = a
a[0] = 42 จะได้ว่า  a[0] = 42 b[0] = 1 และ c[0] = 1 ด้วย
```

*แต่เราสามารถใช้ append เพื่อให้ค่าที่กำหนดใหม่อิสระจาก copy ได้ โดยกำหนดขนาด (length) ที่จะทำการ append เช่น*

```
a.append(4)
a[0] = 777
จะได้ว่า b[0] = 1 และ c[0]= 1
```

ถ้าต้องการจะเช็คว่า array ทั้งสองใช้ค่าเดียวกันอยู่หรือไม่ให้ใช้ === และ !=== เช่น

```
If a === c จะได้ false
  If b[0...1] === c[0...1] จะได้ true
```

## Properties

Properties เป็นค่าที่ใช้เชื่อมโยงกับ particular class, structure หรือ enumeration เมื่อสร้าง Inatance ขึ้นมา เราก็จะมี Properties ที่มีความสัมพันธ์ กับ class, structures หรือ enumerations นั้นติดมาด้วย

Properties แบ่งเป็น Stores Properties และ Computed Properties ทั้ง 2 แบบของแต่ละ Instance จะต้องอ้างอิงกับ Type ที่แน่นอน เช่น เป็น Int, String, Double เป็นต้น หรือกำหนดด้วย Type ด้วยการอ้างอิงกับตัวเอง เรียกว่า Type Property

สามารถกำหนด Property Observer เพื่อใช้ตรวจสอบว่า หากมีการเปลี่ยนแปลงค่าใน Property แล้วจะไปรันโค้ดอะไรบ้าง เช่น นำไปกำหนดค่าให้กับ Stored Property ภายใน Class หรือ Structure นั้นๆ หรือกำหนดค่าให้กับ Property ที่สืบทอดมาจาก Superclass อื่นๆได้

### Stored Properties

เป็น properties ที่ได้จากค่าของ Variable หรือ Constant ซึ่งเป็นส่วนประกอบของ Instance นั้นๆ โดย Stored Properties จะมีใช้กับ Class และ Structure เท่านั้น

- *Constant Stored Properties เวลาประกาศ Property จะต้องใช้ let*
- *Variable Stored Properties เวลาประกาศ Property จะต้องใช้ var*

ตัวอย่าง

```
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
```

**Lazy Stored Properties**

คือ Stored Property ที่ยังไม่มีการกำหนดค่าเริ่มต้น จนกว่าจะได้ถูกใช้งานในครั้งแรก โดย Lazy Stored Properties จะใช้กับ Variable เท่านั้น เพราะต้องอนุญาตให้มีการเปลี่ยนแปลงค่า จึงใช้กับ Constant ไม่ได้ (เพราะว่า Constant จะต้องมีค่าก่อนที่ Instance จะสร้างเสร็จเรียบร้อย)

การใช้ Lazy Stored Properties มีจุดมุ่งหมายดังนี้

- *ไม่ทราบค่าเริ่มต้นในตอนแรก แต่จะทราบเมื่อ Instance จะถูกสร้างเสร็จเรียบร้อยไปแล้ว เช่น Import ไฟล์ประเภทข้อความเข้ามา เราจะไม่ทราบว่า จำนวนตัวอักษรเริ่มต้นมีค่าเป็นเท่าไหร่ จนกว่าจะ Import ไฟล์นั้นเข้ามาเสร็จเรียบร้อย*
- *การคำนวณค่าให้กับ Stored Property มีความซับซ้อน Lazy Stored Properties ช่วยให้ไม่ต้องคำนวณค่าเมื่อยังไม่ได้ใช้งาน*

ตัวอย่าง

```
class DataImporter {
    var fileName = "data.txt"
    }
class DataManager {
  lazy var importer = DataImporter()
   var data = [String]()   }
let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
```

**Computed Properties**

เป็น properties ที่ได้จากการคำนวณค่า ซึ่งจะใช่กับ class, structures และ enumerations

ตัวอย่าง

```
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))")
```

**Property Observers**

ใช้ Property Observers ติดตามค่า Property เมื่อมีการเปลี่ยนแปลงค่า Property เราสามารถใช้ Property Observers ติดตามการเปลี่ยนแปลงดังกล่าวได้ เช่น หากมีการเปลี่ยนแปลงค่าแล้วจะไปทำคำสั่งใด เป็นต้น

สิ่งควรทราบอีกประการหนึ่งก็คือ Property Observers จะใช้กับ Stored Property ไม่ใช้กับ Lazy Stored Property และสามารถนำไปใช้กับ Computed Property ที่อยู่ใน Subclass ก็ได้ (ไม่ใช่ กับ Superclass เพราะสามารถตรวจสอบค่า Computed Property ได้จาก setter อยู่แล้ว)

Observers จะมีอยู่ 2 เหตุการณ์ คือ

*willSet* => *ก่อนเปลี่ยนแปลงค่า*

*didSet* => *หลังเปลี่ยนแปลงค่าไปแล้ว*

การใช้งาน Property Observers

*ในส่วนของ willSet จะมีการส่งผ่าน Parameter ที่เป็นค่าใหม่เข้าไป (newValue)*

*และในส่วนของ didSet ก็จะผ่าน Parameter ที่เป็นค่าเดิมเข้าไป (oldValue)*

ซึ่งเราสามารถนำเอา newValue และ OldValue ไปใช้งานได้ ตัวอย่างเช่น

```
class StepCounter {
    var totalSteps: Int = 0 {
    willSet(newTotalSteps) {
        print("About to set totalSteps to \(newTotalSteps)")
    }
    didSet {
        if totalSteps > oldValue  {
            print("Added \(totalSteps - oldValue) steps")
        }
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
//About to set totalSteps to 200
//Added 200 steps
stepCounter.totalSteps = 360
//About to set totalSteps to 360
//Added 160 steps
stepCounter.totalSteps = 896
//About to set totalSteps to 896
//Added 536 steps
```

**Global and Local Variables**

ตัวแประประเภท Global คือตัวแปรที่ประกาศภายนอก function, method, closure, or type context. ส่วนตัวแปรประเภท Local คือตัวแปรที่ประกาศภายใน function, method, or closure context ทั้งตัวแปร global and ที่เราพูดถึงในบทนี้ทั้งหมดนั้นเป็น *stored variables*

Stored variables ก็เหมือนกับ stored properties ให้จัดเก็บค่าบางประเภทและอนุญาตให้เปลี่ยนค่าและดึงค่าของข้อมูล

**Type Properties**

ในภาษา Swift คือ Property ของ Type ซึ่งจะเป็นการกำหนดคุณสมบัติ ที่ Type ไม่ใช่กำหนดที่ Instance ของ Type

โดยจะมี keyword สำหรับการกำหนด Type Properties อยู่ 2 ตัว คือ static และ class

- *static ใช้กับ value type (เช่น ใช้กับ structure และ enumeration)*
- *class ใช้กับ class type (เช่น ใช้กับ class)*

ตัวอย่าง

```
struct SomeStructure {
  static var storedTypeProperty = "Some value."
  static var computedTypeProperty: Int {
  return 1
  }
}
enum SomeEnumeration {
  static var storedTypeProperty = "Some value."
  static var computedTypeProperty: Int {
  return 6
  }
}
class SomeClass {
   static var storedTypeProperty = "Some value."
   static var computedTypeProperty: Int {
      return 27
   }
   class var overrideableComputedTypeProperty: Int {
      return 107
   }
}
```

## Methods

Methods คือฟังก์ชั่นที่เกี่ยวข้องกับคลาส โครงสร้าง มีจุดประสงค์ที่ชัดเจน ใช้ในการแก้ปัญหาต่าง ๆ ที่เป็นที่นิยม ส่วนวิธีการทำงานจะไม่เปิดเภยต่อภายนอก จะให้แต่ผลลัพธ์กลับมาเท่านั้น

### Instance Methods

คือ Method ที่อยู่ภายในคลาสใด ๆ หรือโครงสร้างใด ๆ จะคอยช่วยให้การทำงานเกี่ยวกับคลาสนั้น ๆ เป็นเรื่องง่ายขึ้น เช่นการเข้าถึงตัวแปรในคลาส หรือการแก้ไขค่าตัวแปร ซึ่งมีวิธีการเขียนเหมือนกับฟังก์ชั่น ดังตัวอย่าง

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
    count = 0
   }
}
```

จะเห็นได้ว่า method ต่าง ๆ ที่เห็นในคลาสก็จะมีรูปแบบเหมือนฟังก์ชั่นนั้นละ แต่พอมันถูกกำหนดขึ้นในคลาส เราก็จะเรียกมันว่า method แทน ซึ่งการเรียกใช้ method จำเป็นที่จะต้องเรียกจาก instance ที่เป็นของคลาสนั้น ๆ เพียงเท่านั้น เช่น

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.incrementBy(amount: 5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

จะเห็นได้ว่ามีการสร้าง instance ในคลาส Counter ในบรรทัดแรก ถ้าเราต้องการเรียกใช้ method ก็แค่ใช้ชื่อ instance แล้วตามด้วยชื่อ method ที่ต้องการจะใช้งาน โดยขั้นกลางด้วยสัญลักษณ์ dot(.) ซึ่ง instance จากคลาสอื่นจะไม่สามารถเรียกใช้ method ต่างคลาสกันได้

**Local and External Parameter Names for Methods**

โดยปกติแล้วเวลาที่เราสร้างฟังก์ชั่น เราจะต้องเป็นคนกำหนดเองว่าจะให้มีพารามิเตอร์เป็นแบบใด แต่ในเรื่องของ method ทาง

สวิฟท์จะกำหนดให้พารามิเตอร์ตัวแรกเป็นแบบ local เสมอ ส่วนพารามิเตอร์ตัวที่สองและตัวต่อ ๆ ไปจะถูกสร้างให้เป็นทั้งแบบ external และ local ให้โดยอัตโนมัติ ดังตัวอย่าง

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int, numberOfTimes: Int) {
        count += amount * numberOfTimes
    }
}
```

เป็นคนจัดการเรื่องต่าง ๆ ที่กล่าวไปข้างต้นให้เอง

```
let counter = Counter()
counter.incrementBy(amount: 5, numberOfTimes: 3)
// counter value is now 15
```

จากรูปด้านบนจะเห็นได้ว่าเวลาเราเรียกใช้ method นั้น Parameter name ของ method จะถูกนำมาใช้เป็น Agrument Label อัตโนมัติ

**Modifying External Parameter Name Behavior for Methods**

เวลาเรียกใช้งานเมธอด หากไม่ต้องการใส่ Agrument Label ตอนประกาศเมธอดต้องใส่ _ นำหน้าชื่อพารามิเตอร์ ไม่เช่นนั้นแล้ว Parameter name ของเมธอดขะถูกนำมาใช้เป็น Agrument Label อัตโนมัติ ดังนั้นเวลาเรียกใช้งานเมธอก ก็ต้องใส่ชื่อพารามิเตอร์ให้ถูกต้องเสมอ

**The self Property**

ในทุก ๆ instance (ออปเจกของคลาส) จะมี property ที่มีชื่อว่า self ไว้อ้างอิงถึง instance ของมันเอง

```
func increment() {
    self.count += 1
}
```

จาก method ด้านบนจะมีการใช้ค่าว่า self นั้นก็คืออ้างอิงค่า count ของ instance ตัวที่เข้ามานั้นเอง

ซึ่งปกติแล้ว ไม่จำเป็นต้องเขียน self ข้างหน้าก็ได้ เนื่องจากทางสวิฟท์จะคิดว่าคุณนั้นอ้างอิงตัว instance ที่เข้ามาเป็นตัวอ้างอิงปกติอยู่แล้ว ดังนั้นก็สามารถใช้แค่ count =+ 1 ก็ได้

แต่ถ้าเป็นกรณีแบบนี้

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOfX(x: Double) -> Bool {
        return self.x > x }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(x : 1.0) {
    print("This point is to the right of the line where x == 1.0")
}
```

แปรไน instance ที่มีชื่อว่า x เช่นกัน ดังนั้นในกรณีนี้จำเป็นที่จะต้องใช้ self เพื่ออ้างอิง ไม่งั้นตัวโปรแกรมอาจจะ Error ได้ เพราะไม่รู้ว่าเราหมายถึง x ตัวไหน เช่นในบรรทัดที่ 8 ตอนเรียกใช้ method เราส่งค่า 1.0 ให้กับ x ที่เป็นพารามิเตอร์ ดังนั้น self.x ก็จะเท่ากับ 4.0 นั้นเอง แต่ถ้าในกรณีนี้เราไม่เขียน self ทางสวิฟท์จะคิดว่าเราใช้ค่า x ที่เป็นพารามิเตอร์ทั้งคู่

**Modifying Value Types from Within Instance Methods**

โดยปกติแล้วใน struct และ enumeration เราจะไม่สามารถแก้ไขค่าตัวแปรประเภท value ได้ด้วย instance method แต่เรามีวิธีที่สามารถทำให้แก้ไขได้ นั้นก็คือการใส่คำว่า mutating เข้าไปข้างหน้า ก่อนทำการสร้าง method

ดังตัวอย่าง

```
struct MyScore{
    var score = 0
    mutating func addScore () {
        score = score + 1
    }
}
var MyExam = MyScore()
MyeExam.addScore()
```

จะเห็นได้ว่า method addScore มีคำว่า mutating ข้างหน้า ดังนั้นจึงแก้ไขค่า value ต่าง ๆ ใน instance ได้แล้ว จากบรรทัดที่ 7 ซึ่งสร้าง instance มีค่า MyExam = 0 จากนั้นบรรทัดที่ 8 ก็เรียกใช้ method ซึ่งผลลัพธ์จะกลายเป็นว่า instance นั้นมีค่า MyExam = 1 แทนแล้ว แต่ในกรณีที่เราไม่กำหนด mutating แล้วยังพยายามที่จะแก้ไขค่า value ทางสวิฟท์จะแจ้ง Error กลับมา

**Assigning to self Within a Mutating Method**

การใช้ self ร่วมกับ mutating ก็สามารถทำได้เช่นกัน ดังตัวอย่าง

```
struct MyScore {
    var score = 0
    mutating func addScore (_ inputScore: Int) {
        self = MyScore(score: score + inputScore)
    }
}
var test = MyScore(score:10)
test.addScore(9)
```

จะเห็นว่ารูปด้านบนมีการใช้ self ด้วย นั้นก็คือเมื่อทำการเรียก method นี้ จะมีการสร้างค่า MyScore ขึ้นมาใหม่เลย แล้วค่อยนำ self ไปอ้างอิงถึง ซึ่งผลลัพธ์ก็จะเหมือนกับในหัวข้อก่อน เพียงแต่มีการทำงานที่ไม่เหมือนกันเท่านั้นเอง อันก่อนคือการแก้ไขค่า แต่อันนี้คือการสร้างขึ้นมาใหม่แล้วค่อยนำไปอ้างอิงถึง ส่วนในเรื่องของ enumerations ก็สามารถใช้ self ได้เหมือนกัน เช่น

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
switch self {
        case .Off:
            self = .Low
        case .Low:
            self = .High
        case .High:
       self = .Off  }
  }
}
var openLight = TriStateSwitch.Low
openLight.next() //openLight is now equal to .High
openLight.next()//openLight is now equal to .Off
```

High และ Off ตามลำดับ

## Type Methods

ปกติแล้ว instance แต่ละตัว จะเป็นเอกเทศต่อกัน มีค่าเป็นของตัวเอง ไม่อ้างอิงค่าของ instance ตัวอื่น ๆ คุณสามารถสร้าง method ที่มีชนิดที่เรียกว่า type ได้โดยการเพิ่มคำว่า class ไปข้างหน้า method (สำหรับ class) แต่ถ้าคุณต้องการสร้าง type method สำหรับ struct และ enu จะเพิ่มคำว่า static แทน

```
class SomeClass {
   class func someTypeMethod() {
      // type method implementation goes here
   }}
SomeClass.someTypeMethod()
```

จากตัวอย่างด้านบนจะเป็นวิธีการเขียน type method และการเรียกใช้ ซึ่งต้องใช้ชื่อของคลาส ไม่ใช่ instance

พูดถึง type method อีกซักหน่อย สำหรับ self ข้างในการทำงานของ type method จะหมายถึงคลาสไม่ใช่ instance เนื่องจาก instance ไม่สามารถเรียกใช้ type method ได้ ซึ่งในส่วนของ struct และ enu คุณสามารถใช้ self ในการแยกแยะระหว่าง static propertirs กับ static method parameter ได้นั้นเอง เหมือนที่ทำกับ instance properties กับ instance method parameter นั้น

```
struct LevelTracker {
   static var highestUnlockedLevel = 1
   var currentLevel = 1
   static func unlock(_ level: Int) {
      if level > highestUnlockedLevel { highestUnlockedLevel = level }
   }
   static func isUnlocked(_ level: Int) -> Bool {
      return level <= highestUnlockedLevel
   }
```

```
    @discardableResult
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

ตัวอย่าง

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
```

จากรูปด้านบน ก็เหมือนเกม ๆ หนึ่ง ที่มีการเก็บสถิติและรายละเอียดผู้เล่นไว้ ซึ่งในเกมนี้ จะมีเลเวลด่านที่มีผู้เล่นปลดล็อกได้สูงสุดเก็บไว้เป็นสถิติ(highestUnlockedLevel) และก็จะมีเลเวลด่านปัจจุบันของผู้เล่นแต่ละคนเก็บไว้ด้วยเช่นกัน(currentLevel) ซึ่งถ้าผู้เล่น เล่นเกมไปเรื่อย ๆ เลเวลของด่านก็จะสูงขึ้นเรื่อย ๆ เมื่อผ่านในแต่ละเลเวล ก็จะมีการเรียกใช้ method unlock เสมอ เพื่อไปเช็คดูว่าเลเวลที่ผ่านนั้นสามารถเป็นสถิติของเกมได้หรือไม่ แล้วก็จะมีการเรียกใช้ method advance(to level) ซึ่ง

จะแก้ไขค่าเลเวลปัจจุบันของผู้เล่นคนนั้น ๆ เช่นกัน ซึ่งที่อธิบายมาตั้งเยอะก็เพื่อที่จะให้แยกออกระหว่าง ตัวแปรของเกม (struct) กับ ผู้เล่น (instance) ออกก็เท่านั้นเอง

```
var player = Player(name: "ArgyriOS")
player.complete(level: 1)
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// Prints "highest unlocked level is now 2"
```

highestUnlockedLevel ก็จะเป็น 2 นั้นเอง

```
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")
}
// prints "level 6 has not yet been unlocked"
```

จากรูปของผู้เล่นคนที่ 2 ซึ่งเขาต้องการที่จะโกงโดยการที่มาถึงก็ไปที่เลเวล 6 เลย ก็จะไม่สามารถทำได้ เนื่องจาก highestUnlockedLevel ยังอยู่ที่ 2 อยู่เลย อยู่ดี ๆ จะขึ้นไป 6 เลยก็จะไม่ได้ (ขึ้นอยู่กับว่าเราเขียนป้องกันการโกงได้ดีแค่ไหนด้วย)

## Subscripts

Classes structures และ enumerations สามารถกำหนด subscripts ซึ่งเป็นทางลัดสำหรับเข้าถึงองค์ประกอบหนึ่งของคอลเลกชัน รายการ หรือลำดับ คุณใช้ subscripts ตั้งค่า และเรียกค่าดัชนีโดยไม่จำเป็นวิธีการแยกต่างหากสำหรับการตั้งค่า และเรียก ตัวอย่าง คุณเข้าถึงองค์ประกอบใน Array อินสแตนซ์ เป็นsomeArray[index] และองค์ประกอบในพจนานุกรมตัวอย่างเป็น someDictionary[key].

คุณสามารถกำหนด subscripts หลายชนิดเดียว และ subscripts ที่เหมาะสมโอเวอร์โหลดจะใช้ไว้ตามชนิดของค่าดัชนีที่คุณส่งไป subscripts. Subscripts ไม่จำกัดขนาดเดียว และคุณสามารถกำหนด subscripts ด้วยหลายพารามิเตอร์ป้อนเข้าเพื่อให้เหมาะกับความต้องการของชนิดเอง

### Subscript Syntax

Subscripts ช่วยให้คุณสามารถสอบถามการอินสแตนซ์ของชนิด โดยการเขียนอย่าง น้อยหนึ่งค่าในวงเล็บสี่เหลี่ยมหลังชื่ออินสแตนซ์ ไวยากรณ์จะคล้ายกับวิธีการอินสแตนซ์ทั้งสอง ไวยากรณ์และไวยากรณ์คำนวณคุณสมบัติ คุณเขียนคำนิยาม subscripts ด้วย subscripts คำสำคัญ และระบุอย่าง น้อยหนึ่งพารามิเตอร์ที่ป้อนเข้าและชนิดการส่งคืนสินค้า วิธีการเดียวกัน เป็นวิธีการอินสแตนซ์ ซึ่งแตกต่างจากวิธีการอินสแตนซ์ subscripts สามารถอ่านและเขียน หรืออ่านเพียงอย่างเดียว โดยมีการทำงานนี้ และตัวเซ็ตเดียวเป็นการคำนวณคุณสมบัติ

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

ชนิดของ newValue จะเหมือนกับค่าที่ส่งคืนของ subscript ตามด้วยคำนวณ คุณสมบัติ คุณสามารถเลือกไม่ระบุที่ตัวเซ็ตของ (newValue) พารามิเตอร์ได้ ค่าเริ่มต้น พารามิเตอร์ที่เรียกว่า newValue ที่ให้แก่ตัวเซ็ตของคุณถ้าคุณไม่ให้ด้วยตนเองตามอ่านคำนวณคุณสมบัติ คุณสามารถให้ความสำคัญรับสำหรับอ่านอย่างเดียว

```
subscript(index: Int) -> Int {
    // return an appropriate subscript value here
}
```

นี่คือตัวอย่างการอ่านตัวห้อยดำเนินการ ซึ่งกำหนดเป็น TimesTable โครงสร้างถึง n-เท่าตารางที่มีจำนวนเต็ม

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let fiveTimesTable = TimesTable(multiplier: 5)
print("eight times five is \(fiveTimesTable[8])")
// prints "eight times five is 40"
```

ในตัวอย่างนี้ มีสร้างอินสแตนซ์ใหม่ของ TimesTable ถึงสามเวลาตารางนี้แสดง โดยช่วยให้ค่า 5 ตัวของโครงสร้างเป็นค่าที่จะใช้สำหรับอินสแตนซ์ตัวคูณพารามิเตอร์ คุณสามารถสอบถามอินสแตนซ์โดยการเรียก subscript ของ fiveTimesTable ดังที่แสดงในการเรียก fiveTimesTable [8] นี้ขอรายการเดียวในสามเวลาตารางซึ่งส่งกลับค่าการ 40 หรือ 5 ครั้งที่ 8

*NOTE*

*N-เท่าตารางที่ยึดตามกฎคณิตศาสตร์ถาวร ไม่เหมาะสมกับการตั้งค่า fiveTimesTable [someIndex] ค่าใหม่ และเพื่อ ให้ตัวห้อยสำหรับ TimesTable ถูกกำหนดเป็นแบบอ่านอย่างเดียว*

**Subscript Usage**

ความหมายของ " subscript " แน่นอนขึ้นอยู่กับบริบทที่ใช้ subscript จะใช้เป็นทางลัดสำหรับการเข้าถึงองค์ประกอบของสมาชิกในกลุ่ม รายการ หรือ ลำดับนั้น คุณมีอิสระในการใช้ตัวห้อยในลักษณะที่เหมาะสมที่สุดสำหรับคุณ คลาสเฉพาะหรือโครงสร้างของฟังก์ชัน ตัวอย่าง Swift พจนานุกรมชนิดใช้ตัวห้อยเพื่อตั้งค่า และเรียกค่าเก็บไว้ในอินสแตนซ์ที่พจนานุกรม คุณสามารถตั้งค่าในพจนานุกรม โดยการให้ความสำคัญของชนิดหลักของพจนานุกรมภายในวงเล็บตัวห้อย และการกำหนดค่าของการชนิดค่าตัวห้อย

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

ตัวอย่างข้างต้นกำหนดตัวแปรที่เรียกว่า numberOfLegs และเริ่มต้น ด้วยพจนานุกรม ตัวที่ประกอบด้วยสามคู่ค่าคีย์ สรุปชนิดของพจนานุกรม numberOfLegs เป็น พจนานุกรม. หลังจากสร้างพจนานุกรม ตัวอย่างนี้ใช้ตัวห้อย กำหนดให้เพิ่มคีย์สายของ "นก" และค่า Int 2 ในพจนานุกรม สำหรับข้อมูลเพิ่มเติมเกี่ยวกับ subscripting พจนานุกรม ดู Accessing Modifying เป็นพจนานุกรม

*NOTE*

*Subscripting ของค่าคีย์เป็น subscript ที่ใช้ และได้รับการเลือกใช้ชนิดของ Swift พจนานุกรม ชนิดของ สำหรับ numberOfLegs พจนานุกรมข้าง subscript คีย์ค่าใช้ และส่งกลับค่าชนิด Int ?, หรือ "เลือก int" ชนิดพจนานุกรมใช้ชนิด subscript ได้เลือกแบบจะไม่ครบทุกแป้น มีค่า และให้วิธีการลบค่าสำหรับคีย์การกำหนดค่าคีย์ที่ nil*

**Subscript Options**

Subscript สามารถใช้จำนวนพารามิเตอร์ป้อนเข้า และพารามิเตอร์เหล่านี้สามารถชนิดใดก็ subscript สามารถส่งคืนชนิดใดก็ subscript สามารถใช้พารามิเตอร์ตัวแปร และ พารามิเตอร์ variadic แต่ไม่สามารถใช้พารามิเตอร์ในเช็ค หรือให้พารามิเตอร์เริ่มต้นค่า

ชั้นหรือโครงสร้างที่สามารถให้ใช้งาน subscript มากเท่าที่จำเป็น และตัวห้อยที่เหมาะสมที่จะใช้จะสรุปตามชนิดของค่าหรือค่าที่อยู่ในวงเล็บตัวห้อยที่จุดที่ subscript ที่ใช้ นี้ คำจำกัดความของ subscript หลาย เรียกว่า subscript overloading

ในขณะที่พบมากที่สุดสำหรับตัวห้อยใช้พารามิเตอร์เดียว คุณยังสามารถกำหนด subscript มีหลายพารามิเตอร์ถ้าเหมาะสมกับชนิดของคุณ ต่อไปนี้ ตัวอย่างกำหนดโครงสร้างเมตริกซ์ ซึ่งแสดงเมตริกซ์สองของสองค่า เมตริกซ์ของโครงสร้างตัวห้อยจะเต็มสองพารามิเตอร์

```
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
        }
    }
```

เก็บแถว * ค่าคอลัมน์ของชนิดเดี่ยว แต่ละตำแหน่งในเมตริกซ์จะกำหนดค่าเริ่มต้นของ 0.0 เพื่อให้บรรลุนี้ อาร์เรย์ของ ขนาด และ อันแรก เซลล์ค่า 0.0 ส่งผ่านไปที่ตัวอาร์เรย์ที่สร้าง และเริ่มต้นแถวใหม่ ขนาดถูกต้อง ตัวนี้จะอธิบายในรายละเอียดเพิ่มเติมในการสร้างและการ Initializing อาร์เรย์ คุณสามารถสร้างอินสแตนซ์เมตริกซ์ใหม่ โดยผ่านการสมแถวและจำนวนคอลัมน์ต้องการตัว

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

ตัวอย่างก่อนหน้านี้สร้างอินสแตนซ์เมตริกซ์ใหม่ มีสองแถวและคอลัมน์ที่สอง ที่อ่านอาเรย์ตารางสำหรับนี้อินสแตนซ์เป็นอย่างมีประสิทธิภาพแต่ละรุ่นของเมตริกซ์ โดยอ่านเมตริกซ์ จากบนซ้ายลงล่างขวา

ทั้งนี้สามารถตั้งค่าในเมตริกซ์ โดยส่งค่าที่แถวและคอลัมน์เป็นตัวห้อย คั่น ด้วยเครื่องหมายจุลภาค

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

Subscript ตัวเซ็ตการตั้งค่า 1.5 ด้านบนขวาเรียกคำสั่งเหล่านี้สองตำแหน่งของเมทริกซ์ (ที่ 0 คือแถว และคอลัมน์คือ 1), และ 3.2 ในตำแหน่งซ้ายล่าง(แถว 1 และคอลัมน์เป็น 0)

ในเมตริกซ์ของมีและตัวเซ็ตทั้งการยืนยันการตรวจสอบว่าประกอบด้วยการค่าตัวห้อยของแถวและคอลัมน์ไม่ถูกต้อง ช่วย ด้วย assertions เหล่านี้ มีเมตริกซ์การวิธีสะดวกที่เรียกว่า indexIsValid การร้องขอแถวหรือคอลัมน์ว่านอกขอบเขตของเมทริกซ์

```
let someValue = matrix[2, 2]
// this triggers an assert, because [2, 2] is outside of the matrix bounds
```

## Inheritance

class สามารถสืบทอดคุณสมบัติ , Method และคุณลักษณะอื่นๆมาจาก class อื่นๆได้ class ที่สืบทอดมาจาก class เราจะเรียกว่า subclass และ class ที่ถูกสืบทอดนั้น เราจะเรียก superclass

class ใน Swift สามารถเรียกและเข้าถึง method, properties และ subscript จาก superclass ของตัวเองได้และยังสามารรถเขียน method , properties และ subscript เป็นของตัวเองได้อีกด้วย (overriding)

Swift จะช่วยเช็คว่าสิ่งที่ override ขึ้นมานั้นถูกต้องหรือเปล่าโดยการเช็คว่า match กับ superclass หรือไม่

**Defining a Base Class**

Base class คือ class ที่ไม่ได้สืบทอดมาจาก class อื่นๆ ตัวอย่างด้านล่างแสดงการ define base class ชื่อ Vehicle มี 2 properties คือ currentSpeed และ description ซึ่ง 2 properties นี้จะถูกเรียกใช้ใน method ที่ชื่อ makeNoise ที่จะ returns String ที่จะถูกแบ่งประเภทโดย subclasses ของ class Vehicle ในภายหลัง

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {    }
}
let someVehicle = Vehicle()
print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

การสร้าง new instance ของ Vehicle จะทำการเรียก initializer โดยเขียน TypeName ตามด้วย ()

```
let someVehicle = Vehicle()
```

initializer สำหรับ Vehicle ในเริ่มแรก

```
currentSpeed = 0.0
```

**Subclassing**

class ที่สืบทอดมาจาก superclass ซึ่งเราสามารถปรับแต่งเพิ่มเติมคุณสมบัติได้

การระบุว่า class สืบทอดมาหรือเปล่า? สืบทอดมาจากไหน? ทำได้โดย เขียนชื่อ superclass ไว้หลังชื่อ class แล้วคั่นด้วย colon:

```
class SomeSubClass: SomeSuperclass {
   // class definition goes here
}
```

ตัวอย่างต่อไปนี้เป็นการประกาศให้ vehicle เฉพาะเจาะจงเข้าไปอีก ชื่อ Bicycle class ใหม่นี้ก็มีพื้นฐานมาจาก Vehicle โดยใช้การสืบทอดเข้ามาช่วย

```
class Bicycle: Vehicle {
   var hasBasket = false
} let bicycle = Bicycle()
bicycle.hasBasket = true
bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

ได้รับคุณสมบัติจาก Vehicle ด้วย ได้แก่ currentSpeed และ description ซึ่งคุณสามารถตัดหรือเพิ่มคุณสมบัติให้กับ class Bicycle ได้ด้วย

Subclasses สามารถเป็น Subclasses ของตัวมันเองได้ Tandem สืบทอดทั้งเมธอดและพรอพเพอตี้จาก Bicycle ซึ่งสืบทอดจาก Vehivle อีกที โดยที่subclass tadem สามารถเพิ่มพรอพเพอตี้ใหม่ชื่อว่า currentNumberOfPassengers มีค่าเริ่มต้นเป็น 0

ซึ่งสามารถเขียนได้ดังนี้

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

**Overriding**

Subclass เขียนขึ้นมาใหม่ซ้ำกับของ superclass ซึ่งหากต้องการเรียก method ใน superclass เราสามารถทำได้โดยเติม prefix super ไปข้างหน้าตามด้วย .ชื่อ method

เช่น *super.somemethod*

**Overriding Methods**

คุณสามารถเขียน instance หรือ method ทับกับของที่มีใน superclass ได้ โดยอาจจะตัดหรือใช้ทางเลือกอื่นใน method เพื่อเอามาใช้ใน subclass

ตัวอย่างด้านล่างเป็นการประกาศ subclass ของ Vehicle หรือเรียก ที่เขียน overrides method ที่สืบทอดมาจาก Vehicle

```
class Train: Vehicle {
        override func makeNoise() {
         print("Choo Choo")
         }
}
let train = Train()
train.makeNoise()
// Prints "Choo Choo"
```

เราสามารถเขียน override get และ set ใน subclass ได้

```
class SpeedLimitedCar: Car {
  override var speed: Double  {
  get {
    return super.speed
  }
  set {
    super.speed = min(newValue, 40.0)
  }
  }
```

**Overriding Property Observers**

เราสามารถ override observer ใน subclass ได้ ดังตัวอย่างข้างล่างนี้ class Automatic Car เป็น subclass ของ car

ในส่วนของ Description สามารถระบุได้ว่าตอนนี้อยู่ gear ไหนแล้ว ซึ่ง gear จะถูก set ใน method didSet

```
class AutomaticCar: Car {
        override var currentSpeed: Double {
      didSet {
        gear = Int(currentSpeed / 10.0) + 1
      }}}
Let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

ทั้งนี้สามารถป้องกันไม่ให้ method , property หรือ subscript ไม่ให้ถูก override ได้โดยใช้ final วางไว้ข้างหน้าตัวที่ไม่ต้องการให้ถูก override และหาก final class แล้วจะทำให้ไม่สามารถสืบทอด class นั่นๆได้

## Initialization

Class และ structure ต้องมีการกำหนดคุณสมบัติไว้ในส่วนเริ่มต้น เราสามารถเซตค่าเริ่มต้นได้โดยใช้คุณสมบัติที่ถูกเก็บไว้ initializers

Initializers จะถูกเรียกเมื่อมีการสร้าง instance ขึ้นมาใหม่ หากไม่มี parameter init จะถูกเรียกโดยอัตโนมัติ

ตัวอย่างด้านล่างเป็นการกำหนด structure ขึ้นมาใหม่ชื่อ Fahrenheit เพื่อเก็บอุณหภูมิในหน่วยฟาเรนไฮต์ ซึ่งใน Fahrenheit structure ก็จะมีตัวแปร temperature ซึ่งมีชนิดเป็น Double

```
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"
```

เราสามารถกำหนดค่า default มีการกำหนดคุณสมบัติใน Fahrenheit structure ได้อีกแบบตามโค้ดข้างล่างนี้

```
struct Fahrenheit {
    var temperature = 32.0
}
```

**Initialization Parameters**

คุณสามารถสั่งค่าparameter ให้กับ initializer ได้นะ โดยกำหนด initializer ให้ตรง syntax กับ parameter ที่จะส่ง

ตัวอย่างโค้ดข้างล่างนี้ที่ init ต้องการค่า Double เพื่อคำนวณอุณหภูมิ

```
struct Celsius {
    var temperatureInCelsius: Double = 0.0
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
```

**Local and External Parameter Names**

ทั้ง function, method parameters, initialization parameters สามารถมีได้แบบ local เพื่อใช้ภายใน initializer body และแบบ externalสำหรับใช้เมื่อมีการเรียก initializer อย่างไรก็ดี initializers ไม่มีการระบุตัวตนของชื่อฟังก์ชั่นก่อนวงเล็บภายในฟังก์ชั่นเลย ดังนั้นชื่อและชนิดของ parameter ที่ initializer นั้นสำคัญมากว่า initializer ตัวไหนจะเป็นตัวที่ถูกเรียก เพราะเหตุนี้ Swift จึงให้ parameter ใน initializer ทั้งหมดเป็นแบบ external ถ้าคุณไม่ได้ระบุไว้ว่าจะให้ใช้แบบไหน

ตัวอย่างด้านล้างเป็นการกำหนด structure ชื่อ Color ที่มี 3 constant ได้แก้ red, green, blue ทั้ง 3 ค่านี้มีค่าอยู่ระหว่าง 0.0 ถึง 1.0 จะต้องส่งค่า Double ให้ initializer เพื่อกำหนดค่า

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
```

เมื่อคุณสร้าง color ใหม่ขึ้นมา คุณจะเรียก initializer ที่ใช้ Agrument Label

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

จะเห็นว่าเป็นไปไม่ได้เลยที่จะเรียกโดยไม่ใช่ Agrument Label ถ้าไม่ได้กำหนด Agrument Label จะทำให้เกิด error ดังตัวอย่าง

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - external names are required
```

**Optional Property Types**

ถ้า custom type ของคุณมีคุณสมบัติที่อนุญาตให้ไม่กำหนดค่าได้ อาจจะเพราะค่าดังกล่าวไม่สามารถถูกกำหนดได้ตอนเริ่มต้น หรืออาจเพราะอนุญาตให้ไม่มีค่าอะไรเลยซึ่ง ณ จุดนี้สามารถกำหนด

คุณสมบัติให้เป็น optional type ได้ คุณสมบัติของ optional type คือ การกำหนดค่าเริ่มต้นให้เป็น nil โดยอัตโนมัติ

ตัวอย่างด้านล่างเป็นการกำหนด class ชื่อ SurveyQuestion ด้วยคุณสมบัติ optional String property ชื่อ response

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
```

Respond ถูกประกาศเป็น type String? หรือ optional String นั่นเอง respond จะถูกกำหนดให้เป็น nil ตั้งแต่แรก จนกว่าจะมีการมากำหนดค่านี้ใหม่

**Default Initializers**

ถ้าไม่มีการกำหนด init ไว้ ค่าที่จะเอามาใช้กำหนดค่าเริ่มต้นเมื่อมีการสร้าง instance ก็จะเป็นค่าที่กำหนดไว้ให้แต่ละตัวแปรใน class นั้น

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

## Memberwise Initializers for Structure Types

ถึงแม้จะมี Default Initializer แต่ถ้าอยากจะกำหนดค่าเริ่มต้นเองก็ทำได้นะ

```
struct Size {
   var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

## Initializer Delegation for Value Types

Initializer สามารถที่จะเรียก initializer อันอื่นได้ เพื่อที่จะกำหนดค่าเริ่มต้นให้กับ instance เรา เรียก initializer delegation

ตัวอย่างด้านล่างเป็นการกำหนด Rect Structure ซึ่งจะต้องมี Size และ Point ทั้ง 2 ค่านี้มี

```
struct Size {
   var width = 0.0, height = 0.0
}
struct Point {
   var x = 0.0, y = 0.0
}
```

คุณสามารถกำหนดค่าเริ่มต้นให้กับ Rect Structure ได้ตามโค้ดข้างล่างนี้ ซึ่งกำหนดได้ 3 แบบ

1. ให้ origin และ size เป็น 0
2. กำหนดค่าเริ่มต้นให้ origin และ size
3. หรือกำหนดจุดกลางให้กับ center และ size

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
      self.init(origin: Point(x: originX, y: originY), size: size)
    }
```

**Designated Initializers and Convenience Initializers**

Designated Initializers เป็น initializer ขั้นแรกของ class โดยจะเรียก initializer ของ superclass โดยทั่วไปทุกๆ class มักจะมีแค่ 1 designated class แต่ในบางกรณี ก็อาจจะมีมากกว่า 1 designated ที่มาจาก superclass

Convenience initializers เป็น initializer รองที่เป็นส่วนเสริมของ class คุณสามารถกำหนดขึ้นเองได้

**Initializer Chaining**

เพื่อให้ง่ายในความสัมพันธ์ระหว่าง designated และ convenience initializer Swift ประยุกต์กฎ 3 ข้อ สำหรับเรียกใช้ initializer ระหว่างกัน

*Rule 1 : Designated initializer เรียก designated initializer จาก superclass ได้เพียง 1 ขั้นเท่านั้น*

*Rule 2 : convenience initializer เรียก initializer อื่นได้ภายใน class เดียวกันเท่านั้น*

*Rule 3 : convenience initializer ต้องจบด้วยการเรียก designated initialize*

จำง่ายๆคือ

1. Designated มักจะเรียก superclass
2. Convenience มักจะเรียกภายใน class

ซึ่งอธิบายได้ดังรูป

จากรูป superclass มี 1 designated initializer และมี 2 convenience มี convenience ที่เรียก convenience อีกตัว ซึ่งตัวนั้นก็เรียก designated ซึ่งเป็นไปตามกฎ 2 และ 3 แต่ 1 ไม่มีนะ เพราะไม่มี superclass

Subclass มี 2 designated และมี 1 convenience และ convenience จะเรียก designated ได้เพียงตัวใดตัวหนึ่งเท่านั้น ซึ่งเป็นไปตามกฎข้อ 2, 3 และ designated ทั้งคู่ เรียก designated ของ superclass ซึ่งเป็นไปตามกฎข้อ 1

**Two-Phase Initialization**

Initialization ใน Swift ประกอบด้วย 2 phase

Phase 1 เก็บคุณสมบัติที่กำหนดค่าเริ่มต้นให้กับ class เมื่อเก็บทุกอย่างครบแล้วจึงเริ่ม phase 2

การใช้ 2 phase initialization ทำให้มั่นใจว่า การเรียกใช้ initialization จะถูกป้องกันไม่ให้เกิดค่าที่ไม่พึงประสงค์

Swift compiler ใช้ 4 ตัวช่วยในการ check เพื่อให้มั่นใจว่า 2-phase initialization จะไม่เกิด error

Safety check 1: designated initializer จะต้องมั่นใจว่ารู้จักคุณสมบัติครบทุกอันภายใน class ตัวเองก่อนที่จะเรียก initializer ของ superclass

Safety check 2: convenience initializer ต้องเรียก initializer ตัวอื่นๆก่อนจะกำหนดค่าให้กับทุกคุณสมบัติ ถ้าไม่ทำแบบนั้นจะทำให้ค่าที่ถูกกำหนดเกิดการเขียนทับกัน

Safety check 3: designate initializer ต้องเรียก initializer จาก superclass ก่อนที่จะกำหนดค่าให้กับทุกคุณสมบัติ ถ้าไม่ทำแบบนั้นจะทำให้ค่าที่ถูกกำหนดเกิดการเขียนทับกัน

Safety check 4: initializer ไม่สามารถที่จะเรียก instance method ได้ ไม่ว่าจะเป็นค่าหรืออ้างอิงค่าโดยใช้ seft จนกว่าจะเสร็จสิ้น phase 1

ทีนี้มาดูกันว่า two-Phase Initialization ทำงานโดยใช้ 4 safety-check ยังไง

**Phase 1**

Designated หรือ convenience ที่ถูกเรียกจาก class หน่วยความจำสำหรับ new instance จะถูกจองไว้ แต่ยังไม่ได้ใส่ค่าลงไปนะ

Designated จะต้องยืนยันว่า properties ที่ถูกเก็บไว้ครบแล้วหรือยัง ซึ่งในส่วนนี้จะเริ่มเก็บค่า properties แล้ว

Designated จะไปถึงค่าของ properties ของ superclass มาเก็บไว้ ซึ่งในส่วนนี้จะทำจนกว่าจะถึง superclass ชั้นบนสุด

เมื่อไปถึงชั้นบนสุดแล้ว และที่ปลายสุดถึงเก็บค่าให้คุณสมบัติแล้วเป็นอันเสร็จสิ้น phase 1

**Phase 2**

การทำงานจะกลับจากชั้นบนสุดมนแต่ละ designated initializer ในตอนนี้สามารถใช้ self แก้ไขคุณสมบัติของตัวเองและเรียก instance method ได้แล้ว ในท้ายที่สุด ทุกๆ convenience initializer ในสายจะสามารถกำหนดค่าinstance ได้ด้วย self

## Deinitialization

Deinitializer คือ การเรียกทันที ก่อนอินสแตนซ์ของคลาสจะแบ่งสัดส่วน(deallocated) คุณเขียน deinitializers กับคำว่า deinit ที่เป็น keyword คล้ายกับวิธีเขียน intializers กับคำสำคัญ init ที่เป็น keyword เริ่มต้น Deinitializers มีเฉพาะชนิดของคลาส

### How Deinitialization Works

Swift deallocates อินสแตนซ์ของคุณโดยอัตโนมัติเมื่อไม่จำเป็นต้องเพิ่มทรัพยากร Swift จัดการหน่วยความจำของอินสแตนซ์ผ่าน automatic reference counting (ARC), อธิบายไว้ในหัวข้อ Automatic Recerence Counting โดยทั่วไปคุณไม่ต้อง clean-up เมื่อ instances ไม่ถูกใช้( deallocated) อย่างไรก็ตามเมื่อคุณกำลังทำงานกับทรัพยากรของคุณเอง คุณอาจจำเป็นต้องทำบางอย่างเพิ่มเติม ตัวอย่างเช่น ถ้าคุณสร้าง custom class เพื่อเปิดไฟล์ และเขียนข้อมูลบางอย่าง คุณอาจต้องการปิดไฟล์ก่อนที่ คลาส instance จะ deallocated

คลาสหนึ่งคลาสจะมี deinitializer ได้มากที่สุดเพียงหนึ่ง deinitializer ต่อคลาส Deinitializer ไม่มีพารามิเตอร์ใด ๆ และเขียนโดยไม่ต้องมีวงเล็บ:

```
deinit {
// perform the deinitialization
}
```

Deinitializers ถูกเรียกโดยอัตโนมัติ ก่อนที่ instance จะ deallocation คุณไม่สามารถเรียก deinitializer ด้วยตัวเอง Deinitializers ซุปเปอร์คลาสถ่ายทอดคุณสมบัติให้กับ subclass และ superclass deinitializer ถูกเรียกโดยอัตโนมัติเมื่อสิ้นสุดการดำเนินการ Superclass deinitializers ถูกเรียกเสมอแม้ว่า subclass จะไม่มี deinitializer ของตัวเอง

เนื่องจาก instance ไม่ถูก deallocated จนกว่า deinitializer จะถูกเรียก, deinitializer ที่สามารถเข้าถึงคุณสมบัติทั้งหมดของอินสแตนซ์ที่ ถูกเรียก และสามารถปรับเปลี่ยนลักษณะการทำงานของมันตามคุณสมบัติ (เช่นค้นหาชื่อของไฟล์ที่ต้องถูกปิด)

**Deinitializers in Action**

นี่คือตัวอย่างของ deinitializer in action ตัวอย่างนี้กำหนดสอง type ใหม่ คือ Bank และ Player สำหรับเกมง่าย ๆ โครงสร้าง Bank จัดการสกุลเงินสมมติ ซึ่งไม่สามารถมีมากกว่า 10000 เหรียญ ในการหมุนเวียน ดังนั้น Bank เป็นโครงสร้างที่มีคุณสมบัติเป็น static และจัดเก็บ สถานะปัจจุบัน:

```
class Bank {
   static var coinsInBank = 10_000
   static func distribute(coins numberOfCoinsRequested: Int) -> Int {
      let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
      coinsInBank -= numberOfCoinsToVend
      return numberOfCoinsToVend
   }
   static func receive(coins: Int) {
      coinsInBank += coins
   }
```

- Bank เก็บจำนวนเหรียญ ณ ปัจจุบัน จะเก็บ คุณสมบัติ coinsInBank ยังมีวิธีการอีกสองวิธี คือ distribute การจัดการแจกจ่าย และ receive การเรียกเก็บเงินเหรียญ
- distribute ตรวจสอบว่า มีเหรียญพอที่ธนาคารจะกระจาย ถ้าเหรียญไม่เพียงพอ ธนาคารส่งกลับตัวเลขน้อยกว่าจำนวนที่ร้องขอ (และส่งคืนศูนย์ถ้าเหรียญไม่มีเหลือในธนาคาร) distribute ประกาศ numberOfCoinsRequest เป็นพารามิเตอร์ ตัวแปรเพื่อให้สามารถปรับเปลี่ยนหมายเลขภายใน method body โดยที่ไม่จำเป็นต้องประกาศตัวแปรใหม่ จะส่งกลับค่าจำนวนเต็มเพื่อแสดงจำนวนเหรียญที่ได้จัดเตรียมไว้
- method receive เพิ่มจำนวนเหรียญไว้เก็บเหรียญของธนาคาร
- Player classอธิบายผู้เล่นในเกม แต่ละคนมีจำนวนเหรียญที่เก็บไว้ในกระเป๋าเงินของพวกเขาตลอดเวลา จะแสดงคุณสมบัติของ coinsInPurse:

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
```

แต่ละ Player instance จะ ด้วยค่าเริ่มต้นของจำนวนเหรียญจากธนาคารที่ระบุในระหว่างการเริ่มต้น แม้ว่า Player instance อาจได้รับจำนวนเหรียญไม่เพียงพอ

Player class จะกำหนด method win ซึ่งดึงจำนวนเหรียญจากธนาคาร และเพิ่มเงินใส่กระเป๋าเงินของผู้เล่น Player class ยังใช้ deinitializer ซึ่งคือ Player class จะถูกเรียกก่อนที่จะ deallocated ถึงตอนนี้ deinitializer จะ คืนเหรียญทั้งหมดของผู้เล่นให้กับธนาคาร

```
var playerOne: Player? = Player(coins: 100)
print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// prints "A new player has joined the game with 100 coins"
print("There are now \(Bank.coinsInBank) coins left in the bank")
// prints "There are now 9900 coins left in the bank"
```

Player instance ถูกสร้างขึ้นมาใหม่ และถูกร้องขอจำนวน 100 เหรียญถ้ามีอยู่ Player instance นี้ถูกเก็บไว้ในตัวแปร Player ที่เรียกว่า playerOne ตัวแปรที่เลือกจะใช้ที่นี่ เนื่องจากผู้เล่นสามารถออกจากเกมได้ตลอดเวลา ตัวเลือกที่ช่วยให้คุณสามารถติดตามว่าปัจจุบันมีผู้เล่นในเกม เพราะ playerOne เป็นตัวเลือก, มันมีคุณสมบัติที่มีเครื่องหมาย (!) เมื่อมีการเข้าถึงคุณสมบัติ coinsInPurse เพื่อพิมพ์หมายเลขเริ่มต้นของเหรียญ และเมื่อใดก็ ตามที่เรียก

method win:

```
playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// prints "PlayerOne won 2000 coins & now has 2100 coins"
print("The bank now only has \(Bank.coinsInBank) coins left")
// prints "The bank now only has 7900 coins left"
```

ผู้เล่นได้รับรางวัล 2000 เหรียญ ตอนนี้กระเป๋าเงินของผู้เล่นมีเหรียญ 2,100 และธนาคารมีเหรียญ 7,900 เหรียญเท่านั้นที่เหลือ

```
playerOne = nil
print("PlayerOne has left the game")
// prints "PlayerOne has left the game"
print("The bank now has \(Bank.coinsInBank) coins")
// prints "The bank now has 10000 coins"
```

ผู้เล่นได้ออกจากเกมในขณะนี้ โดยการตั้งค่าตัวแปร playerOne ให้เป็น nil หมายความว่า "ไม่มี Player instance" ในจุดที่ว่านี้ ตัวแปร playerOne ที่อ้างถึง Player instance เกิดความเสียหาย คุณสมบัติหรือตัวแปรอื่นยังคงอ้างถึง Player instance และ มันก็จะ deallocated เพื่อเพิ่มหน่วยความจำ ก่อนเกิดเหตุการณ์นี้ deinitializer ถูกเรียกโดยอัตโนมัติ และเหรียญจะถูกส่งกลับไปธนาคาร

## Automatic Reference Counting

สวิฟท์ใช้ Automatic Reference Counting (ARC) เพื่อติดตามและจัดการการใช้งานหน่วยความจำของ แอพ ส่วนใหญ่จะหมายถึง การจัดการหน่วยความจำจะทำงาน ในฉับพลัน และคุณไม่จำเป็นต้องคิด เกี่ยวกับการจัดการตัวเองของหน่วยความจำ ARC จะจัดการให้โดยอัตโนมัติ โดยใช้ class instances อย่างไรก็ตาม ในบางกรณี ARC ต้องการข้อมูลเพิ่มเติมเกี่ยวกับความสัมพันธ์ระหว่าง ส่วนของ โค้ด เพื่อจัดการกับหน่วยความจำ ในบทนี้จะอธิบายถึงสถานการณ์ และจะแสดงให้เห็นว่า คุณจะสามารถจัดการกับหน่วยความจำของแอพได้อย่างไร

### How ARC Works

ทุก ๆ ครั้งที่สร้าง instance ใหม่ ARC จะทำการจองหน่วยความจำเพื่อจัดเก็บข้อมูล ของ instance นั้น หน่วยความจำนี้จะเก็บข้อมูลเกี่ยวกับ ชนิดของ instance นั้น และคุณสมบัติอื่น ๆ ด้วย

นอกจากนี้ เมื่อ instance ไม่ได้ใช้ ARC จะจัดการให้หน่วยความจำสามารถใช้สำหรับวัตถุประสงค์อื่นแทน เพื่อให้มั่นใจว่า instance ไม่ใช้พื้นที่ในหน่วยความจำ เมื่อไม่จำเป็น

อย่างไรก็ตาม ถ้า ARC ไม่ได้จองพื้นที่หน่วยความจำ instance ที่ใช้จะไม่มีสิทธิ์เข้าถึงคุณสมบัติ หรือ เรียก method ของ instance ได้ แน่นอน ถ้าคุณพยายามที่จะเข้าถึงสิทธิ์ในการใช้ instance แอพ ของคุณอาจจะผิดพลาด

**ARC in Action**

ตัวอย่างของ Automatic Reference Counting

```
class Person {
let name: String
init(name: String) {
  self.name = name
  print("\(name) is being initialized")
}
deinit {
  print("\(name) is being deinitialized")
}
```

ตัวอย่างโค้ด ถัดมา

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

คุณสามารถสร้าง Person instance ใหม่และกำหนดมันให้กับตัวแปรทั้งสามตัว

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

เพราะ Person instance ถูกกำหนดให้ ตัวแปร reference1 ARC ทำให้มันใจว่า Person ถูกเก็บในหน่วยความจำแล้ว

ถ้าคุณกำหนดค่าที่เหมือนกันให้กับ ตัวแปร Person ให้กับอีกสองตัวแปร เป็น สองตัวแปรที่เรียกว่า

```
strong references
reference2 = reference1
reference3 = reference1
```

ขณะนี้มี ตัวแปรสามตัวที่เป็น strong references

ถ้าคุณแบ่งสองตัวแปรที่เป็น strong references โดยกำหนดค่าให้เป็น nil อีกหนึ่งตัวแปรที่เหลือยังคงเป็น strong reference เพราะฉะนั้น Person instance จะไม่จอง(deallocated)

```
reference1 = nil
reference2 = nil
```

ARC จะไม่จอง Person instance จนกว่าตัวแปรตัวที่สาม reference3 กับตัวแปร strong reference จะไม่ถูกต้องสมบูรณ์ ,เป็นที่ชัดเจนว่าคุณไม่ได้ใช้ Person instance:

```
Reference3 = nil
// prints "John Appleseed is being deinitialized"
```

**Strong Reference Cycles Between Class Instances**

จากตัวอย่างข้างบน, ARCสามารถติดตามจำนวนของ ตัวแปร Person instance ที่คุณสร้างขึ้น และ deallocate ตัวแปร Persone instance เมื่อไม่ได้ใช้

อย่างไรก็ตาม เป็นไปได้ที่จะเขียนโค้ดที่ instance ของ คลาสไม่เคยได้จุดที่มี zero strong references สามารถเกิดขึ้นได้หากสองคลาสมีตัวแปร strong reference เป็นที่รู้จักกันคือ strong reference cycle

คุณสามารถแก้ไข strong reference cycles ได้โดยกำหนดบางส่วนของความสัมพันธ์ระหว่างคลาส ที่ weak หรือ unowned references แทนที่จะเป็น strong references กระบวนการนี้จะอธิบายการแก้ปัญหาระหว่าง Strong Reference Cycles กับ Class Instances อย่างไรก็ตาม ก่อนที่คุณจะเรียนรู้ถึงการแก้ปัญหาของ strong reference cycle จะเป็นประโยชน์มากต่อการทำความเข้าใจ วิธีการของ ที่ก่อให้เกิด cycle

นี่คือตัวอย่างของ strong reference cycle ที่สร้างขึ้น ในตัวอย่าง กำหนดสองคลาส ที่มีชื่อว่า Person และ Apartment

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

ทุก ๆ Person instance มี name เป็นตัวแปรประเภท string และมี apartment ที่มีค่าเป็น nil apartment มีคุณสมบัติเป็น ตัวเลือก(optional) เพราะ บุคคล (person) อาจจะไม่มี อพาร์ทเม้นท์เสมอไป

ในทำนองเดียวกัน ทุก ๆ Apartment instance จะมี number ที่เป็นตัวแปรประเภท Int และมี tenant มีค่าเริ่มต้นเป็น nil tenant เป็นตัวเลือก (optional) เพราะว่า อพาร์ทเม้นท์ อาจจะไม่จำเป็นต้องมีผู้เช่า (tenant) เสมอไป

โค้ดต่อไป จะเป็นการกำหนด สองตัวแปร ที่มีชื่อว่า john และ unit4A ซึ่งจะถูกกำหนดเป็น ตัวแปรเฉพาะ คือตัวแปร Apartment และ Person ทั้งสองตัวแปรนี้ จะมีค่าเริ่มต้นเป็น nil

```
var john: Person?
var unit4A: Apartment?
```

ตอนนี้คุณสามารถสร้างตัวแปรเฉพาะ Person instance และ Apartment instance และกำหนดตัวแปร john และ unit4A

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

นี่คือ strong reference หลังจากการสร้างและการกำหนด สอง instance คือตัวแปร john ตอนนี้เป็น strong reference มีค่าเป็น Person instance และอีกตัวแปรคือ unit4A ก็เป็น strong reference ซึ่งมีค่าเป็น Apartment instance

ถึงตอนนี้สามารถเชื่อม สองตัวแปรเข้าด้วยกัน ดังนั้น person มีคุณสมบัติเหมือน apartment และ apartment มีคุณสมบัติเหมือน tenant โปรดสังเกต เครื่องหมาย ! ที่อยู่ข้างหลัง ตัวแปร john และ unit4A

*john!.apartment = number73*

```
john!.apartment = unit4A
unit4A!.tenant = john
```

หลังจากเชื่อมตัวแปรทั้งสองตัวแปรเข้าด้วยกัน

แต่เมื่อเชื่อมสองตัวแปร ที่เป็น strong reference คือ Person instance ที่มีคุณสมบัติเหมือนกับตัวแปร Apartment และ ตัวแปร Apartment ก็มีคุณสมบัติเหมือนกับ ตัวแปร Person ดังนั้น เมื่อคุณ break strong refernces โดย ตัวแปร john และ unit4A แล้ว การอ้างอิงถึงตัวแปรห้ามเป็น zero และ instances ไม่อนุญาตให้ deallocated โดย ARC:

*john = nil*

*number73 = nil*

*Note*

*เมื่อ deinitializer ถูกเรียกตอนที่คุณกำหนดค่าให้กับ สองตัวแปรมีค่าเป็น nil strong reference cycle จะถูกป้องปัน โดย ตัวแปร Person และ ตัวแปร Apartment จากการ deallocated ก่อให้เกิดการรั่วไหลของหน่วยความจำใน แอพของคุณ*

หลังจากที่คุณกำหนดค่าใหห้กับตัวแปล john และ unit4A ให้มีค่าเป็น nil:

## Resolving Strong Reference Cycles Between Class Instances

Swift มีสองวิธีที่จะแก้ไข strong reference cycles เมื่อคุณทำงานกับคุณสมบัติของประเภทของคลาส: weak references และ unowned references

Weak และ unowned references สามารเป็นตัวแปรที่ใช้อ้างอิงถึง ตัวแปรอื่น ๆ ได้ โดยไม่เก็บค่าเอาไว้ ตัวแปรนี้สามารถอ้างถึงตัวแปรอื่น ๆ ได้โดยไม่ต้องสร้าง strong reference cycle

เมื่อใช้ weak reference เมื่อไหร่ก็ตามที่อ้างอิงให้เป็น nil ในทางกลับกัน ใช้อ้างอิง unowned reference เมื่อคุณทราบว่าตัวแปรที่ใช้อ้างอิงไม่เคยเป็น nil เลย จะไม่มีการตั้งค่าในการเริ่มต้น

## Weak References

ตัวอย่างข้างล่างจะเหมือนกับตัวอย่างของ Person และ Apartmentจากตัวอย่างข้างต้น แต่สิ่งที่แตกต่างกันก็คือ ประเภทของ Apartment และ คุณสมบัติของ tenant จะถูกประกาศเป็น Weak References

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

อินสแตนซ์ที่ถูกสร้างขึ้นก่อน

```
var john: Person?
var unit4A: Apartment?
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
```

Person instance ยังคงเป็น strong reference ของ Apartment instance แต่ Apartment instance เป็น weak reference ของ Person instance ซึ่งหมายความว่า เมื่อคุณ break strong reference โดย ตัวแปร john จะทำให้ไม่มี strong reference ของ Person instance เพราะไม่มี strong references ของ Person instance มันจึง deallocated

```
john = nil
// prints "John Appleseed is being deinitialized"
```

strong reference ของตัวแปร Apartment instance ที่เหลืออยู่ มาจากตัวแปร unit4A ถ้าคุณแบ่ง strong reference ก็จะไม่มี strong references ของ Apartment instance

เพราะว่าไม่มี strong references ของ Apartment instance ดังนั้นมันจึงไม่ deallocated:

```
Unit4A = nil
// prints "Apartment 4A is being deinitialized"
```

instance ได้แสดงข้อความ “deinitialized” หลังจากที่ตัวแปร john และ number73 ถูกกำหนดค่าให้เป็น nil การพิสูจน์นี้แสดงว่า reference cycle ได้ถูกทำลายไป

## Unowned References

Unowned Refernces คล้ายกับ weak references คือ unowned references จะไม่เก็บ strong instance ที่มันอ้างถึง ข้อแตกต่างกับ weak references คือ unowned reference จะมีค่าเสมอ เพราะฉะนั้น unknowed reference จำต้องประกาศให้เป็น non-optional type เสมอ คุณสามารถบอกว่าตัวแปรไหนเป็น unowned reference โดยการวางคำว่า unowned ไว้หน้า property หรือ ก่อนการประกาศตัวแปร เนื่องจาก unowned reference เป็น non-optional จึงไม่จำเป็นต้อง unwrap ทุกครั้งเวลาใช้ unowned reference สามารถเข้าถึงได้โดยตรง อย่างไรก็ตาม ARC ไม่สามารถ กำหนดให้ reference เป็น nil เมื่อตัวแปรที่อ้างถึง deallocated เพราะว่า ตัวแปรที่เป็น non-optional type จึงไม่สามารถกำหนดให้ค่าเป็น nil ได้

> *Note*
> *ถ้าคุณพยายามที่จะเข้าถึง unowned reference หลังจากที่ตัวแปรที่ถูกอ้างถึงไม่ได้ถูกใช้แล้ว จะทำให้เกิด runtime error ใช้ unowned reference ก็ต่อเมื่อแน่ใจแล้วเท่านั้นว่า reference จะต้องอ้างถึงตัวแปร*

ภาษา Swift รับประกันว่า แอพของคุณอาจเสียหายได้ ถ้าคุณพยายามที่จะเข้าถึง unowned referece หลังจากตัวแปรที่ถูกอ้างถึงไม่ได้ถูกใช้แล้ว เพราะฉะนั้นอย่าทำ

ตัวอย่างด้านล่างจะเป็นการกำหนด 2 คลาสคือ Customer และ CreditCard เป็นความสัมพันธ์ระหว่าง bank customer กับ Credit card ของ customer ทั้งสองคลาส จะเก็บตัวแปรของอีกคลาส ความสัมพันธ์นี้อาจจะทำให้เกิด strong reference cycle

ความสัมพันธ์ระหว่าง Customer และ CreditCard แตกต่างจากความสัมพันธ์ ระหว่าง Apartment กับ Person เล็กน้อยดังที่เห็น จากตัวอย่างใน weak reference ในความสัมพันธ์นี้ customer อาจจะมี หรือไม่มี creditcard ก็ได้ แต่ credit card จำเป็นต้องมี customer คือ Customer class มี card optional property แต่ CreditCard class มี non-optional property

นอกจากนี้ การสร้างตัวแปร CreditCard ขึ้นมาใหม่ สามารถทำได้โดยใช้ค่า number และตัวแปร customer ที่จะกำหนด CreditCard การทำแบบนี้จะทำให้เวลาสร้างตัวแปร CreditCard จะมีตัวแปร Customer มาเกี่ยวข้องด้วยเสมอ

เนื่องจาก credit card จะต้องมี customer เสมอ จึงต้องประกาศ customer property ของ credit card ให้เป็น unowned reference เพื่อเลี่ยง strong reference cycle

```
class Customer {
   let name: String
   var card: CreditCard?
   init(name: String) {
            self.name = name
   }
   deinit { print("\(name) is being deinitialized") }
}
class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
       self.number = number
       self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
```

โค้ดต่อไปนี้กำหนดตัวแปรลูกค้าไม่จำเป็นที่เรียกว่าจอห์น ซึ่งจะถูกใช้เพื่อเก็บการอ้างอิงลูกค้าเฉพาะ ตัวแปรนี้มีค่าเริ่มต้นของ nil อาศัยถูกเลือก

```
var  john: Customer?
```

คุณสามารถสร้าง Customer instance ของลูกค้าได้ทันที และใช้ในการเริ่มต้นและกำหนด CreditCard instanceใหม่ เป็นคุณสมบัติที่ card ลูกค้า

```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

Customer instance ขณะนี้มีการอ้างอิงแบบ strong กับ CreditCard instance และ CreditCard instance มีการอ้างอิงแบบ unowned กับ Customer instance

เนื่องจากการอ้างอิงแบบ unowned กับ customer เมื่อคุณแบ่งการอ้างอิงแบบ strong โดยตัวแปรของ john ไม่มีการอ้างอิงแบบ strong กับ Customer instance

เนื่องจากไม่มีการอ้างอิงแบบ strong กับ Customer instance ซึ่งก็คือ deallocated หลังจากเกิดสิ่งนี้ขึ้น และไม่มีการอ้างอิงแบบ strong กับ CreditCard instance และมันก็เป็น deallocated:

```
john = nil
// prints "John Appleseed is being deinitialized"
// prints "Card #1234567890123456 is being deinitialized"
```

โค้ดสุดท้ายข้างต้นแสดงว่า deinitializers สำหรับ Customer instance และ CreditCard instance ทั้งพิมพ์ข้อความ "deinitialized" หลังจากตั้งค่าตัวแปร john เป็น nil นั่นเอง

## Unowned References and Implicitly Unwrapped Optional Properties

ตัวอย่างสำหรับอ้างอิงแบบ weak และแบบ unowned ข้างต้นครอบคลุมสองสถานการณ์ทั่วไปซึ่งจำเป็นต้องตัดวงจรการอ้างอิงแบบ strong

ตัวอย่าง Person และ Apartment แสดงให้เห็นถึงสถานการณ์ที่สองคุณสมบัติ ซึ่งทั้งสองได้รับอนุญาตให้เป็น nil มีศักยภาพในการทำให้วงจรอ้างอิงแบบ strong สถานการณ์นี้เป็นส่วนแก้ไขการอ้างอิงแบบ weak

ตัวอย่าง Customer และ CreditCard แสดงสถานการณ์ที่ที่แห่งหนึ่งที่ได้รับอนุญาตให้เป็น nil และอีกแห่งหนึ่งที่คุณสมบัติไม่สามารถเป็น nil ได้แต่มีคุณสมบัติในการทำให้เกิดวงจรอ้างอิงแบบ strong สถานการณ์นี้ได้รับการแก้ไขที่ดีที่สุดจากการอ้างอิงแบบ unowned แล้ว

อย่างไรก็ตาม มีสถานการณ์สมมติอยู่สามสถานการณ์ ซึ่งคุณสมบัติทั้งสองควรมีค่า และไม่มีคุณสมบัติจะเป็น nil เมื่อ initialization เสร็จสมบูรณ์แล้ว ในสถานการณ์สมมตินี้ ประโยชน์รวมถึงคุณสมบัติของ unowned ในระดับหนึ่งที่มีคุณสมบัติเป็นตัวเลือกที่ต้องการ โดยปริยายในระดับอื่น ๆ

คุณสมบัตินี้จะช่วยให้ทั้งสองจะได้รับการเข้าถึงได้โดยตรง (โดยไม่ต้องแกะตัวเลือก) เริ่มต้น initialization เสร็จสมบูรณ์แล้ว ขณะที่ยังคงหลีกเลี่ยงวงจรอ้างอิง ในส่วนนี้จะแสดงให้คุณเห็นวิธีการตั้งค่าความสัมพันธ์ดังกล่าว

ตัวอย่างด้านล่างกำหนดสอง class คือ myBook และ Description ซึ่งเก็บ instance ของ class อื่น ๆ ที่มีคุณสมบัติที่ต้องการ ในรูปแบบข้อมูลนี้ ทุก book ต้องมี writer และทุก Description จะต้องเป็น book การแสดงนี้ myBook class มีคุณสมบัติ bookDetail และ Description class มีคุณสมบัติเป็น book

```
class myBook {
    let name: String
    var bookDetail: Description!
    init(name: String, writer: String) {
        self.name = name
        self.bookDetail = Description(name: writer, book: self)
    }
}
class Description {
    let name: String
    unowned let book: myBook
    init(name: String, book: myBook) {
        self.name = name
        self.book = book
```

การตั้งค่าความเชื่อมโยงกันระหว่าง 2 class ตัว initializer สำหรับ Desciption ใช้ myBook instance และเก็บ instanceนี้ในคุณสมบัติของ book

ตัว initializer สำหรับ Description จะเรียกจากภายในตัว initializer สำหรับ myBook อย่างไรก็ตาม ตัวตัว initializer สำหรับ myBook ไม่สามารถส่ง self ไปยัง Description initializer จนกว่า myBook instance ใหม่จะ initialized ทั้งหมด อธิบายใน Two-Phase Initialization

เพื่อรับมือกับความต้องการนี้ คุณต้องกำหนดคุณสมบัติ bookDetail ของ myBook ซึ่งเป็นตำแหน่งตัวเลือกคุณสมบัติ ระบุด้วยเครื่องหมายอัศเจรีย์ในตอนท้ายของประเภทคำอธิบายประกอบ ( Description! ) หมายความ ว่า คุณสมบัติ bookDetail มีค่าเริ่มต้นของ nil เช่นเดียวกับตัวเลือกอื่น ๆ แต่สามารถเข้าถึงได้โดยไม่จำเป็นต้องแกะค่าของมัน ตามที่อธิบายไว้ใน Implicitly Unwrapped Optional

เนื่องจากbookDetail มีค่าเริ่มต้นเป็น nil ดังนั้น myBook instance ใหม่ จะถือว่า initialized ทั้งหมดเป็น myBook instance โดยจะตั้งค่าคุณสมบัติ name ภายใน initialize ของมันเอง ซึ่ง

หมายความ ว่า myBook initializer สามารถเริ่มต้นการอ้างอิงและผ่านคุณสมบัติต่างๆได้ด้วยตัวเองทันทีที่มีการตั้งค่าคุณสมบัติ name

myBook initializer จึงสามารถส่งผ่านตัวเองเป็นหนึ่งในพารามิเตอร์สำหรับการเริ่มต้น Description initializer เมื่อ myBook initializer กำลังจัดการตั้งค่าสถานที่ให้บริการ bookDetail ของตัวเอง

ทั้งหมดนี้หมายความ ว่า คุณสามารถสร้าง instances ของ myBook และ Description ในครั้งเดียว ไม่มีการสร้างวงจรอ้างอิงแบบ strong และคุณสมบัติ bookDetail สามารถเข้าถึงได้โดยตรง ไม่จำเป็นต้องใช้

เครื่องหมายอัศเจรีย์เพื่อไม่ตัดค่าของตัวเลือก :

```
var book = myBook(name: "Twilight", writer: "Stephenie Meyer")
print("\(book.name) is written by \(book.bookDetail.name)")
// prints " Twilight is written by Stephenie Meyer”
```

initialization เสร็จสมบูรณ์ ขณะที่ยังคงหลีกเลี่ยงวงจรอ้างอิงแบบ strong

**Strong Reference Cycles for Closures**

คุณเห็นข้างบนว่าวงจรอ้างอิงแบบ strong สามารถสร้างได้เมื่อคุณสมบัติของอิน instance ของคลาส 2 คลาสเก็บการอ้างอิงแบบ strong ซึ่งกันและกัน นอกจากนี้คุณยังเห็นวิธีการอ้างอิงแบบ weak และแบบ unowned ที่จะหยุดวงจรการอ้างอิงแบบ strong เหล่านี้

รอบอ้างอิงแบบ strong อาจเกิดขึ้นหากคุณกำหนดการปิดคุณสมบัติของ class instance และรวบรวมเนื้อความของการปิด instance ภาพนี้อาจเกิดขึ้น เพราะ body ของการเข้าถึงคุณสมบัติของอินสแตนซ์ เช่น self.someProperty หรือการปิดการเรียกวิธีการบน instance เช่น self.someMethod() ในกรณีอย่างใดอย่างหนึ่ง เหล่านี้ทำให้การปิดการ "capture " ตัวเอง สร้างวงจรอ้างอิงแบบ strong

วงจรอ้างอิงแบบ strong นี้เกิดขึ้นเนื่องจากการปิด เหมือน class มีชนิดการอ้างอิง เมื่อคุณกำหนดการปิดคุณสมบัติ คุณกำลังกำหนดการอ้างอิงไปที่ที่ปิด ในสาระสำคัญมันเป็นปัญหาเดียวกันกับข้างต้น โดยการอ้างอิงทั้งสองแบบมีให้เห็นกันในชีวิตประจำวัน อย่างไรก็ตามมากกว่าสองคลาสของ instances ในเวลานี้จะมี instances ของคลาสและการปิดที่จะทำให้อันอื่น ๆ ยังทำงานอยู่

Swift ให้วิธีแก้ปัญหานี้ เป็นการปิด capture อย่างไรก็ตาม ก่อนที่คุณเรียนรู้วิธีการตัดวงจรอ้างอิงแบบ strong กับการปิด capture ได้ประโยชน์ที่จะเข้าใจว่าอาจเกิดวงจรดังกล่าว ตัวอย่างด้านล่างแสดงว่าคุณสามารถสร้างการวงจรอ้างอิงแบบ strong เมื่อใช้การปิดที่อ้างอิงตัวเอง ตัวอย่างนี้กำหนดคลาสที่เรียกว่า HTMLElement ซึ่งเป็นตัวอย่างสำหรับแต่ละองค์ประกอบภายในเอกสาร HTML [4-5]

```
class HTMLElement {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) is being deinitialized")
```

คลาส HTMLElement กำหนดคุณสมบัติ name ซึ่งบ่งชี้ชื่อขององค์ประกอบ เช่น "p" สำหรับเป็นองค์ประกอบย่อหน้า หรือ "br" สำหรับองค์ประกอบการเป็นเว้นบรรทัด HTMLElement กำหนดลักษณะของ text ที่เลือกได้ ซึ่งคุณสามารถตั้งค่า string ที่แสดงถึงข้อความจะแสดงองค์ประกอบภายในของ HTML นั้น

นอกจากเรื่องสองตัวอย่างคุณสมบัติเหล่านี้ คลาสHTMLElement ยังกำหนดคุณสมบัติ lazy ที่เรียกว่า asHTML ซึ่งคุณสมบัตินี่จะอ้างอิงการปิดที่รวม name และ text ในส่วนของ string เป็น HTML ด้วย คุณสมบัติ ของ asHTML เป็นชนิด () -> String หรือ "ฟังก์ชันที่ไม่ใช้พารามิเตอร์ และส่งกลับค่า String เป็นผลลัพธ์"

โดยค่าเริ่มต้นของ คุณสมบัติ asHTML ถูกกำหนดให้ปิดการส่งกลับค่า String การแสดงของแท็ก HTML แท็กนี้ประกอบด้วยค่า text ในตัวเลือกมีอยู่ หรือไม่มี text ถ้า text มีอยู่สำหรับองค์ประกอบย่อหน้า การปิดจะกลับมา " some text " หรือ " " ขึ้นอยู่กับว่าคุณสมบัติ text เท่ากับ "some text" หรือ nil นั้น

คุณสมบัติ asHTML เป็นชื่อ และใช้เหมือน instance method อย่างไรก็ตาม เนื่องจาก asHTML มีคุณสมบัติการปิดมากกว่า instance method คุณสามารถแทนค่าเริ่มต้นของคุณสมบัติ asHTML กับการปิดเอง ถ้าคุณต้องการเปลี่ยนการแสดงผล HTML สำหรับองค์ประกอบ HTML ที่เฉพาะ

*NOTE*

*ประกาศคุณสมบัติ asHTML เป็นคุณสมบัติ lazy เพราะเท่านั้นการองค์ประกอบจริงต้องการจะแสดงเป็นค่า string สำหรับเป้าหมายของบางผลลัพธ์ของ HTML ความจริงว่า asHTML หมายถึงคุณสมบัติ lazy ที่คุณสามารถอ้างอิงถึง self ในการเริ่มต้นการปิด เนื่องจากคุณสมบัติ lazy จะไม่สามารถเข้าถึงจนกว่าจะเสร็จสิ้นการเตรียมใช้งาน และ self รู้ว่ามีอยู่*

คลาส HTMLElement ให้เป็นตัวเดียว ซึ่งใช้อาร์กิวเมนต์ name และ (ถ้าต้องการ) อาร์กิวเมนต์ text เริ่มต้นองค์ประกอบใหม่ คลาส deinitializer ซึ่งพิมพ์ข้อความที่แสดงเมื่อ HTMLElement instance คือ deallocated

นี่คือวิธีที่คุณใช้คลาส HTMLElement เพื่อสร้าง และพิมพ์อินสแตนซ์ใหม่:

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// prints "<p>hello, world</p>"
```

*NOTE*

*ตัวแปรย่อหน้าข้างต้นถูกกำหนดเป็น HTMLElement ดังนั้นมันสามารถกำหนดไปด้านล่าง nil นั้นเพื่อแสดงให้เห็นถึงสถานะของรอบอ้างอิงแบบ strong*

แต่เสียตรงที่คลาส HTMLElement ตามที่เขียนไว้ข้างต้น สร้างวงจรอ้างอิงแบบ strong ระหว่าง อินสแตนซ์ HTMLElement และการปิดใช้ค่าเริ่มต้น asHTML นี่คือลักษณะวงจร

อินสแตนซ์ HTMLElement และการปิดใช้ค่าเริ่มต้น asHTML นี่คือลักษณะวงจร

คุณสมบัติของอินสแตนซ์ asHTML คือการปิดการอ้างอิงแบบ strong อย่างไรก็ตาม เนื่องจากการปิดหมายถึง self ภายใน body (เป็นวิธีการอ้างอิง self.name และ self.text), การปิดจับภาพ self ซึ่งหมายความ ว่า มันมีการอ้างอิงแบบ strong กลับไปยังอินสแตนซ์ HTMLElement มีการสร้างวงจรอ้างอิงแบบ strong ระหว่างทั้งสอง(สำหรับข้อมูลเพิ่มเติมเกี่ยวกับค่าการปิดจับภาพ ดูที่ Capturing Values

*NOTE*

*แม้ว่าปิดอ้างอิงถึง self หลายครั้ง มันเท่ากับการอ้างอิงแบบ strong กับอินสแตนซ์ HTMLElement นั่นเอง*

ถ้าคุณตั้งค่า paragraph ตัวแปร nil และหยุดอ้างอิงแบบ strong กับ อินสแตนซ์ HTMLElement จะไม่มีอินสแตนซ์ HTMLElement หรือการปิดมันจะทำให้ deallocated เนื่องจากวงจรอ้างอิงแบบ strong :

```
paragraph = nil
```

โปรดสังเกตว่า ข้อความใน HTMLElement deinitializer จะไม่ prin แสดงว่า อินสแตนซ์ HTMLElement จะไม่ deallocated

**Resolving Strong Reference Cycles for Closures**

คุณแก้ไขวงจรอ้างอิงแบบ strong ระหว่างการปิด กับ คลาสอินสแตนซ์ โดยกำหนด capture ให้เป็นส่วนหนึ่งของคำจำกัดความของการปิด capture กำหนดกฎเพื่อใช้ในการ capture อย่างน้อยหนึ่งชนิดหรือมากกว่าการอ้างอิงภายในเนื้อหาของการปิด เช่นเดียวกับวงจรอ้างอิงแบบ strong ระหว่างสองคลาสอินสแตนซ์ คุณcaptured reference ต้องการอ้างอิงแบบ weak หรือ unowned แทนการอ้างอิงแบบ strong ทางเลือกที่เหมาะสมของ weak หรือ unowned ขึ้นอยู่กับความสัมพันธ์ระหว่างส่วนต่าง ๆ ของโค้ดของคุณ

> *NOTE*
>
> *Swift ต้องเขียน self.someProperty หรือ self.someMethod (มากกว่าเพียงแค่ someProperty หรือ someMethod) เมื่อใดก็ตามที่คุณอ้างถึงสมาชิกของ self ภายในการปิดนี้ช่วยให้คุณจำไว้ว่า จะสามารถ capture ด้วย self*

แต่ละ item ใน capture เป็นการจับคู่ของ weak หรือ unowned ที่เป็น keyword มีการอ้างอิงอินสแตนซ์ของคลาส (เช่น self or หรือ someInstance) Pairings เหล่านี้จะถูกเขียนในคู่ของวงเล็บสี่เหลี่ยม คั่นด้วยเครื่องหมายจุลภาค

ทำรายการ capture ก่อนรายการพารามิเตอร์ของการปิด และส่งคืน type ถ้ามีอยู่ :

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!] (index: Int, stringToProcess: String) -> String
in
    // closure body goes here
}
```

```
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```

**Weak and Unowned References**

กำหนดการ capture ในแบบปิดเป็นการอ้างอิงแบบ unowned เมื่อการปิดอินสแตนซ์ที่จะ capture จะเหมือนกัน และจะถูก deallocated ในเวลาเดียวกัน

ในทางกลับกัน การกำหนดการ capture โดยอ้างอิงแบบ weak เมื่ออ้างอิง capture อาจเป็น nil บางจุดในอนาคต การอ้างอิงที่แบบ weak มักเลือกชนิด และกลายเป็น nil โดยอัตโนมัติเมื่อเกิดการ deallocated อินสแตนซ์ที่จะอ้างอิง ซึ่งจะช่วยให้คุณสามารถตรวจสอบการมีที่อยู่ภายใน body แบบปิด

*NOTE*
*ถ้าการอ้างอิง captured ไม่เคยเป็น nil มันควรจะบันทึกเป็นการอ้างอิงแบบ unowned แทน การอ้างอิงแบบ weak*

การอ้างอิงแบบ unowned เป็นวิธี captured ที่เหมาะสมที่จะใช้แก้ไขวงจรอ้างอิงแบบ stong ในตัวอย่าง HTMLElement จากก่อนหน้านี้ นี่คือวิธีที่คุณเขียนคลาส HTMLElement เพื่อหลีกเลี่ยงวงจรอ้างอิงแบบ stong

```
class HTMLElement {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
```

การดำเนินงานของ HtmlElement นี้ เป็นเหมือนการดำเนินการก่อนหน้านี้ นอกเหนือจากการเพิ่มขึ้นของ capture รายการที่อยู่ในการปิดของ asHTML แล้ว ในกรณีนี้ รายการ capture คือ [unowned self] ซึ่งหมายความว่า " capture self เป็นการอ้างอิงแบบ unowned แทนที่เป็นการอ้างอิงแบบ strong"

คุณสามารถสร้าง และ print อินสแตนซ์ HTMLElement ดังนี้ :

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

นี่คือการอ้างอิงลักษณะการ capture ในสถานที่

เวลานี้ การ capture ของ self โดยการปิดการอ้างอิงแบบ unowned และไม่เก็บ strong ไว้บนอินสแตนซ์ HTMLElement โดยการ captured ถ้าคุณตั้งค่าการอ้างอิงแบบ strong จาก paragraph ตัวแปร nil และ อินสแตนซ์ HTMLElement คือ deallocated สามารถเห็นได้จากการพิมพ์ข้อความของ deinitializer ในตัวอย่างด้านล่าง:

```
paragraph = nil
// prints "p is being deinitialized"
```

## Optional Chaining

คือกระบวนการสำหรับการ query การเรียก properties, methods, และ subscript แบบ optional โดยที่ optional สามารถเป็น nil ได้. ถ้าเกิดว่า optional มีค่าอยู่ properties, method หรือ subscript จะถูกเรียกได้อย่างสำเร็จ แต่ถ้าหาก optional ถูกกำหนดว่าเป็น nil จะได้รับการ return กลับมาเป็น nil.

โดยที่สามารถทำการ query เป็นแบบลูกโซ่ได้(chain) และการ chain จะจบลงแบบผิดพลาด เมื่อมี link ในนั้นเป็น nil

### Optional Chaining as an Alternative to Forced Unwrapping

เราสามารถระบุ optional chaining ได้ โดยการใส่ ? ข้างหลังค่าของ optional ที่เราต้องการเรียกใช้ properties , methods หรือ subscripts ถ้าค่าของ optional ไม่เป็น nil. การทำแบบนี้จะคล้ายกับการทำ force unwrapping กับค่าของ optional โดยการใส่ ! หลังค่า.โดยคามแตกต่างคือ chaining จะจบการทำงานแบบไม่มี error เมื่อค่าของ optional เป็น nil แต่ force จะทำให้เกิด runtime error เมื่อค่าของ optional เป็น nil

```
class Person {
    var residence: Residence?
}
class Residence {
    var numberOfRooms = 1
}
```

จากตัวอย่าง เมื่อมีการสร้าง instance ใหม่ของ Person จะพบว่า residence จะมีค่าโดย default เป็น nil จากการที่เป็น optional

```
let john = Person()
```

เมื่อสร้าง john พบว่า john จะมี residence เป็น nil

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

ถ้ามีการเข้าถึงแบบ force unwrap พบว่าจะเกิด runtime error เนื่องจากไม่มีค่า residence. แต่ code นี้จำทำงานได้สำเร็จหากว่าค่าของ john.residence ไม่เป็น nil

แต่เราสามารถเข้าถึงค่าของ residence โดยไม่เกิด runtime error เมื่อมีค่าเป็น nil ได้อีกวิธีคือการเข้าผ่านแบบ optional โดยเปลี่ยนจาก ! เป็น ?

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// prints "Unable to retrieve the number of rooms."
```

จะพบว่าหาก residence มีค่าก็จะได้เป็น Int ออกมา แต่หากว่าไม่มีค่าก็จะได้ nil ออกมาโดยไม่เกิด runtime error

**Defining Model Classes for Optional Chaining**

โดยปกติเราวามรถใช้การเข้าถึงแบบ optional ได้มากกว่า 1 ขั้นลงไป และใช้เช็คว่าค่านั้น method นั้น หรือ properties นั้นๆเข้าถึงได้หรือไม่

```
class Person {
    var residence: Residence?
}
class Residence {
    var rooms = Room[]()
    var numberOfRooms: Int {
    return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
```

ในรอบนี้ Residence จะเพิ่ม empty array ที่มี type เป็น room เข้าไป

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
    } else {
        return nil
    }
```

จากตัวอย่าง code พบว่าเราสามารถเข้าถึงตัวแปร method ต่างๆ โดยใช้ optional เป็นตัวช่วยได้ โดยมีการเข้าถึงผ่านหลายระดับ ถ้าเกิดว่ามีจุดไหนจุดหนึ่งที่หาค่าไม่ได้ หรือไม่ได้กำหนดค่าก็จะได้ nil ออกมา

**Accessing Properties Through Optional Chaining**

จากตัวอย่าง Optional Chaining as an Alternative to Forced Unwrapping เราพบว่าสามารถเข้าถึงแบบ optional แทนการเข้าถึงแบบ force unwrap ได้โดยใช้ ? เพื่อดูการกำหนดค่าได้ และเข้าถึงได้โดยไม่เกิด error

จากตัวอย่าง class ด้านบน ถ้าเราใช้สร้าง instance ของ Person และพยายามที่จะเข้าถึง properties ของ numberOfRooms ของมัน ดังนี้

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// prints "Unable to retrieve the number of rooms."
```

เนื่องจาก john.residence เท่ากับ nil การใช้การเข้าถึงแบบ optional ก็จะจบลงที่การไม่สำเร็จ เช่นเดิม ถ้าเราพยายามจะเข้าถึงเพื่อ set ค่าผ่าน optional แบบนี้

```
let someAddress = Address ( )
someAddress.buildingNumber  =  "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

เราก็จะไม่สามารถกำหนดได้ เนื่องจาก john.residence เป็น nil

**Calling Methods Through Optional Chaining**

เราสามารถใช้การเรียก method แบบ optional เพื่อเช็คการเข้าถึงได้ ถึงแม้ว่า method นั้นจะไม่ได้ประกาศ return value ก็ตาม

```
func printNumberOfRooms() {
    print("The number of rooms is \(numberOfRooms)")
}
```

ถึงแม้ไม่ได้มีการกำหนดการ return ค่า แต่ตาม default แล้ว จะมีการ return เป็น void แต่ถ้าเราตรวจสอบว่า method ใช้ได้หรือไม่โดยการใช้ optional เราจะไม่ได้รับค่า void ในกรณีที่ไม่ได้มีการประกาศ return ไว้ แต่จะได้รับเป็น nil แทน จึงเป็นเหตุให้เราสามารถทำแบบนี้ได้ เช่น

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

**Accessing Subscripts Through Optional Chaining**

เราสามารถใช้ optional chaining ในการดึงค่าและเซตค่าจาก subscripts โดยผ่าน optional value ได้. และยังสามารถเช็คได้อีกว่า subscript นั้นสามารถเรียกใช้งานได้สำเร็จหรือไม่

ตัวอย่างต่อไปนี้เป็นการดึงค่า ชื่อของห้องแรกจาก array room ของ john.residence

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

```
john.redence?[0] = Room (name : "Bathroom")
```

เช่นเดียวกันกับการใช้คำสั่งนี้ จะมีการ fail เหมือนกัน

ถ้าเราสร้าง หรือ เซทค่าให้กับ Residence instance ซึ่งคือ john.resedence. โดยที่สร้าง element 1 หรือมากกว่าใน room เราจะสามารถใช้ optional chaining เข้าถึงค่าใน room array ได้ ดังนี้

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "The first room name is Living Room."
```

**Linking Multiple Levels of Chaining**

เราสามารถ link การ chain optional หลายๆระดับเข้าด้วยกันได้ แต่อย่างไรก็ตามการทำ multiple optional chaining ไม่ได้กำหนด return value ของ level อื่นๆให้เป็น optional ไปด้วย ยกเว้นเราจะได้ optional จาก level แรกที่เราเรียกไป

- *ถ้าเราต้องการดึงค่าแบบไม่เป็น optional ผ่าน optional chaining เราจะทำไม่ได้ และได้ค่าที่เป็น optional*
- *ถ้าเราต้องการดึงค่าที่เป็น optional และมันเป็นอยู่แล้วเราก็จะได้ค่าที่เป็น optional*

*ดังนั้น*

- *ถ้าต้องงการ Int ผ่านการเรียกผ่าน optional chaining เราก็จะได้ Int?*
- *ถ้าเราต้องการ Int? ผ่าน optional Chaining เราก็จะได้ Int?*

**Chaining on Methods with Optional Return Values**

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")
    } else {
        print("John's building identifier does not begin with \"The\".")
    }
}
// Prints "John's building identifier begins with "The"."
```

เราจะพบว่า buildingIdentifier() method ก็จะได้ค่าเป็น String? ออกมาดังที่ได้กล่าวไว้หัวข้อด้านบน

## Type Casting

ใช้เช็ค type ของตัวแปร หรือทำให้ตัวแปรเปลี่ยนไป ทำ type casting โดยใช้ is, as operator

### Defining a Class Hierarchy for Type Casting

สามารถใช้ type casting กับ class ที่เป็นลำดับชั้นและ subclass เพื่อเช็ค type และเพื่อ cast ตัวแปรให้เป็น class อื่นๆในลำดับชั้นเดียวกัน

Code 3 ส่วนข้างล่างเป็นการประกาศ hierarchy of classes and an array ที่เก็บค่าตัวแปรของ class ดังกล่าวเพื่อใช้ในการยกตัวอย่าง type casting

```
class MediaItem {
        var name: String
  init(name: String) {
        self.name = name
        }
}
```

Class แรก MediaItem

- *ตัวแปร name เป็นตัวแปรเก็บค่า string*
- *init construcction*

ใน code ส่วนที่ 2 จะเป็น subclasses of MediaItem.

Subclass แรกเป็น Movie เก็บข้อมูลเกี่ยวกับภาพยนตร์หรือ film ใน class นี้ จะมี director เพิ่มเข้ามา ในส่วนของ constructor จะทำการรับ 2 ค่า คือ name, director

```
class Movie: MediaItem {
   var director: String
   init(name: String, director: String) {
      self.director = director
      super.init(name: name) }
```

Director จะทำการกำหนดภายใน class movie,ใช้ self ส่วน name จะเรียก constructor ของ class Media โดยใช้ super

```
class Song: MediaItem {
  var artist: String
   init(name: String, artist: String) {
      self.artist = artist
      super.init(name: name)
   }
}
```

Class song เป็น subclass ของ Media เช่นกัน โดยมีตัวแปร artist เพิ่มเข้ามา

- Final snippet สร้างตัวแปร Array ชื่อ library ที่ทำหน้าที่เก็บหนัง 2 เรื่อง และเพลง 3 เพลง

```
let library = [
   Movie(name: "Casablanca", director: "Michael Curtiz"),
   Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
   Movie(name: "Citizen Kane", director: "Orson Welles"),
   Song(name: "The One And Only", artist: "Chesney Hawkes"),
   Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
```

อย่างไรก็ตามเวลา library คืนค่า Movie และ Song กลับค่าดังกล่าวจะไม่ได้อยู่ใน class Movie หรือ Song แต่เป็น class MedisItem ดังนั้นจึงจำเป็นต้อง Check type ก่อนนำไปใช้

**Checking Type**

จะใช้ operator (is) เพื่อ check ว่าตัวแปรนั้นเป็น type ของ subclass หรือไม่

- *reture true ถ้าใช่*
- *return false ถ้าไม่ใช่*

ในตัวอย่างทำการประกาศตัวแปร movieCount , songCount ที่ทำหน้าที่นับจำนวน Movie และ Song ที่ถูกเก็บใน library array

```
var movieCount = 0
var songCount = 0
for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
```

จากโค้ดจะทำการนับ item ที่เก็บใน library ทีละ item ถ้า item อยู่ใน subclass Movie ก็จะ return true ถ้าไม่จะ return false

สำหรับ item is Song ก็เช่นกัน

ถ้าผ่านเงื่อนไขดังกล่าว movieCount / songCount จะถูกเพิ่มค่า ทำให้สามารถระบุได้ว่าใน mediaItem มีจำนวนของแต่ละ item เท่าไร

**Downcasting**

เราสามารถ downcast เพื่อให้เป็น subclass ได้ โดยใช้ operator (as)

เพราะ downcasting สามารถทำให้เกิดปัญหาได้ type cast operator จึงมี 2 รูปแบบ

- *as? Return ค่าความจริง ว่ามารถ downcast ได้หรือไม่*
- *as บังคับให้ downcast เลย*

ใช้ as? เมื่อไม่มั่นใจว่า downcast จะสำเร็จ หากไม่สามารถ downcast จะ return nil

ใช้ as เมื่อมั่นใจว่า downcast แล้วสำเร็จแน่นอน ถ้าไม่สามารถ downcast ได้ จะเกิด error

ในตัวอย่างนี้ แต่ละ item ที่ถูกเก็บใน array อาจจะได้เป็น Movie หรือ Song ก็ได้ แต่เราไม่มีทางรู้เลยว่าเป็น class ไหนกันแน่ เราจึงควรใช้ as? เพื่อเช็คก่อนว่า downcast ได้หรือไม่

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}
```

Downcasting เป็น movie จะ fails เมื่อพยายามจะ downcast item ที่จริงๆแล้วเป็น Song การรับมือกับปัญหาที่จะเกิดขึ้นนี้โดยใช้ as?

## Type Casting for Any and AnyObject

- AnyObject : สามารถแทนได้ทุก class type.
- Any : สามารถแทนได้ทุก type นอกเหนือจาก function types

## AnyObject

```
let someObjects: [AnyObject] = [
    Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
    Movie(name: "Moon", director: "Duncan Jones"),
    Movie(name: "Alien", director: "Ridley Scott")  ]
```

ตัวอย่างนี้เป็นการประกาศ array type [AnyObject] และใน array มี 3 Instances ของ Movie

เพราะ array นี้เป็นที่รู้ดีว่าเก็บค่าเฉพาะ Movie Instance คุณสามารถที่จะ downcast และ unwrap ได้โดยตรง

```
for object in someObjects {
    let movie = object as Movie
    print("Movie: \(movie.name), dir. \(movie.director)")
}
```

สำหรับอีกรูปแบบหนึ่งของ loop ดังกล่าว ทำได้โดย downcast some object array เป็น Movie เลย

```
for movie in someObjects as Movie[] {
    print("Movie: \(movie.name), dir. \(movie.director)")
}
```

**Any**

นี่เป็นตัวอย่างของการใช้งาน any กับ type ที่แตกต่างกัน

จากตัวอย่างสร้าง array ที่ things ที่เก็บค่าของทุก type

```
var things = [Any]()
things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

ใน things array ประกอบด้วย Int 2 ค่า, Double 2 ค่า ,String, และ tuple (Double,Double) และ movie ชื่อ “Ghostbusters” ที่กำกับโดย Ivan Reitman

คุณสามารถใช้ is , as ร่วมกับ switch ได้ด้วย โดยแต่ละ case จะถูกผูกไว้กับค่าที่เหมาะสมกับเงื่อนไข

```
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
```

## Nested Types

Enumerators มักถูกสร้างขึ้นเพื่อช่วยใน function ของ class ต่างๆ ในทำนองเดียวกัน Enum ก็สะดวกในการกำหนด utility class และ โครงสร้างของ type ที่ซับซ้อนได้ง่ายขึ้น เพื่อบรรลุวัตถุประสงค์ดังกล่าว Swift ให้คุณสามารถกำหนด nested type ด้วยวิธีการสร้าง supporting enumeration, class และ structure ภายใต้การกำหนด type support ต่างๆ

### Nested Types in Action

ตัวอย่างด้านล่างเป็นการกำหนด struct ชื่อ BlackjackCard ที่ออกแบบการเล่นไพ่ในเกมส์ โดย BlackjackCard structure ประกอบด้วย 2 nested enumeration ชื่อ Suit และ Rank ซึ่งใน Blackjack ไพ่ Ace มีค่าได้ทั้ง 1 หรือ 11 โดยฟีเจอร์นี้ถูกแทนด้วย structure ที่ชื่อ values อยู่ภายใน Rank enumeration

```
struct BlackjackCard {
   // nested Suit enumeration
   enum Suit: Character {
      case Spades = "♠", Hearts = "♡", Diamonds = "♢", Clubs = "♣"
   }
   // nested Rank enumeration
   enum Rank: Int {
   case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
   case Jack, Queen, King, Ace
   struct Values {
      let first: Int, second: Int?
   }
   var values: Values {
   switch self {
   case .Ace:
      return Values(first: 1, second: 11)
   case .Jack, .Queen, .King:
     return Values(first: 10, second: nil)
```

```
    default:
        return Values(first: self.rawValue, second: nil)
        }
    }
// BlackjackCard properties and methods
 let rank: Rank, suit: Suit
var description: String {
var output = "suit is \(suit.rawValue),"
    output += " value is \(rank.values.first)"
    if let second = rank.values.second {
        output += " or \(second)"
    }
    return output
```

Suit eumeration คือรูปแบบของการ์ดในสำรับ ต่อมา Rank enumeration คือลำดับของการ์ด 13 ใบในสำรับ มีการกำหนด nested structure ภายในโครงสร้างของตัวเองที่ชื่อว่า values, structure เป็นตัวช่วยกำหนดว่าการ์ดแต่ละใบจะมีเพียงค่าเดียวเท่านั้น มีแค่เฉพาะ Ace ที่มีได้ 2 ค่า นั่นคือ 1 และ 11 โดย values structure กำหนด 2 คุณสมบัติดังนี้ first เป็น Int และ second ที่เป็น Int? หรือ optional Int ในส่วนของ Rank ทำหน้าที่กำหนดการคำนวณค่าที่จะ return ค่าของ values structure การคำนวณคุณสมบัตินี้จะถูกพิจารณาตามลำดับของการ์ด และจะมีค่าพิเศษให้กับ Jack, Queen, King ส่วนของ BlackjackCard structure จะประกอบด้วย 2 คุณสมบัติคือ rank และ suit นอกจากนี้ยังมีคุณสมบัติที่ชื่อ description ที่ใช้เพื่ออธิบายชื่อและค่าของการ์ดจาก Rank และ Suit เพราะ BlackjackCard เป็น structure ที่ไม่มี custom initializer มันจึงมี memberwise initializer โดยปริยาย

เราสามารถใช้ initializer นี้เพื่อกำหนดค่าให้กับตัวแปรในตอนเริ่มต้นชื่อ TheAceOfSpades

```
let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
print("theAceOfSpades: \(theAceOfSpades.description)")
// prints "theAceOfSpades: suit is ♠, value is 1 or 11"
```

ถึงแม้ว่า Rank และ Suit จะเป็น nested ภายใน BlackjackCard แต่ type ของทั้ง Rank และ Suit สามารถอ้างอิงจาก Context ได้ ดังนั้น การกำหนดค่าเริ่มต้นของตัวแปรก็จะสามารถอ้างถึงสมาชิกของ enumeration ด้วยชื่อของสมาชิก (Ace และ Spades ในตัวอย่างข้างต้น โดยคุณสมบัติ description จะแสดงค่าของ Ace of Spades ซึ่งมีค่าได้ทั้ง 1 และ 11)

## Extensions

ส่วนขยายเพิ่มฟังก์ชันใหม่ที่มีอยู่ในระดับ โครงสร้างหรือ enumeration type ซึ่งรวมถึงความสามารถในการขยายประเภทที่คุณไม่ได้มีการเข้าถึงรหัสแหล่งที่มาต้นฉบับ ( ที่รู้จักกันเป็นแบบย้อนหลัง ) ส่วนขยายจะคล้ายกับประเภท objective-c. (ต่างจาก Objective_C, Swift ส่วนขยายไม่มีชื่อ)

ส่วนขยายใน Swift:

- เพิ่มคุณสมบัติในการคำนวณและคำนวณคุณสมบัติคงที่
- กำหนด instance method และ ชนิดของ method
- จัดเตรียม initializers ใหม่
- กำหนด subscripts
- กำหนด และใช้ nested types ใหม่
- ทำให้เป็นชนิดที่สอดคล้องกับ โปรโตคอล

> *Note*
> *ถ้าคุณกำหนดส่วนขยายเพื่อเพิ่มฟังก์ชันใหม่เป็นชนิดที่มีอยู่ ฟังก์ชันใหม่จะสามารถใช้บนอินสแตนซ์ที่มีอยู่ทั้งหมดของชนิด แม้ว่าพวกเขาสร้างขึ้นก่อนส่วนขยายถูกกำหนด*

**Extension Syntax**

การประกาศส่วนด้วยคีย์เวิร์ด extension

```
extension SomeType {
                    // new functionality to add to SomeType goes here
                  }
```

ส่วนขยายสามารถขยายประเภทที่มีอยู่ให้ใช้หนึ่งหรือหลายโปรโตคอล ในกรณีนี้ ชื่อของโพรโทคอลจะเขียนในแบบเดียวกับชั้นหรือโครงสร้าง

```
extension SomeType: SomeProtocol, AnotherProtocol {
        // implementation of protocol requirements goes here
    }
```

การเพิ่ม โปรโตคอลถูกอธิบายไว้ใน Adding Protocol Conformance with an Extension.

## Computed Properties

ส่วนขยายที่สามารถเพิ่มคุณสมบัติเช่นคำนวณคุณสมบัติชนิดและประเภทที่มีอยู่ตัวอย่างนี้เพิ่มการคำนวณคุณสมบัติของอินสแตนซ์ของ Swift มี 2 ชนิด เพื่อให้การสนับสนุนพื้นฐานสำหรับการทำงานกับ distance units

```
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
("Three feet is \(threeFeet) meters")
//prints "Three feet is 0.914399970739201 meters"
```

คุณสมบัติของการคำนวณแสดงให้เห็นว่า Double value ควรจะถือเป็นหนึ่งหน่วยของความยาว ถึงแม้ว่าพวกเขาจะใช้คำนวณคุณสมบัติ เช่น ชื่อ คุณสมบัติเหล่านี้สามารถผนวกกับมูลค่าที่แท้จริงจุด - จุดไวยากรณ์เป็นวิธีที่จะใช้อักษร ค่าแสดงการแปลงระยะทาง

ตัวอย่างนี้ พิจารณา Double value ของ 1.0 ถูกพิจารณาเป็นตัวแทนของ “one meter” นี่คือเหตุผลที่ m คำนวณคุณสมบัติส่งกลับค่าตัวเอง — นิพจน์ 1.m ถูกพิจารณาการคำนวณค่าของ Double valure ของ 1.0

หน่วยอื่น ๆ ต้องใช้การแปลงบางอย่างเพื่อแสดงค่าที่วัดเป็นเมตร 1 กิโลเมตร คล้าย ๆ กับ 1000 เมตร ดังนั้น km การคำนวณคุณสมบัติคูณค่า โดย 1_000.00 แปลงเป็นตัวเลขที่แสดงในหน่วยเมตร ใน

ทำนองเดียวกัน มี ความยาว3.28024 ฟุต และเพื่อคำนวณ ft คำนวณคุณสมบัติพื้นฐานสองค่า โดยแบ่ง 3.28024 , เพื่อแปลงจากฟุตเป็นเมตร

คุณสมบัติเหล่านี้มีคุณสมบัติคำนวณแบบอ่านอย่างเดียว และดังนั้น พวกเขาจะแสดง โดยปราศจากคำสำคัญคือ get ค่าที่ส่งคืนจะเป็น Double และสามารถใช้ในการคำนวณทางคณิตศาสตร์:

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

*Note*

*ส่วนขยายสามารถเพิ่มคุณสมบัติใหม่ในการคำนวณ แต่พวกเขาไม่สามารถเก็บคุณสมบัติเก็บที่มีอยู่แล้ว*

**Initializers**

ส่วนต่อขยายที่สามารถเพิ่ม initializers ใหม่เป็นประเภทที่มีอยู่แล้ว ซึ่งจะช่วยให้คุณสามารถขยายส่วนอื่น ๆให้ยอมรับชนิดของการกำหนดเองเป็นพารามิเตอร์หรือเพื่อให้ตัวเลือกเริ่มต้นเพิ่มเติมที่ไม่ได้ถูกรวมเป็นส่วนหนึ่งของการดำเนินงานเดิมของประเภท

ส่วนขยายสามารถเพิ่ม initializers แต่ไม่สามารถเพิ่ม designated initializers หรือ deinitializers ในคลาส Designated initializers และ deinitializers จำเป็นต้องใช้คลาสเดิม

*Note*

*ถ้าคุณใช้ส่วนขยายเพื่อเพิ่ม initializer เป็นค่าเริ่มต้นสำหรับเก็บคุณสมบัติทั้งหมด และกำหนด initializers ใด ๆ เอง คุณสามารถเรียกค่าเริ่มต้นและค่า initializer memberwise initializer ที่พิมพ์จากภายใน initializer ของคุณ*

*นี้จะไม่เป็นกรณีถ้าคุณได้เขียน initializer เป็นส่วนหนึ่งของการดำเนินงานเดิมประเภทค่าที่ได้อธิบายไว้ใน Initializer Delegation for Value Types.*

ตัวอย่างด้านล่างกำหนดโครงสร้าง Rect เพื่อแสดงถึงสี่เหลี่ยมเรขาคณิต ตัวอย่างกำหนดสองโครงสร้างที่เรียกว่า Sizeและ Point ซึ่งทั้งสองให้ค่าเริ่มต้น 0.0 สำหรับคุณสมบัติ

```
struct Size {
var width = 0.0, height = 0.0
}
struct Point {
var x = 0.0, y = 0.0
}
struct Rect {
var origin = Point()
var size = Size()
```

เนื่องจากโครงสร้าง Rect กำหนดค่าเริ่มต้นสำหรับคุณสมบัติทั้งหมด ได้รับเป็นค่าเริ่มต้นของ และตัว Initializers และ memberwise Initializers โดยอัตโนมัติ ตามที่อธิบายไว้ใน Default Initializer. โดยInitializers เหล่านี้สามารถใช้เพื่อสร้างอินสแตนซ์ Rect ใหม่ได้

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
  size: Size(width: 5.0, height: 5.0))
```

คุณสามารถขยายโครงสร้างของ Rect เพื่อให้เป็นตัวเพิ่ม initialize ที่ใช้เฉพาะจุดศูนย์กลางและขนาด

```
extension Rect {
   init(center: Point, size: Size) {
      let originX = center.x - (size.width / 2)
      let originY = center.y - (size.height / 2)
      self.init(origin: Point(x: originX, y: originY), size: size)
   }
}
```

ตัว initializer ใหม่นี้เริ่มจากการคำนวณหาจุดกำเนิดที่เหมาะสมตามจุดศูนย์กลางและค่าขนาด เมื่อตัว initializer เรียกโครงสร้างนี้โดยอัตโนมัติ init(origin:size:), ที่เก็บค่าใหม่ของจุดเริ่มต้นและค่าใหม่ของขนาด ในที่ที่มีคุณสมบัติเหมาะสม

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

*Note*

*ถ้าคุณให้ initializer ใหม่กับส่วนขยาย, คุณจะยังคงรับผิดชอบในการตรวจสอบว่า แต่ละอินสแตนซ์ทั้งหมดได้เริ่มต้นเมื่อ initializer เสร็จสมบูรณ์*

ส่วนขยายสามารถเพิ่ม instance methods ใหม่ และ ชนิดของ Methods ตามชนิดที่มีอยู่ ตัวอย่างต่อไปนี้เป็นการเพิ่ม instance methods ใหม่โดยเรียกใช้ repetitions ตามชนิด Int

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}
```

Repetitions method (task:) ที่อาร์กิวเมนต์เดี่ยวชนิด () -> Void, ซึ่งบ่งชี้ว่า ฟังก์ชันไม่มีพารามิเตอร์ และไม่ return ค่า หลังจากกำหนดส่วนขยายนี้ คุณสามารถเรียก repetitions method บนจำนวนเต็มใดๆก็จะทำงานให้ตามจำนวนครั้ง

```
3.repetitions {
    print("Hello!")
```

```
    }
// Hello!
// Hello!
// Hello!
```

**Mutating Instance Methods**

การเพิ่ม Instance Methods ด้วยส่วนขยายโดยสามารถแก้ไข (หรือผ่าเหล่า) instance ของตัวเองได้ โครงสร้างและ enumeration methods ที่ปรับเปลี่ยนตัวเองหรือคุณสมบัติต้องทำเครื่องหมาย instance method เป็น mutating, จะเหมือนการ mutating method จากการใช้งานต้นฉบับ

ตัวอย่างด้านล่างเป็นการเพิ่ม mutating methods ใหม่ที่จะเรียก square ของ Swift ชนิด Int ที่สี่เหลี่ยมค่าเริ่มต้น

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt is now 9
```

ส่วนขยายที่สามารถเพิ่ม subscripts ใหม่ในชนิดที่มีอยู่ ตัวอย่างนี้เพิ่ม subscript เป็นจำนวนเต็มชนิด Int ใน Swift ตัว Subscript [n] นี้จะส่งตัวเลขทศนิยม n กลับจากตำแหน่งด้านขวาหมายเลข:

```
123456789[0]       return9
123456789[1]       return8
```

```
extension Int {
 subscript(digitIndex: Int) -> Int {
    var decimalBase = 1
       for _ in 1..<digitIndex {
          decimalBase *= 10
       }
       return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

ถ้าค่า Int มีหลักไม่เพียงพอสำหรับดัชนีร้องขอ ให้ใช้ subscript ส่ง 0 กลับไป เช่นถ้าหมายเลขมีการเบาะกับศูนย์ไปทางซ้าย

```
746381295[9]
// returns 0, as if you had requested:
746381295[9]
```

## Nested Types

ส่วนต่อขยายที่สามารถเพิ่ม Nested Types ใหม่ใน classes ที่มีอยู่, โครงสร้างและ enumerations :

```
extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
```

ตัวอย่างนี้เพิ่ม nested enumeration ใหม่ลงใน Int Enumerationนี้จะเรียก Kind แสดงชนิดของตัวเลขที่ใช้แสดงแทนจำนวนเต็มหนึ่ง ๆโดยเฉพาะ และแสดงว่าตัวเลขนี้เป็นค่าลบ, ศูนย์ หรือ ค่าบวก

ตัวอย่างนี้ยังเพิ่มคุณสมบัติใหม่ที่มีการคำนวณ Int เรียก kind, ซึ่งส่งกลับสมาชิกแจงการนับชนิดที่เหมาะสมสำหรับตัจำนวนเต็ม:

```
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
                print("+ ", terminator: "")
            }
        }
        print("")
    }
    printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
    // Prints "+ + - 0 - 0 + "
```

ฟังก์ชันนี้ printIntegerKinds(_:) ใช้ค่า Int ที่ป้อนข้อมูล และคำนวณซ้ำช่วงของค่านั้น สำหรับแต่ละจำนวน โดยพิจารณาคุณสมบัติจากการคำนวณสำหรับตัวเลข และพิมพ์คำอธิบายที่เหมาะสมของชนิดนั้น ฟังก์ชัน printIntegerKinds(_:) สามารถเรียกพิมพ์ชนิดของจำนวนเต็มในคำทั้งหมด การแสดงนี้สำหรับจำนวน *[3, 19, -27, 0, -6, 0, 7]*

> *NOTE*
>
> *number.kind ที่รู้จักเป็นชนิด Int.Kind ด้วยเหตุนี้ ทั้งหมดของค่าสมาชิก Int.Kind สามารถเขียนย่อแบบฟอร์มภายในสั่ง switch เช่น .negative มีมากกว่า Int.Kind.negative.*

## Protocols

คือแม่แบบสำหรับ methods , properties และความต้องการอื่นๆ เพื่อให้สามารถทำงานตามจุดประสงค์ที่วางไว้

### Protocol Syntax

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

สำหรับการเรียกใช้ protocol ก็ทำตามรูป

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

นั้นก็คือ ทำการเติมชื่อ protocol ที่ต้องการจะเรียกใช้ไว้ข้างหลัง ถ้ามีมากกว่า 1 protocol ก็ให้คั่นด้วย comma (การเรียกใช้แบบนี้อาจเรียกอีกอย่างว่า adopted protocol)

แต่ในกรณีที่เป็น class แล้วมี superclass ด้วย ต้องทำการใส่ superclass ก่อน protocol ดังนี้

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

**Property Requirements**

การประกาศ property เพียงแค่ทำการบอกว่าตัวแปรนั้นๆ สามารถทำอะไรได้บ้าง เช่น อ่านได้อย่างเดียว หรือ ทั้งอ่านทั้งเขียน โดยไม่ต้องกำหนดค่าให้ตัวแปร และต้องขึ้นด้วย var เท่านั้น ส่วนชนิดของตัวแปรจะเป็นอะไรก็ได้ เช่น Int, Double และอื่นๆ

*protocol SomeProtocol {*

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

ตัวอย่างการใช้งาน

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

จะเห็นว่า struct Person ได้มีการ adopted Fullynamed protocol เข้ามา จึงทำให้ใน struct จำเป็นที่ต้องมีตัวแปร fullName ที่เป็น String เหมือนกับทางฝั่ง protocol (เรียกว่าการ conform) ต่อมาเวลาสร้าง instance ของ struct Person จึงต้องใส่ค่า fullName ซึ่งเป็นค่าเริ่มต้นให้ เวลาทำการเรียกก็จะได้ค่านั้นออกมา แต่จะไม่สามารถแก้ไขค่าได้อีกต่อไปเพราะ ทางฝั่ง protocol กำหนดให้เป็น get อย่างเดียวเท่านั้น

**Method Requirements**

การประกาศ method ใน protocol จะมีรูปแบบเหมือนๆ กับประกาศฟังก์ชันปกติ คือนำด้วย func ต่อด้วยชื่อฟังก์ชัน (มีพารามิเตอร์ได้) ตามด้วยชนิดข้อมูลส่งกลับ ดังรูป

```
protocol SomeProtocol {
    class func someTypeMethod()
}
```

ซึ่งการที่ใครจะมา adopt และ conform protocol ใดๆ ก็จะต้องทำตามข้อตกลงที่ protocol นั้นได้วางไว้ ในที่นี้คือต้องมี func rand() -> Double ดังนั้นตัวอย่างการเรียกใช้ดูตามรูป

```
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"
```

จะเห็นได้ว่า class ข้างต้นได้มีการ adopted RandomNumberGenerator protocol และได้ทำการ conform เรียบร้อยแล้วโดยการสร้าง func random() -> Double ตามข้อกำหนด

**Mutating Method Requirements**

ในบางครั้งที่เราต้องการที่จะทำให้ protocol มีข้อกำหนดแบบ mutating func เพื่อให้มีประสิทธิภาพในการทำงานตามที่ต้องการ เราก็สามารถทำได้เพียงแค่เติมคำว่า mutating ไปข้างหน้าตามรูป

```
protocol Togglable {
    mutating func toggle()
}
```

เพียงเท่านี้ ใครก็ตามที่มา adopt protocol นี้จะต้องเขียน mutating func toggle() เท่านั้นจึงจะผ่าน

**Initializer Requirements**

เราสามารถกำหนด intializers ได้เช่นกัน เพื่อให้มีการกำหนดเริ่มต้นต่างๆ ที่จำเป็นต้องใช้สำหรับใครก็ตามที่มา adopt protocol นี้ ซึ่งสามารถเขียนได้ดังนี้

```
protocol SomeProtocol {
   init (someParameter : Int)
}
```

โดยใครก็ตามที่มา adopt จะต้องมี init(someParameter: Int) อยู่ในโครงสร้าง ซึ่งต้องเติมคำว่า required ไว้ข้างหน้าด้วย เป็นข้อบังคับ ตามรูป

```
Class SomeClass : SomeProtocol {
   Required init (someParameter : Int) {
            // initializer implementation goen here
   }
}
```

แต่ถ้ากรณีที่ subclass มาทำการ adopt protocol นี้ และยังได้มีการ override init อีกต่างจาก จะต้องทำการใส่ทั้ง required และ override ไว้ข้างหน้าด้วย

**Protocols as Types**

เราสามารถนำ protocol ต่างๆ ที่ได้สร้างขึ้นมาไปเป็น type ให้กับตัวแปรต่างๆ ได้ ซึ่งจะมีคุณสมบัติเหมือนกับ type ประเภทอื่นๆ เช่น เป็นพารามิเตอร์ ค่ารีเทิร์นของฟังก์ชัน ค่าคงที่ ค่าในอาเรย์ และอื่นๆ

ตัวอย่างการใช้ protocol เป็น type

```
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
    }
}
```

จะเห็นได้ว่า property generator มี type เป็น RandomNumberGenerator ซึ่งเป็น protocol จึงทำให้ตอนที่กำหนดค่าให้ property ตัวนี้จะต้องเป็นใครก็ตามที่ได้ทำการ adopt protocol นั้นๆ ดังรูป

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

การสร้าง instance d6 ได้มีการส่งค่าให้กับ generator คือ LinearCongruentialGenerator ซึ่งเป็น class ที่ได้ทำการ adopt RandomNumberGenerator ดังนั้นจึงเป็นไปตามข้อตกลง

**Delegation**

คือแนวทางการออกแบบที่อนุญาติให้ class หรือ struct สามารถมอบหมายหน้าที่บางอย่างให้กับ instance ของ type อื่นๆ ได้ ซึ่ง delegation จะกำหนด protocol เพื่อให้ instance อื่นๆ ทำงานแทน โดยไม่เปิดเผยขั้นตอนการทำงานใดๆ เหมือนๆ กับเราเป็นเจ้านายที่สามารถสั่งลูกน้องให้ไปทำงานแทนโดยที่เราเตรียมอุปกรณ์ต่างๆ ที่จำเป็นสำหรับการทำงานของลูกน้องคนนั้นไว้แล้ว

ตัวอย่าง

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: Int[]
    init() {
        board = Int[](count: finalSquare + 1, repeatedValue: 0)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var delegate: DiceGameDelegate?
func play() {
  square = 0
  delegate?.gameDidStart(self)
  gameLoop: while square != finalSquare {
    let diceRoll = dice.roll()
    delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
    switch square + diceRoll {
    case finalSquare:
      break gameLoop
    case let newSquare where newSquare > finalSquare:
      continue gameLoop
    default:
      square += diceRoll
      square += board[square]
    }
  }
  delegate?.gameDidEnd(self)
```

**Adding Protocol Conformance with an Extension**

เราสามารถเพิ่มโปรโตคอลให้กับใครก็ตามก็ได้ โดยที่ไม่จำเป็นต้องรู้ source code นั้นๆ เพื่อเพิ่ม function ตามที่ต้องการลงไป จากการใช้ extension เข้ามาช่วย ดังรูป

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

จะเห็นได้ว่า Dice ได้ทำการ extension เพิ่มเติมคือการ adopt TextRepresentable protocol เข้ามา แล้วก็ conform protocol นั้น ก็เป็นอันเสร็จสิ้น

```
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
print(d12. textualDescription)
// prints "A 12-sided dice"
```

ส่งผลให้ d12 ที่เป็น instance ของ Dice ก็สามารถเรียกใช้งาน textualDescription ได้แล้ว

**Declaring Protocol Adoption with an Extension**

ในกรณีที่ใครก็ตามที่จะมา adopt protocol นั้นๆ โดยที่ภายในนั้นได้มีการ conform ไว้เรียบร้อยแล้ว(กล่าวคือโค๊ดข้างในไม่ต้องทำการเพิ่มใดๆ ก็สามารถ adopt ได้ผ่านเลย) ก็จะมีการใช้ extension เข้ามาช่วย เพื่อให้ทำการ adopt protocol นั้นๆ เพิ่มเติมเพื่อความเป็นระเบียบเรียบร้อย ดังรูป

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// prints "A hamster named Simon"
```

จากรูปจะเห็นได้ว่า Hamster ได้ทำการ extension TexTRepresentable protocol แต่ไม่ได้ทำการเพิ่มอะไรเลย เพราะว่าตัว Hamster เองก็สมบูรณ์แบบอยู่แล้วตามที่ protocol นั้นต้องการ ส่วนรูปด้านล่างเป็นตัวอย่างการใช้งานว่า instance ของ Hamster สามารถใช้งานได้สำหรับ protocol นั้นๆ

**Collections of Protocol Types**

protocol ต่างๆ สามารถใช้เสมือนเป็น type ได้ ดังนั้นจึงสามารถนำไปเก็บเป็นค่าใน array หรือ dictionary ได้ ตามตัวอย่าง

```
let things:[ TextRepresentable] = [game, d12, simonTheHamster]
```

จะเห็นได้ว่าค่าใน array ของ things จำเป็นจะต้องเป็นค่าของใครก็ตามที่ต้อง adopt TextRepresentable protocol อยู่เท่านั้น

```
for thing in things {
    print(thing.textualDescription)
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

ตัวอย่างการใช้งานตามรูปด้านบนจะเห็นได้ว่า แต่ละค่าใน array สามารถใช้ func asText ได้ เพราะว่าต้อง adopt protocol นั้นๆ อยู่แล้ว

**Protocol Inheritance**

เราสามารถสืบทอด protocol หนึ่งๆ หรือมากกว่าหนึ่ง ไปสู่ protocol อื่นๆ ได้ และสามารถเพิ่มข้อกำหนดเพิ่มเติมได้

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

โดยมี syntax การเขียนเหมือนๆ กับเรื่อง class ตามรูปด้านบน

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

ซึ่งเราสามารถเพิ่มเติมข้อกำหนดต่างๆ เองได้เช่นกัน พร้อมๆ กับการได้รับสืบทอดมาจาก protocol แม่

**Class-Only Protocols**

เราสามารถจำกัดได้ว่า protocol ที่เราสร้างขึ้นนั้น ให้แค่ใครก็ตามที่ต้องเป็น class เท่านั้นในการ adopt protocol ของเรา เพียงแค่ทำการเพิ่มคำว่า class เข้าไป ตามรูป

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

แต่ถ้าเราต้องการที่จะได้รับการสืบทอดมาจาก protocol แม่ด้วย เราจำเป็นจะต้องใส่คำว่า class ไว้ตัวแรกก่อนเสมอ แล้วจึงตามด้วย protocol แม่ เพียงเท่านี้ก็จะมีแค่ class เท่านั้นที่จะ adopt protocol นี้ได้ struct และ enum จะไม่สามารถ adopt ได้

**Protocol Composition**

เราสามารถที่จะรับค่าที่ต้องการให้เป็นค่าที่มาจาก protocol 2 อันพร้อมๆ กันได้ (กล่าวคือค่าที่ได้รับต้องเป็นค่าที่มาจากใครก็ตามที่ adopt ทั้ง 2 protocol นั้น) ซึ่งมีวิธีการเขียนคือ protocol<1,2> โดย 1 และ 2 แทนชื่อ protocol ที่ต้องการ

*ไม่จำกัดแค่ 2 protocol สามารถทำหลายๆ protocol ก็ได้เช่นกัน*

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

จะเห็นได้ว่า func wishHappyBirthday มีการรับค่า โดยที่ค่านั้นต้องเป็นค่าที่มาจากใครก็ตามที่ adopt ทั้ง Named และ Aged protocol ซึ่งจากตัวอย่างด้านบนก็ได้แสดงวิธีการใช้งานด้วยเช่นกัน

**Checking for Protocol Conformance**

เราสามารถใช้ is และ as operator เพื่อตรวจสอบความสอดคล้องกับ protocol ได้

```
protocol HasArea {
    var area: Double { get }
}
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

สุดท้ายไม่ได้ ดังนั้นเมื่อมีการกำหนดค่า instance ของแต่ละ class ลงใน array ของ objects แล้วนำไปผ่านกระบวนการสุดท้ายคือการตรวจสอบว่า แต่ละ instance ใน array มีความสอดคล้องกับ protocol HasArea หรือไม่ ซึ่งจะพบว่าใช้ as? ในการตรวจสอบ และจะพบว่า instance ตัวสุดท้ายไม่สอดคล้อง จึง output ค่าออกมาไม่เหมือน instance ตัวอื่นๆ

**Optional Protocol Requirements**

เราสามารถกำหนด optional สำหรับ protocol ได้ โดยที่ใครก็ตามที่นำไป adopt และ conform ไม่ต้องกำหนดค่านี้ตาม ซึ่งจะแตกต่างกับ property และ method ใน protocol ที่ใครก็ตามที่นำไปใช้ จะต้องมีเหมือนกัน โดยการใส่ optional ใน protocol นั้น ทำได้ตามรูป

```
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
```

ตัวอย่างการ adopt โดยใช้แค่ fixedIncrement แค่อย่างเดียว ก็ไม่ถือว่ามีปัญหา ตามรูป

```
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
```

ตัวอย่างการ adopt โดยใช้แค่ func increment แค่อย่างเดียว ก็ไม่ถือว่ามีปัญหาเช่นกัน ตามรูป

```
class TowardsZeroSource: NSObject, CounterDataSource {
    func increment(forCount count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
```

## Generics

Generics code ทำให้สามารถเขียนโค้ดได้อย่างยืดหยุ่น สามารถนำฟังก์ชั่นมาใช้ซ้ำได้ และ type สามารถทำงานได้กับทุก type Generics เป็น feature หนึ่งของ Swift ที่ทรงพลังมากๆ library ของ Swift ส่วนใหญ่ล้วนสร้างมาจาก generics code ตัวอย่างเช่น Array และ Dictionary ก็เป็นส่วนหนึ่งของ generics คุณสามารถสร้าง array ที่เก็บค่า Int หรือค่า String หรือ type อื่นๆ ซึ่งสามารถสร้างได้ใน Swift ในทางเดียวกัน คุณก็สามารถสร้าง dictionary เพื่อเก็บค่า type เฉพาะซึ่งไม่ใช่จำกัดจำนวน type ที่จะเก็บอีกด้วย

### The Problem That Generics Solve

นี่คือ non-generics function ชื่อ swapTwoIntsที่สลับค่าของ 2 ตัวแปร Int function นี้จะทำการรับค่า a,b เข้ามาแล้วทำการสลับค่าระหว่าง a,b

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

จะเห็นว่า swapTwoInts fuction ใช้งานได้ดีเลย แต่มันสลับเฉพาะค่าที่ Int เท่านั้น

ถ้าคุณต้องการสลับค่า String, Double คุณก็จะต้องเขียนฟังก์ชั่นเพิ่ม

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}
func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

มากเลยต่างกันแค่ชนิดของตัวแปรที่รับเข้าไปเท่านั้นเอง

มันคงจะดีกว่านี้ถ้าเราเขียน function เดียวแล้วสามารถสลับได้ทุกชนิด นี่เป็นปัญหาหนึ่งที่ generic เข้ามาช่วยแก้ปัญหา

## Generic Functions

Generic functions สามารถใช้งานได้กับทุก type นี่เป็น generic version ของ swapTwoInts ที่เคยได้กล่าวไปชื่อswapTwoValues

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

ส่วนของ body ของ swapTwoValues function เป็นส่วนที่เหมือนกับ swapTwoInts function แต่บรรทัดแรกของ swapTwoValues จะแตกต่างไปจาก swapTwoInts สักหน่อย

ลองมาเปรียบเทียบดู

```
func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

ใน generic version ของ function จะกำหนด type เป็น T และเพิ่ม <T> ต่อท้ายชื่อฟังก์ชั่น

## Type Parameters

จากตัวอย่าง swapTwoValues ข้างต้น ตรงส่วนที่กำหนด type T ก็เป็นตัวอย่างหนึ่งของ type parameter

Type parameter สามารถถูกนำมาใช้ในการประกาศ type ของ parameter ในฟังก์ชั่นได้ด้วย หรือชนิดค่าที่ฟังก์ชั่นใช้ return ได้ด้วย

คุณมาสารถเพิ่ม type parameter ได้มากกว่า 1 ค่าโดยคั่นด้วย ;

## Naming Type Parameters

โดยทั่วไป generic function หรือ generic type จำเป็นต้องอ้างถึงตัวแปรเดี่ยวๆที่ใช้กำหนด type ปกติก็จะใช้ตัวแปร T สำหรับ type parameter

อย่างไรก็ตามคุณก็สามารถใช้ตัวแปรนอกจาก T ได้นะถ้าคุณต้องการประกาศ generic function ให้มันซับซ้อนเข้าไปอีก หรือ generic type ที่มีหลายๆ parameter ซึ่งก็ช่วยในการอธิบายหน้าที่ของตัวแปรด้วย

ตัวอย่างเช่น type ของ Dictionary ใน Swift ที่มี 2 type parameter ตัวแรกเป็น key อีกตัวเป็น value ถ้าคุณเคยเขียน Dictionary คุณอาจตั้งชื่อของ 2 ตัวแปรนี้เป็น KeyType และ ValueType เพื่อคอยเตือนว่าจะใช้แต่ละตัวจากอะไร

**Generic Types**

ใน Swift ให้คุณสามารถสร้าง generic types เป็นของตัวเองได้ ในส่วนนี้จะเป็นการบอกถึงวิธีการเขียน generic collection type ที่เรียกว่า Stack, Stack เป็นการเก็บค่าตามลำดับคล้ายๆ array แต่ operations ไม่ได้มีมากเท่า Array อนุญาตให้เพิ่ม item ใหม่ๆลงไปในส่วนไหนของ Array ก็ได้ แต่ Stack ไม่ใช่แบบนั้น item เพิ่มลงไปใน Stack จะอยู่ส่วนท้ายสุด และเมื่อต้องการนำ item ออกไปก็ต้องเอาส่วนท้ายสุดออกก่อนเสมอ

1. ในตอนเริ่มต้น Stack มี 3 item
2. Item ที่ 4 เข้ามาด้วย “push” และจะอยู่บนสุดของ Stack
3. ตอนนี้ใน Stack มี 4 item แล้ว
4. ส่วนบนสุดของ Stack กำลังถูกดึงออกไปแล้ว เราเรียก “pop”
5. หลังจาก pop ออกไป Stack ก็จะเหลือ 3 item

ด้านล่างนี้เป็นการเขียน non-generic version ของ Stack สำหรับ Stack ที่เก็บค่า Int

```
struct IntStack {
    var items = [Int]()
    mutating func push(_item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

Int Stack นีสามารถเก็บค่า Int ได้เพียงอย่างเดียว มันคงจะดีกว่านี ถ้าทำให้เป็น generic Stack ซึ่งสามารถช่วยจัดการค่าได้ทุก type

นี่คือโค้ดใน generic version ของ Stack

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

Stack

ในส่วนของ Element จะกำหนดอยู่ 3 ตำแหน่งคือ

1. ตอนสร้าง var item
2. ใน push nethod
3. ค่านี้จะถูก return โดย pop method

หากต้องการระบุว่า Stack ต้องการเก็บข้อมูลประเภทไหนก็สามารถทำได้โดยเพิ่ม <type> ในตอนสร้าง Stack ขึ้นมา

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
```

ซึ่งลำดับการทำงานของโค้ดด้านบนจะเป็นไปตามลำดับต่อไปนี้

เมื่อ pop ค่าออกจาก Stack ค่าที่อยู่บนสุดของ Stack จะถูก return และถูกลบออกจาก Stack ไปด้วย

## Type Constraints

swapTwoValues function และ Stack ที่กล่าวมาสามารถใช้ได้กับข้อมูลทุก type แต่บางครั้งเราจำเป็นต้องจำกัดชนิดของ type เพื่อความสะดวกในการใช้งาน

ตัวอย่างเช่น Dictionary ค่าที่จะเอาออกมาเป็น keys ใน dictionary ต้อง hashable นั่นคือต้องมีค่าไม่ซ้ำกัน Dictionary จำเป็นต้องมี key ที่ hashable เพื่อตรวจสอบว่ามีค่าที่ถูกเก็บใน key นั้นๆไปหรือยัง ถ้าไม่มีคุณสมบัตินี้ Dictionary จะไม่สามารถบอกได้ว่าควรจะinsert หรือแทนที่ค่าสำหรับ key ตัวไหนหรือไม่ก็ไม่สามารถหาค่าของ key ได้เพราะ key ซ้ำกัน

ความต้องการนี้เป็นสิ่งที่ถูกบังคับโดย type constraint สำหรับชนิดของ key ใน dictionary ที่จะต้องเป็นไปตาม Hashable Protocol

คุณสามารถกำหนด type constraint ได้ด้วยตัวเอง ซึ่งจะช่วยให้การเขียน generic มีประสิทธิภาพมากยิ่งขึ้น

## Type Constraint Syntax

คุณสามารถเขียน type constraints ได้โดยเพิ่ม class หรือ protocol ต่อท้าย type parameter โดยคั่นระหว่าง parameter และ protocol หรือ class ด้วย colon (:) ดังตัวอย่างข้างล่างนี้

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

function ด้านบนประกอบไปด้วย 2 type parameter

ตัวแรกเป็น T ที่จำกัดว่า T จะต้องเป็นsubclass ของ SomeClass

ตัวที่ 2 U ที่จำกัดว่า U จะต้องเป็นไปตามข้อบังคับของ SomeProtocol

**Type Constraints in Action**

โค้ดข้างล่างเป็น non-generic function หรือเรียก findStringIndex ที่รับค่า String เพื่อหาว่า Index ไหนเก็บค่า String ดังกล่าวอยู่ function จะ return ค่า index ออกมาเป็น Int และหากไม่เจอจะ return nil

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
```

ด้วย T แต่ค่าที่ถูก return ยังคงเป็น Int เพราะค่าดังกล่าว เป็นค่า index ซึ่งต้องระวังตรงนี้ด้วย

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
```

อย่างไรก็ดีฟังก์ชั่นข้างต้นจะไม่ถูก complie ปัญหาเกิดจาก “if value == valueToFind” เพราะไม่ใช่ข้อมูลทุกชนิดที่จะสามารถเปรียบเทียบกันได้ด้วย == ดังนั้นเราควรจำกัดให้เฉพาะค่าที่จะสามารถเปรียบเทียบกันได้ด้วย == ซึ่งใน Swift library จะมี protocol ที่ชื่อ Equatable ที่จะช่วยแก้ปัญหาตรงนี้ได้

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

## Access Control

Access Control จำกัดเขตการข้าวของโค้ดบางส่วนจากโค้ดที่มาจากแหล่งอื่นเช่น files หรือ modules feature นี้ทำให้คุณสามารถซ่อนรายละเอียดบางส่วนของโค้ดได้ และสามารถระบุเฉพาะได้ว่า interface ไหนที่สามารถเข้าใช้งานโค้ดนั้นๆได้

เราสามารถกำหนด level ของการเข้าเป็นส่วนๆแยกย่อยกันไปเช่น classes, structures และ enumerations และ properties, methods, initializers และ subscript อยู่ใน type นั้นๆด้วย protocols สามารถจำกัดได้ทั้ง context นั้น, global constants, variables และ functions

สำหรับการให้ระดับ access control ที่หลากหลาย Swift ลดความต้องการที่ต้องระบุ explicit access control level โดยให้ default access levels สำหรับ scenario นั้น แต่ถ้าคุณเขียน app ที่มีเป้าหมายเดียว เราอาจไม่จำเป็นต้องใช้ access control

> *Note*
>
> *Access control ที่มีหลากหลายมุมในโค้ด (properties, types, functions) จะเรียกว่า entities*

### Modules and Source Files

รูปแบบ access control ของ swift ขึ้นอยู่กับ Modules และ Source Files

Module เป็นส่วนของโค้ดส่วนหนึ่งที่เขียนไว้ใช้แจกจ่ายอย่างเช่น framework หรือ application ที่สร้างขึ้นและเอาไปใช้ เป็นหนึ่งส่วนย่อยๆสามารถใช้ คำสั่ง import เพื่อนำโค้ดส่วนนี้มาใช้

แต่ละเป้าหมาย เช่น app bundle หรือ framework ใน Xcode เป็น module แยกใน swift ถ้ากลุ่มพร้อมกันเห็นว่ามุมมองโค้ดของแอพแบบ stand-alone framework อาจจะเอามาใช้ได้อีกรอบใน application อื่น และจะทำให้ทุกสิ่งที่เราประกาศไว้เป็น module แยกย่อย

Source file เป็น Swift source code เดี่ยวภายใน module แม้ว่าธรรมดาจะประกาศไว้สำหรับ type หนึ่งๆ ใน source file ที่แยกไว้เดี่ยวๆ

**Access Levels**

Swift ให้ 3 access levels ที่แตกต่างกันสำหรับ entities ภายในโค้ด access level จะสัมพันธ์กับ source file ใน entity ที่ประกาศไว้และสัมพันธ์กับ module ที่ source file อยู่

- *Public access ให้ entities นั้นสามารถใช้ใน source file ใดๆจาก module ที่เขาประกาศไว้และ source file จาก module อื่นสามารถ import module นี้ได้ เราสามารถใช้ public access เมื่อ interface สาธารณะสามารถใช้ได้*
- *Internal access ให้ entities สามารถใช้ใน source file ใดๆจาก module ที่เขาประกาศไว้แต่ไม่ใช่จาก source file ข้างนอก module เราใช้ internal access เมื่อประกาศ แอพหรือframework ที่เป็นโครงสร้างภายใน*
- *Private access ห้ามการใช้ entity ของ source file ตัวเอง ใช้ private access เพื่อซ่อนส่วนรายละเอียดของโค้ดPublic access เป็น access level สูงสุด (จำกัดสิทธิ์น้อยสุด) และ Private access เป็น access level ล่างสุด (จำกัดสิทธิ์มากสุด)*

**Guiding Principle of Access Levels**

Access levels ใน Swift มีหลักว่า: ไม่มี entity ใดที่สามารถประกาศภายใต้เงื่อนไขของ entity อื่นที่อยู่ต่ำกว่า (เข้มงวดกว่า)

อย่างเช่น

- *ตัวแปรสาธารณะ (Public variable) ไม่สามารถมี internal หรือ private type เพราะว่า type อาจจะใช้ไม่ได้ในทุกที่ ที่ตัวแปรสาธารณะใช้*
- *Function ไม่สามารถมี access level ที่สูงกว่า parameter types และ return type เพราะ function จะใช้งานไม่ได้ในกรณีที่ type ของมันไม่มีให้ในโค้ด*

  ความหมายของคำต่างๆใน guiding principle สำหรับมุมมองที่แตกต่างกันของภาษาเขียนไว้ด้านล่างนี้คือ

### Default Access Levels

ทุก entities ในโค้ด (ที่มี exception ไม่มากอย่างที่อธิบายไปก่อนหน้า) มี default access level เป็น internal ถ้าเราไม่ได้ระบุ explicit access level เอง

### Access Levels for Single-Target Apps

ถ้าเขียนแอพโดยที่ไม่ได้มีเป้าหมายอื่น โค้ดในแอพจะเป็นลักษณะเฉพาะของตัวเองและไม่ต้องทำเพื่อใช้ได้ใน module ของแอพอื่น ดังนั้น default access level ของ internal เพียงพอกับความต้องการแล้ว ดังนั้นเราจึงไม่ต้องการระบุ access level เพิ่มเติม แต่ถ้าต้องการกำหนดบางจุดของโค้ดก็สามารถทำได้

### Access Levels for Framework

เมื่อเราพัฒนา framework , mark interface ที่เป็น public ไปที่ framework จะได้มุมมองและการเชื่อมต่อแบบ module อื่น อย่างเช่น แอพที่import framework public-facing interface คือ application programming interface (API) สำหรับ framework

> *Note*
>
> *รายละเอียดโค้ดใน framework ใดๆสามารถใช้ default access level ได้หรือสามารถ mark เป็น private ถ้าเราต้องการซ่อนส่วนนั้นออกจากส่วนอื่นๆของโค้ด framework เราต้อง mark entity เป็นแบบ public เท่านั้นเมื่อเราต้องการให้มันเป็นส่วนของ framework API*

**Access Control Syntax**

ประกาศ access level สำหรับ entity โดยวางคำว่า public, internal หรือ private ก่อน entity's introducer

```
public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
```

```
Public var SomePublicVariable = 0
Internal let someInternalConstant = 0
fileprivate func SomeFilePrivateFunction() {}
Private func somePrivateFunction() {}
```

แม้ว่าจะมีอย่างอื่นกำหนดไว้ default access level ก็ยังเป็น internal หมายความว่า SomeInternalClass และ someInternalConstant สามารถเขียนโดยที่ไม่ต้องแก้ไข explicit access level และยังมี access level แบบ internal:

```
class SomeInternalClass {}          //implicitly internal
let someInternalConstant = 0        //implicitly internal
```

**Custom Types**

ถ้าคุณต้องการระบุ explicit access level สำหรับ type เฉพาะ สามารถทำได้ที่ที่ประกาศ type ไว้ type ใหม่สามารถใช้ที่ไหนก็ได้ที่ access level อนุญาต ตัวอย่างเช่นถ้าประกาศ private class class สามารถใช้เป็น type ของ property เท่านั้นหรือเป็น function parameter หรือ return type ใน source file ที่ private class นั้นๆถูกประกาศ

access control level ของ type ก็มีผลกับ default access level ของ type's member (properties, method, initializers และ subscripts) ถ้าเราต้องการประกาศ access level ของ type เป็นแบบ private default access level ของ member ของมันจะเป็น private เหมือนกัน ถ้าเราประกาศ access level ของ type แบบ internal หรือ public (หรือใช้ default access level ของ internal โดยที่ไม่ระบุ explicit access level ) default access level ของ type member จะเป็น internal

*Note*

*อย่างที่กล่าวไปข้างต้น public type default คือมี internal members ไม่ใช่ public members ถ้าเราต้องการ type member เป็นแบบ public เราต้อง mark มันแบบ explicit requirement ที่ต้องการคือต้องแน่ใจว่า public-facing API สำหรับ type เป็นบางอย่างที่เราเลือกให้เปิดเผยและหลีกเลี่ยงการเปิดเผย internal working type เป็น public API*

```
public class SomePublicClass {                  // explicitly public class
    public var somePublicProperty = 0           // explicitly public class member
    var someInternalProperty = 0                // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class member
    private func somePrivateMethod() {}         // explicitly private class member
}

class SomeInternalClass {                       // implicitly internal class
    var someInternalProperty = 0                // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class member
    private func somePrivateMethod() {}         // explicitly private class member
}

fileprivate class SomeFilePrivateClass {        // explicitly file-private class
    func someFilePrivateMethod() {}             // implicitly file-private class member
    private func somePrivateMethod() {}         // explicitly private class member
}

private class SomePrivateClass {                // explicitly private class
    func somePrivateMethod() {}                 // implicitly private class member
}
```

Access level สำหรับ tuple type เป็น access level ที่จำกัดสิทธิ์มากที่สุดสำหรับ types ที่ใช้ใน tuple นั้น ตัวอย่างเช่น ถ้าเรารวม tuple จากสอง types ที่แตกต่างกัน หนึ่งคือ internal access อีกหนึ่งคือ private access access level ที่ได้จะเป็น private

*Note*
*Tuple ไม่ได้มีความหมายเหมือนกับที่ประกาศ classes, structures, enumerations และ functions Tuple access level สามารถอนุมานได้อัตโนมัติเมื่อ tuple ถูกใช้และไม่สามารถระบุแบบ explicit ได้*

**Function Types**

Access level สำหรับ function ถูกคำนวณแล้วว่า เป็น access level ที่แน่นหนาที่สุดสำหรับ parameter ของ function และ return type เราต้องระบุ access level แบบ explicit ด้วยเหมือนเป็นส่วนหนึ่งของการประกาศ function ถ้า access level ของ function นั้นคิดแล้วไม่ตรงกับบริบทของ default

ตัวอย่างด้านล่างประกาศ global function ชื่อ someFunction โดยที่ไม่มี access level เฉพาะ function เราคาดหวังว่า default access level จะเป็น internal แต่ในกรณีนี้ someFunction จะไม่ใช่แบบนั้น

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    //function implementation goes here
}
```

Types

หนึ่งในนั้นเป็น internal อีกนั้นเป็น private ดังนั้น access level รวมกันจะได้ private

เพราะว่า return type ของ function เป็นแบบ private เราต้อง mark access level ของ function ด้วยคำว่า private ที่การประกาศ function เพื่อให้ compile ผ่าน

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
//function implementation goes here
}
```

defaultเพราะว่าผู้ใช้ publiac หรือ internal อาจจะไม่มีคุณสมบัติที่สามารถเข้าถึง private class ที่ใช้ใน return type ของ function

## Enumeration Types

Cases ของ enumeration จะเอา access level ของ enumeration ที่มันอยู่มาโดยอัตโนมัติ เราไม่สามารถกำหนด access level ที่แตกต่างให้กับแต่ละ case ได้

ในตัวอย่างด้านล่าง CompassPoint enumeration มี access level แบบ public ดังนั้น case North, South, East, West ก็จะมี access level แบบ public

```
public enum CompassPoint {
case North
case South
case East
case West
}
```

**Raw Values and Associated Values**

ประเภทของ access level ของ raw value หรือ associated value ใดๆใน enumeration จะต้องมี access level อย่างน้อยสูงเท่ากับ access level ของ enumeration เราไม่สามารถใช้ private กับ raw value ใน enumeration ที่เป็น internal

**Nested Types**

ประเภท Nested ประกาศภายใน private จะมี access level แบบ private อัตโนมัติ ประเภท Nested ที่ประกาศภายใน public หรือ internal จะมี access level เป็นแบบ internal ถ้าเราต้องการ nested type ที่อยู่ใน public ให้เป็นประเภท public ด้วย เราต้องประกาศด้วยตนเองว่า nested type นี้เป็นแบบ public

**Subclassing**

เราสามารถทำ subclass บน class ใดๆก็ได้ที่สามารถเข้าถึงได้โดย access แบบปัจจุบันแต่ Subclass ไม่สามารถมี access level ที่สูงกว่า superclass ของมันได้ อย่างเช่น เราไม่สามารถเขียน public subclass ที่มี superclass เป็น internal

อีกอย่างคือเราสามารถเขียนทับ (override) class member ใดๆ (method, property, initializer หรือ subscript) ที่เราเห็นและเข้าถึงได้ใน access level ปัจจุบัน

การเขียนทับสามารถทำ inherited class member ให้เข้าถึงได้มากกว่า version ของ superclass ดังตัวอย่างด้านล่าง class A เป็น public class ที่มี private method ชื่อ someMethod class B เป็น subclass ของ A โดยมี access level แบบ internal ถึงอย่างนั้น class B เขียนทับ someMethod ด้วย access level แบบ internal ซึ่งสูงกว่าของต้นฉบับ someMethod เดิม

```
public class A {
    private func someMethod() {}
}
internal class B: A{
    override internal func someMethod() {}}
```

การเขียนแบบนี้ผ่านสำหรับ subclass member ที่เรียก superclass member ที่มี access level ต่ำกว่า subclass member จนกว่าการเรียก superclass member จะกระทำใน access level ที่ยินยอม (เช่น source

file เดียวกันที่มี superclass สำหรับ access level แบบ private หรือภายใน module เดียวกันที่มี superclass สำหรับ access level แบบ internal)

```
public class A {
    private func someMethod() {}
}

internal class B: A{
    override internal func someMethod() {
        super.someMethod()
    }
}
```

เพราะว่า superclass A และ subclass B ถูกประกาศไว้ใน source file เดียวกัน มันจึงถูกต้องสำหรับ B ที่เรียก super.someMethod()

## Constants, Variables, Properties and Subscripts

ค่าคงที่(constant), ตัวแปร(variable) หรือ คุณสมบัติ(property) ไม่สามารถเปิดเผยได้มากกว่าประเภทของมัน มันไม่ถูกต้องที่จะเขียน public property ด้วย private type ตัวอย่างเช่น subscript ไม่สามารถเปิดเผยได้มากกว่า index type หรือ return type

ถ้าค่าคงที่, ตัวแปร, คุณสมบัติ หรือ subscript ทำไว้สำหรับ private ทุกๆประเภทต้องเขียนด้วยคำว่า private

```
private var privateInstance = SomePrivateClass()
```

**Getters and Setters**

Getters และ Setters สำหรับค่าคงที่ ตัวแปร คุณสมบัติ และ subscripts รับ access level เดียวกันกับ ค่าคงที่ ตัวแปร คุณสมบัติ และ subscripts ที่มันอยู่แบบอัตโนมัติ

เราสามารถให้ setter มี access level ที่ต่ำกว่า getter ที่ทำหน้าที่ร่วมกัน เพื่อจำกัดขอบเขตการอ่านเขียนของตัวแปร, คุณสมบัติ หรือ subscript เรากำหนด access level ที่ต่ำกว่าโดยเขียน private(set) หรือ internal(set) ก่อน var หรือ subscript

*Note*

*กฎนี้ใช้เพื่อ properties ตัวกักเก็บ เช่นเดียวกับ properties ตัวคำนวณ แม้ว่าเราไม่ได้เขียน explicit getter และ setter สำหรับ properties ตัวกักเก็บ Swift จะจัดการสร้าง implicit getter และ setter เพื่อเตรียมการเข้าถึง property ตัวกักเก็บ ใช้ private(set) และ internal(set) เพื่อเปลี่ยน access level ของ setter ที่ถูกสร้างขึ้นมาเอง และ property ตัวคำนวณ ก็เป็นแบบนี้เหมือนกัน*

```
Struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
}
```

TrackedString structure ประกาศ ตัวเก็บค่า string ชื่อ value โดยที่ค่าเริ่มต้นคือ "" (empty string) และยังประกาศตัวเก็บ integer property ชื่อ numberOfEdits ที่ใช้เพื่อติดตามตัวเองของเวลาที่ value

เปลี่ยนค่า การติดตามการเปลี่ยนแปลงด้วย didSet ตัวดู property บน value ซึ่งเพิ่ม numberOfEdits ทุกเวลา property ของ value จะเปลี่ยนเป็นค่าใหม่

Structure TrackedString และ property ของ value ไม่มี explicit access level ดังนั้นมันจึงรับ default access level แบบ internal มา แต่ access level สำหรับ numberOfEdits property ถูกmarkไว้ด้วย private(set) เพื่อบอกว่า property ควรตั้งค่าได้จากภายใน source file เดียวกันกับ TrackedString ถูกประกาศไว้ getter ของ property ยังมี access level แบบ internal แต่ setter เป็นแบบ private เพื่อ source file ที่ TrackedString ถูกประกาศไว้ นี่ทำให้ TrackedString เปลี่ยนค่า numberOfEdits property ได้แบบภายใน(internal) แต่ source file อื่นใน module เดียวกัน จะเขียนได้เท่านั้น

ถ้าเราสร้าง TrackedString instance และเปลี่ยนค่า string สักสองสามครั้งเราสามารถเห็นค่า numberOfEdits property อัพเดทเพื่อตรงกับ เลขของการเปลี่ยนค่า

```
var stringToEdit = TrackedString()
stringToEdit.value = “This string will be tracked.”
stringToEdit.value += “ This edit will increment numberOfEdits.”
stringToEdit.value += “ So will this one.”
Print(“The number of edits is \(stringToEdit.numberOfEdits)”)
```

แม้ว่าเราสามารถ query ค่าปัจจุบันของ numberOfEdits property จาก source file อื่นได้ แต่เราไม่สามารถ เปลี่ยนแปลง property จาก source file อื่นได้ ขอจำกัดนี้ป้องกันโค้ดของ TrackedString แก้ไขฟังก์ชั่นการทำงานของมัน ขณะที่มันยังใช้งานในที่อื่น

เราสามารถกำหนด explicit access level ทั้ง getter และ setter ได้ถ้าต้องการ ในตัวอย่างด้านล่างแสดงเวอร์ชั่นของ TrackedString structure ที่ที่มันถูกประกาศโดยมี explicit access level แบบ public memberของstructure รวมทั้ง numberOfEdits property จะมี access level แบบ internal โดยdefault เราสามารถสร้าง structure ของ numberOfEdits property getter เป็น public และ property setter เป็น private โดยรวมคำที่เป็นตัวแก้ไขaccess level public และ private(set)

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
```

**Initializers**

Initializers แบบทำขึ้นเองสามารถกำหนด access level น้อยกว่าหรือเท่ากับ ประเภทที่มัน initialize ข้อยกเว้นเดียวคือสำหรับ required initializer ซึ่งต้องมี access level เดียวกับ class ที่มันอยู่

ด้วย function และ method parameter ประเภทของ parameters initializer ไม่สามารถเป็น private (ส่วนตัว) ได้มากกว่า access level ของ initializer ตัวเอง

**Default Initializers**

Swift จัดการ default initializer ให้อัตโนมัติโดยที่ไม่มี argument ใดสำหรับ structure ใดๆหรือ base class ที่จัดการ default value สำหรับ properties ทั้งหมดและไม่ให้ initializer ตัวใดของมันเลย

Default initializer มี access level แบบเดียวกับของ initializes ยกเว้นประกาศเป็น public สำหรับประเภท public default initializer จะกำหนดให้เป็น internal ถ้าเราต้องการประเภท public ให้ initial ได้โดยที่ไม่มี argument สำหรับ initializer เมื่อใช้ใน module อื่นเราต้องทำ explicit access level ให้

**Default Memberwise Initializers for Structure Types**

Default memberwise initializer สำหรับstructure กำหนดให้เป็น private ถ้า properties ของ structureใดๆเป็นแบบ private ไม่งั้นก็จะมี access level แบบ internal

ด้วย default initializer ข้างต้น ถ้าเราต้องการ public สำหรับ structure เพื่อให้ initial ได้สำหรับ memberwise initializer เมื่อใช้ใน module อื่นเราต้องเตรียม public memberwise initializer ด้วยตัวเราเอง

**Protocols**

ถ้าเราต้องกำหนด explicit access level สำหรับ protocol ทำต้องจุดที่เราประกาศ protocol เลย นี้จะทำให้เราสร้าง protocols ที่สามารถใช้ได้ใน access แบบนั้น

Access level ของแต่ละความต้องการ(requirement)ภายในการประกาศ protocol จะถูกเซตอัตโนมัติที่ access level เดียวกับ protocol เราไม่สามารถเซตprotocolของความต้องการ(requirement)ให้แตกต่างจาก access level ที่protocolรองรับ นี้ทำให้แน่ใจว่า requirement ของ protocol จะมองเห็นได้ด้วย protocol ทุกประเภท

*Note*

*ถ้าเราต้องการประกาศ public protocol ตัวrequirementต้องการ access level แบบ public สำหรับแต่ละrequirementที่มาเขียนด้วย บอกได้ว่านี่เป็นพฤติกรรมที่แตกต่างจากประเภทอื่นๆที่ public เขียนกับ access level แบบ internal สำหรับmemberของมัน*

แค่ตาม protocol ที่มัน inherit มา เราไม่สามารถเขียน public protocol จาก protocol ที่มาจาก internal ได้

**Protocol Conformance**

Protocol จะทำตามประเภทโดย access level ที่ต่ำกว่าตัวมันเอง ตัวอย่างเช่นเราสามารถประกาศ publicที่สามารถใช้ใน module อื่นได้แต่ผู้อื่นที่สอดคล้องกับ internal protocol ใช้ได้แค่ภายใน module ของinternal protocol

ประเภทของtypeที่ทำตามprotocol เป็นขั้นต่ำของ access level ของtype และของ protocol ถ้าtype เป็น publicแต่ protocol ทำตาม internal ความสอดคล้องของtypeก็จะเป็น internal

เมื่อเราเขียนหรือเพิ่ม type เพื่อทำตามprotocol เราต้องแน่ใจว่าการดำเนินงาน(implementation) ของ type ของแต่ละ protocol requirement มีaccess level อย่างน้อยก็เหมือนกับ ความสอดคล้อง (conformance) ของ type สำหรับ protocol เองตัวอย่างเช่น ถ้า type แบบ publicทำตาม internal protocol การดำเนินงานของ type ของแต่ละ protocol requirement ต้องเป็น internal เป็นอย่างน้อย

*Note*

*Swift และ Objective-C ความสอดคล้องของprotocolเป็น global มันเป็นไปไม่ได้ที่ type จะทำตาม protocol ที่แตกต่างกันภายใน program เดียว*

เราสามารถ extend class, structure หรือ enumeration ใน access ใดๆ ที่class, structure หรือ enumeration มีไว้ให้. Type member ใดๆที่ถูกเพิ่มใน extension จะมี default access level แบบเดียวกับที่ type member ประกาศไว้ที่ type ต้นฉบับที่ extend มา ตัวอย่างเช่นถ้าเรา extend public type typemember ใหม่ใดๆที่เราเพิ่มเข้ามาจะมี default access level เป็น internal

อีกวิธีหนึ่งเราสามารถ mark extension ด้วยคำ explicit access level (เช่น private extension) เพื่อเซต default access level ใหม่สำหรับทุก members ที่ประกาศภายใน extension default access level ใหม่นี้ยังสามารถเขียนทับภายใน extensionสำหรับtype members เฉพาะแต่ละตัวด้วย

**Adding Protocol Conformance with an Extension**

เราไม่สามารถให้คำสั่ง explicit access level สำหรับ extensionถ้าเราใช้ extension นั้นเพื่อเพิ่มความสอดคล้องของ protocol แม้ว่า access level ของตัวprotocolเองใช้เพื่อให้default access level สำหรับ protocol requirement ใดๆภายใน extension

**Generic**

Access level สำหรับ generic type หรือ generic function นั้นเป็นขั้นต่ำของ access level ของ generic type หรือ function ตัวมันเองอยู่แล้วและ access level ของ type constraints ใดๆบน type parameters

**Type Aliases**

Type aliases ใดๆที่เราประกาศเป็น distinct type จุดประสงค์เพื่อ access control type aliases สามารถมี access level ที่ต่ำกว่าหรือเท่ากับ access level ของ type ที่มันเป็นนามแฝง ตัวอย่างเช่น private type aliases ไม่สามารถเป็นนามแฝงของ internal หรือ private ได้

*Note*

*กฎนี้ใช้กับ type aliases สำหรับ associated types ที่ใช้เพื่อสอดคล้องกับ protocol*

นอกจาก Basic Operators ต่างๆที่กล่าวไว้ข้างต้น Swift นั้นยังมี Advanced operators อีกบางส่วนเพิ่มเติมเพื่อจัดการกับ ค่าที่ซับซ้อนได้ ซึ่ง operator ที่เพิ่มมานั้นรวมถึง bitwise และ bit shifting ด้วย

arithmetic operator ต่างๆใน Swift นั้นจะไม่เหมือนใน ภาษา C คือจะไม่มีการ Overflow โดย default โดยที่ การOverflow นั้นจะถูกดักจับและถูกตรวจสอบว่าเกิดข้อผิดพลาดขึ้น โดยSwiftจะมีทางเลือกสำหรับ arithmetic operators ที่สามารถเกิด Overflow ได้เช่น เครื่องหมาย (+) จะใช้ เป็น (&+) โดยมี ampersand(&)นำหน้า operator นั้นๆ

Swift นั้นยังให้อิสระในการสาร operators ต่างๆได้เองไม่ว่าจะเป็นการกำหนด infix, prefix, postfix หรือจะกำหนด precedence , associativity values ได้เอง

**Bitwise Operators**

bitwise operators นั้นสามารถให้คุณจัดการกับ ข้อมูลดิบ (raw data bits) ด้วยโครงสร้างข้อมูล ซึ่งส่วนมากถูกใช้ในภาษาระดับ low-level programming. bitwise นั้นจะเป็นประโยชน์อย่างมากในการใช้งานกับ raw data ที่มาจากข้อมูลภายนอก ตัวอย่างเช่นการ encoding, decoding ข้อมูลผ่านทางการสื่อสารจาก protocol.

Swift นั้นสนับสนุนการใช้ bitwise operators ต่างๆที่พบได้ในภาษา C ตามคำอธิบายข้างล่าง

**Bitwise NOT Operator(~)**

Invert ทุกบิทที่ถูก operator นี้กระทำ

bitwise NOT operator เป็น prefix operator โดยจำขึ้นต้นก่อน value ต่อกันโดยไม่มี white space

```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

UInt8 เป็นข้อมูลขนาด 8 bits โดยจะเก็บค่าระหว่าง 0 - 255 ดังตัวอย่างนี้ จะinitialize ค่าเป็น 00001111 หรือ 15 และทำการ กลับบิทจาก ~ จะกลายเป็น 11110000 ซึ่งมีค่าเป็น 240

**Bitwise AND Operator(&)**

bitwise AND จะเอาตัวเลข 2 ตัวมาเปรียบเทียบกันและจะ return ค่าใหม่ออกมา จะ return 1 ถ้าเลขทั้งคู่เป็น 1

ดังตัวอย่างนี้

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100
```

**Bitwise OR Operator(|)**

เปรียบเทียบตัวเลข2 ตัวและreturn 1 ถ้าตัวใดตัวหนึ่งมีค่าเป็น 1 นอกจากนี้จะreturn 0

ดังตัวอย่างต่อไปนี้

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110
```

**Bitwise XOR Operator(^)**

จะreturn ค่า 1 เมื่อbit แตกต่างกัน ถ้าเหมือนกันจะ return 0

ดังตัวอย่างต่อไปนี้

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

**Bitwise Left(<<) and Right Shift(>>) Operators**

เลื่อน bit ไปทางขวาหรือซ้ายตามตำแหน่งที่กำหนด โดยที่จะเปิดจากการที่ คูณ หรือ หาร ด้วย 2 คือ ถ้าเลื่อนบิทไปทาง ซ้ายจะเท่ากับว่าเพิ่มค่าเป็น 2 เท่า ถ้าเลื่อนบิทไปทางขวาคือ ลดค่าเป็น 2 เท่า

**Shifting Behavior for Unsigned Integers**

bit-shifting จะมีลักษณะดังนี้

1. จำนวนบิทจะถูกเลื่อนไปทางขวาหรือซ้ายตามจำนวนตำแหน่งที่กำหนด
2. บิทที่ถูกเลื่อนจะเกินขอบเขตจะถูกกำจัดทิ้ง
3. 0 ถูกแทนที่ตำแหน่งก่อนหน้าที่บิทนั้นเลื่อน

ลักษณะแบบนี้จะเรียกอีกอย่างว่า logical shift

ตัวอย่างข้างล่างแสดงให้เห็นถึง การเลื่อนบิท 11111111 << 1 และ 11111111 >> 1

ดังตัวอย่างโค้ดต่อไปนี้

```
let shiftBits: UInt8 = 4   // 00000100 in binary
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

เราสามารถ bit shifting เพื่อ encoding decoding สำหรับ data type อื่นๆได้

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent is 0xCC, or 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent is 0x66, or 102
let blueComponent = pink & 0x0000FF           // blueComponent is 0x99, or 153
```

**Shifting Behavior for Signed Integer**

shifting behavior จะมีความซับซ้อนเมื่อข้อมูลเป็น signed interger เพราะ signed integer จะใช้บิทแรกนั้นบอกถึงเครื่องหมาย บวกหรือลบ โดย 0 หมายถึงบวก 1 หมายถึง ลบ โดยจะเรียกบิทนี้ว่า value bits ดังตัวอย่างต่อไปนี้

การ encoding สำหรับ จำนวนลบเราจะเรียกอีกอย่างหนึ่งว่า two’s complement representation ดังตัวอย่างนี้

การ shifting ของ two'complement จะเป็นไปตามข้อกำหนดดังนี้

- เมื่อทำการ shift signed integer ไปทางขวา ให้ทำเหมือนการ shift unsigned interger แต่จะเติมช่องว่างทางซ้ายสุดด้วย signed bit แทนการเติม 0 ลงไป

การทำดังตัวอย่างนี้จะเป็นการยืนยันแน่นอนว่า บิทเครื่องหมายยังคงอยู่เราเรียกการ shift แบบนี้ว่า arithmetic shift

**Overflow Operators**

เมื่อคุณพยายาม กำหนดค่าให้กับตัวแปรต่างๆที่เป็น constant ซึ่งไม่สามารถรับค่านั้นได้แล้ว โดย Dafault Swift จะแจ้งเตือน และแสดง error แทนที่จะสร้างตัวแปรที่ผิดพลาดขึ้นมา
ดังตัวอย่างนี้คือ Int16 ที่รับคค่าระหว่าง -32768 - 32767 ถ้ากำหนดค่าเกินนี้จะ error

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the largest value an Int16 can hold
potentialOverflow += 1
// this causes an error
```

อย่างไรก็ตามถ้าต้องการให้เกิดการ Overflow เราจะใช้ Operator ที่เรียกว่า Overflow operators ดังนี้

- *Overflow addition(&+)*
- *Overflow subtraction(&-)*
- *Overflow nultication(&*)*
- *Overflow division(&/)*
- *Overflow remainder(&%)*

**Value Overflow**

นี่คือตัวอย่างของการใช้ Overflow addition(&+)

```
var unsignedOverflow = UInt8.max
// unsignedOverflow equals 255, which is the maximum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &+ 1
// unsignedOverflow is now equal to 0
```

เห็นถึงการOverflow

**Value Underflow**

อาจเกิดขึ้นได้เหมือนกันเมื่อค่านั้นน้อยเกินกว่าค่าที่จะรับได้ดังตัวอย่างนี้

ค่าที่น้อยที่สุดที่ Unint8 รับได้คือ 0 ถ้าลบไป 1 จะunderflow เป็น 11111111

และนี่คือตัวอย่างcode

```
var unsignedOverflow = UInt8.min
// unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1
// unsignedOverflow is now equal to 255
```

และสำหรับ Int8 ดังตัวอย่างนี้

```
var signedOverflow = Int8.min
// signedOverflow equals -128, which is the minimum value an Int8 can hold
signedOverflow = signedOverflow &- 1
// signedOverflow is now equal to 127
```

ถ้าเอาตัวเลขใดๆ หารด้วย 0 หรือ modular ด้วย 0 จะทำให้เกิด error ซึ่ง overflow จะใช้ &/ และ &% แทน

```
let x= 1
let y= x &/0
// y is equal to 0


let x = 1
let y = x &/ 0
// y is equal to 0
```

**Precedence and Association**

Operator Precedence ถ้า operator ตัวไหน มี priority สูงกว่าจะทำตัวนั้นก่อน Operator Association คือ จะ รวมคู่การทำoperation แล้วดูว่าจะทำจากซ้ายไปขวา หรือ ขวาไปซ้าย

ดังตัวอย่างต่อไปนี้

```
2 + 3 * 4 % 5
// this equals 4
```

โดยถ้าเราทำจากซ้ายไปขวาจะได้

- *2 บวก 3 ได้ 5*
- *5 คูณ 4 ได้ 20*
- *20 % 5 ได้ 0*

กระบวนการที่ถูกต้องที่เขียนได้คือ

- *2 + ((3 * 4) % 5)*

## Operator Function

class และ structure สามารถสร้าง operator ของตัวเองได้เราเรียกวิธีนี้ว่า Overload Operator

วิธีข้างล่างจะทำการ สร้าง Operator (+) สำหรับ class ของตัวเอง

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y + right.y)
    }
```

ถ้าเราทำการใช้งาน ตามโค้ดนี้

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

โค้ดข้างต้นเป็นการ บวกเวกเตอร์ 3.0,1.0 กับ 2.0,4.0 จะได้ เวกเตอร์ 5.0,5.0

## Prefix and Postfix Operators

ตัวอย่างของ prefix และ postfix unary operator เช่น -a (prefix) , i++(postfix)

เราทำการสร้าง prefix กับ postfix ได้โดย เขียน modifier เป็น prefix หรือ postfix นำหน้า func keyword เมื่อจะกำหนด operator ดังนี้

```
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
```

และนี่คือตัวอย่างเมื่อนำไปใช้

```
        let positive = Vector2D(x: 3.0, y: 4.0)
        let negative = -positive
        // negative is a Vector2D instance with values of (-3.0, -4.0)
        let alsoPositive = -negative
        // alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

**Compound Assignment Operators**

คือการนำเครื่องหมาย = ไปรวมกับเครื่องหมายอื่นเช่น += จำทำ operation และทำการกำหนดค่าหลังการทำ operation

ดังตัวอย่างการสร้างดังนี้

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
```

และเมื่อนำไปใช้

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

**Equivalence Operators**

เครื่องหมายเปรียบเทียบการเท่ากัน(==) และ ไม่เท่ากัน(!=) โดยจะ return ค่าเป็น true หรือ false

```
extension Vector2D {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) && (left.y == right.y)
    }
    static func != (left: Vector2D, right: Vector2D) -> Bool {
        return !(left == right)
    }
```

ตัวอย่างการใช้งานดังนี้

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
```

## Custom Operators

การทำcustom operator เหมือนตัวอย่างที่ผ่านมา โดยจะต้องไปดู list ที่สามารถทำ custom operator เช่น ตัวอย่าดังต่อไปนี้ คือการทำ prefix operator สำหรับ +++

```
extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
        return vector
    }
}
```

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

**Precedence and Associativity for Custom Infix Operators**

ตัวอย่างการทำ precedence และ associativity เราจะใช้ infix operator ดังตัวอย่างต่อไปนี้

```
infix operator +-: AdditionPrecedence
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y - right.y)
    }}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

# สรุป

เพื่อให้การพัฒนาสามารถดำเนินการได้อย่างมีประสิทธิภาพ ในการเตรียมเบื้องต้นเตรียมคอมพิวเตอร์ในระบบ OSX และอุปกรณ์ที่ใช้ในการทดสอบแอพพลิเคชั่นแล้ว ต้องทำการดาวน์โหลด Xcode ซึ่งเป็นกลุ่มของซอฟต์แวร์ที่ออกแบบมาสำหรับการพัฒนาแอพพลิเคชั่นโดยเฉพาะจาก App Store เมื่อต้องการเริ่มสร้างโปรแกรม

หลักการในการเขียนโปรแกรมภาษา SWIFT จะใช้โครงสร้างในลำดับในการเขียนโปรแกรม และการประมวลผลในส่วนของการทำงานตามขั้นตอนของการวางไว้ในโปรแกรม มีความคล้ายคลึงกับภาษา C และ Objective C ซึ่งชนิดของข้อมูลใน Swift ได้แก่ Int, Double, Float, Bool, String, Collection Type, Array, Dictionary และเพิ่มเติมในส่วนของ Truples (Group ค่าที่มีชนิดข้อมูลต่างกันได้ และ Optionaltype (คล้ายกับ pointer)

การทำงานของ control flow จะล้อกับหลกการของภาษา c ซึ่งประกอบด้วย for และ while เพื่อทำงานเป็น loop หลายๆครั้ง และ if switch เพื่อทำงาน แบบทางเลือก(condition) และ break หรือ continue เพื่อเปลี่ยนสถานะของ flow การทำงานไปที่จุดต่างๆ ใน code ทั้งนี้ Swift ยังมีการใช้ Loop ที่ชื่อว่า for-in เพื่อทำซ้ำในตัวแปร เช่น arrays, dictionaries, range, strings และ sequences

การจัดการ Initializer จะทำให้อัตโนมัติโดยที่ไม่มี argument ใดสำหรับ structure ใดๆหรือ base class ที่จัดการ default value สำหรับ properties ทั้งหมดและไม่ให้ initializer ตัวใดเลย ทั้งนี้Default initializer มี access level แบบเดียวกับของ initializes ยกเว้นประกาศเป็น public สำหรับประเภท public default initializer จะกำหนดให้เป็น internal

นอกจาก Basic Operators ต่างๆที่กล่าวไว้ข้างต้น Swift นั้นยังมี Advanced operators อีกบางส่วน เพิ่มเติมเพื่อจัดการกับ ค่าที่ซับซ้อนได้ ซึ่ง operator ที่เพิ่มมานั้นรวมถึง bitwise และ bit shifting ด้วย รวมถึง Arithmetic Operator ต่างๆใน Swift นั้นจะไม่เหมือนใน ภาษา C คือจะไม่มีการ Overflow โดย default โดยที่ การOverflow นั้นจะถูกดักจับและถูกตรวจสอบว่าเกิดข้อผิดพลาดขึ้น โดยSwiftจะมีทางเลือกสำหรับ arithmetic operators ที่สามารถเกิด Overflow ได้เช่น เครื่องหมาย (+) จะใช้ เป็น (&+) โดยมี ampersand(&)นำหน้า operator นั้นๆ

ในการสร้าง Function เมื่อสร้างขึ้นมา สามารถปรับแต่งได้หลาย ๆ อย่าง เช่น ชื่อ ประเภทของอินพุท ประเภทของเอาท์พุท ทุก ๆ ฟังก์ชั่นจะมีชื่อเป็นของตัวเอง ที่จะบ่งบอกรายละเอียดว่าฟังก์ชั่นนั้น ๆ มีหน้าที่ทำอะไร เวลาที่ต้องการเรียกใช้ฟังก์ชั่นก็แค่เรียกผ่านชื่อของฟังก์ชั่นแล้วตามด้วยค่าที่ต้องการส่งไปให้เป็นอินพุทของฟังก์ชั่นนั้น ๆ (หรือเรียกว่าอาร์กิวเมนท์) โดยจะต้องตรงกันกับประเภทของพารามิเตอร์ในฟังก์ชั่นด้วย โดยปกติแล้วอาร์กิวเมนท์ที่จะส่งให้ฟังก์ชั่นนั้นจะมีลำดับการเรียงเหมือนฟังก์ชั่นพารามิเตอร์

Swift นั้นยังให้อิสระในการสร้าง operators ต่างๆได้เองไม่ว่าจะเป็นการกำหนด infix, prefix, postfix หรือจะกำหนด precedence , associativity values ได้เอง

การเขียนภาษานี้จึงมีอิสระในการดำเนินการ พัฒนาเพิ่มเติมให้มีอิสระ โดยเฉพาะกับอุปกรณ์ในระบบ iOS ที่มีคุณสมบัติในการดำเนินการโดยเฉพาะ

## แบบฝึกหัด

- จงเขียนโปรแกรมให้ได้รูปสี่เหลี่ยมมีขนาดความกว้างยาวตามที่กำหนด เช่น ถ้ารับค่าเป็น 3 จงแสดงผลเป็น
  ***
  ***
  ***

- จงเขียนโปรแกรมที่แสดงสามเหลี่ยมที่มีขนาดฐานตามที่กำหนด โดยให้สามเหลี่ยมเอียงซ้าย เช่น ถ้ารับค่าเป็น 3 จงแสดงผลเป็น
  *
  **
  ***
- จงเขียนโปรแกรมที่แสดงสามเหลี่ยมที่มีขนาดฐานตามที่กำหนด โดยให้สามเหลี่ยมเอียงขวา เช่น ถ้ารับค่าเป็น 3 จงแสดงผลเป็น
    *
   **
  ***
- จงเขียนโปรแกรมที่แสดงสามเหลี่ยมที่มีขนาดฐานตามที่กำหนด โดยให้สามเหลี่ยมสมมาตร เช่น ถ้ารับค่าเป็น 3 จงแสดงผลเป็น
   *
  **
  ***
- จงเขียนโปรแกรมที่คำนวณคะแนนและเกรดของ นักศึกษา ตามเกณฑ์ ดังต่อไปนี้

  score >= 80 : A
  70 <= score < 80 : B
  60 <= score < 70 : C
  50 <= score < 60 : D
  score < 50 : F

- จงเขียนโปรแกรมที่คำนวณอนุกรม Fibonacci โดยให้รับค่า input กำหนดตำแหน่งของอนุกรม
- จงเขียนโปรแกรมเพื่อตรวจสอบว่าค่าจำนวนเต็มที่ได้รับเป็นจำวนเฉพาะหรือไม่
- จงเขียนโปรแกรมเพื่อให้แสดงว่าวันสุดท้ายของเดือนนี้ตรงกับ วัน และ วันที่อะไร

# บรรณานุกรม


[1.] M. Galloway, 'Swift Tutorial', https://www.raywenderlich.com/, (accessed Jan 2015)

[2.] Udemy Acdemy, 'Introduction to Swift', https://www.udemy.com/introduction-to-swift/?dtcode=MjTZm4O2V2LX, (accessed Jan 2015)

[3.] Appcoda, 'Getting Started with Swift', https://www.appcoda.com/swift-programming-language-intro/, (accessed Jan 2015)

[4.] V. Standord, 'Developing iOS Apps with Swift', https://itunes.apple.com/de/course/developing-ios-8-apps-swift/id961180099, (accessed Jan 2015)

[5.] Apple Company, 'The Swift Programming Language', https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html, (accessed Jan 2015)

[6.] Weheartswift, 'Swift Programming from Scratch', https://www.weheartswift.com/swift-programming-scratch-100-exercises/, (accessed Jan 2015)

[7.] C. Chares. 'Let's Make a Swift App', http://blog.chares.io/lets-make-a-swift-app/, (accessed Jan 2015)

[8.] M. Thompson and N. Cook, 'Obscure Topics in COCOA & SWIFT', https://nshipster.com/, (accessed Jan 2015)

[9.] HackingWithSwift,'Hacking with Swift', https://www.hackingwithswift.com/, (accessed Jan 2015)

[10.] N. Hanan, 'From a beginner for beginners', http://www.codingexplorer.com/, (accessed Jan 2015)