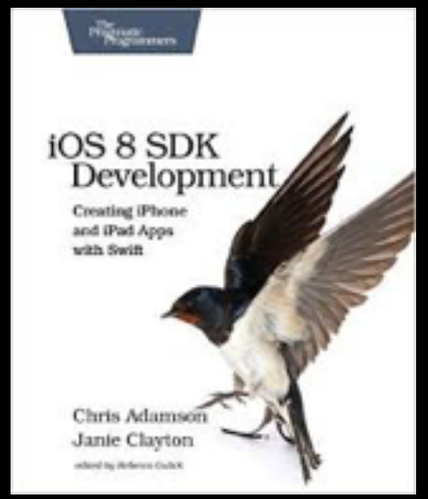
Swift

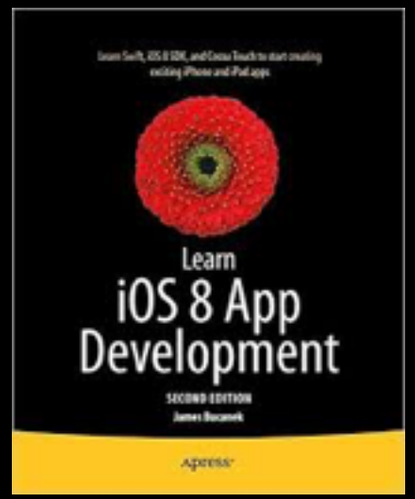
« Swift est un nouveau langage de programmation à la fois puissant et intuitif, créé par Apple pour l'élaboration d'apps iOS et Mac. (…) Swift est un langage facile à apprendre et à utiliser, même si vous n'avez jamais codé. (…) »

Références

Ouvrages :

- Learn iOS 8 App Development, Authors: Bucanek, James
- iOS 8 SDK Development, Creating iPhone and iPad Apps with Swift, by Chris Adamson and Janie Clayton





Webographie : La documentation Apple concernant swift est vraiment très bien faite, le cours s'en inspire que que beaucoup, certaines passages sont identiques à la documentation : https://developer.apple.com et swift.org

www.weheartswift.com/

https://www.raywenderlich.com/

swiftyeti.com/generics/

appventure.me/

https://learnxinyminutes.com/docs/swift/

stackoverflow.com/

http://codewithchris.com/learn-swift-from-objective-c/

dean.cafelembas.com/

Prologue

Swift

- Swift a été développé afin de faciliter la réalisation des applications. En effet, Swift est beaucoup plus rapide, plus concis, plus simple que l'Objective-C.

- Possible de concilier les 2 langages au sein d’une même application.

- Avec Swift, vous écrirez toutefois moins de lignes de codes, vous utiliserez moins de fichiers et vous disposerez d'une syntaxe facile à comprendre. Sa syntaxe a été conçue de façon à ce qu'un débutant puisse rapidement prendre ses marques.

Historique

- Père de Swift : ***Chris Lattner***, (Apple depuis 2005) où il commence le ddv de Swift mi 2010.
- S'inspirer de Objective-C, Haskell, Ruby, Python ou C#, Scala, …
- La première version publique de ce langage a été présentée le 2 juin 2014 lors de la WWDC.
- Le 8 juin 2015, Apple annonce que le code du compilateur et des bibliothèques standard seront distribués sous une licence open source12.
- Apple affirme que des contributions de la communauté seront acceptées et encouragées13.
- Depuis le 3 novembre 2015, Swift version 2.2 est désormais disponible sous licence Apache 2.0. Le code ouvert inclut le compilateur et les bibliothèques standard du langage.
- Swift 3.0 : la communauté open-source a son site swift.org et Apple ajoute le support officiel de Linux !

Evolution d’Objective-C

- De nombreuses fonctionnalités ont été ajoutées au fil des ans
- sub-scripting
- syntaxe par block
- ARC
- dot-syntax
- the move away from “id”
- literals
- and so on

Influences from other languages

- **Javascript** -> closure syntax, ‘function’ keyword, identity operators, var, etc.
- **Python** -> ranges, object-oriented & functional elements, inferred static typing
- **Ruby** -> string interpolation, optional binding, implicit returns
- **Java** -> generics are quite similar,
- **Rust** -> null pointer impossible,+ safety and correctness
- **Haskell, C# (http://swiftcomparsion.qiniudn.com), CLU**, Ceylon (operator ->optional) and **SCALA** !!! (https://leverich.github.io/swiftislikescala/)
- **Groovy** -> lists and maps, closures, named parameters, lazy instantiation, safe navigation operator,
- **Objective-C** -> Cocoa, LLVM, emphasis on enumerations, ARC, categories, XCode is actually great.

Introduction

Swift et Xcode

Swift et Xcode

- 2 façons de coder :
  - avec un projet classique : Xcode Project
  - avec un playground : Xcode playground, permet de tester en temps réel son code
  - Sous linux : https://swift.org/download/#latest-development-snapshots

Swift et Xcode


```
//: Playground - noun: a place where people can play
import UIKit
var str = "Hello, playground"
```

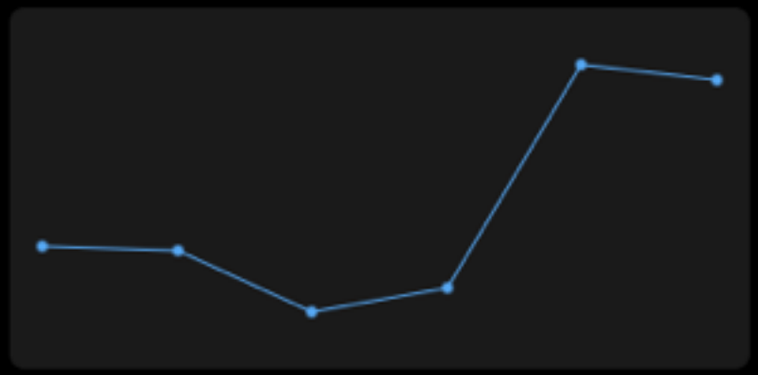
Playground

If the initial value doesn't provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

```
let implicitInteger = 70                    70
let implicitDouble = 70.0                   70
let explicitDouble: Double = 70             70
```

Playground

```
numbers.map({
    (number: Int) -> Int in
    let result = (number % 2)
    number
```



```
    return result
```

```
})
```

[0, 1, 1, 0, 0, 1]

(6 times)

(6 times)

(6 times)

Playground

MyPlayground.playgroundEdited

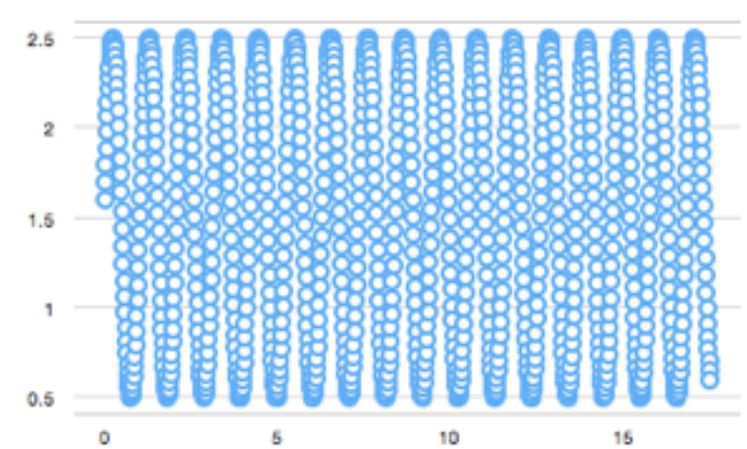
MyPlayground.playground 〉 init(size:)

```
// Playground - noun: a place where people can play
// Think as below as your Main class, basically the Stage

// this imports higher level APIs like Starling
import SpriteKit
import XCPlayground

// our main logic inside this class
class GameScene: SKScene {

    // properties initialization
    // note that the spriteNode property below is not initialized
    // we initialize it through the init initializer below
    var spriteNode: SKSpriteNode
    var i = 0.0

    // this is our initializer, called once when the scene is created
    // we do our initialization/setup here
    init(size: CGSize){

        // let's grab an image, like [Embed] in AS3, results in image data like
            BitmapData
        // let is to declare a constant, var a variable
        // note that we don't type things, you actually can to resolve
            ambiguity sometimes
        // but it is inferred by default and does not cause performance issues
            to not statically type
        let sprite = NSImage(contentsOfFile:"/Users/timbert/Documents/
            Adium.png")

        // let's create a bitmap, like Bitmap in AS3
        let myTexture = SKTexture(image: sprite)

        // let's wrap it inside a node
        spriteNode = SKSpriteNode(texture: myTexture)

        // we position it, we could scale it, etc.
        spriteNode.position = CGPoint (x: 250, y: 250)

        // we complete the initialization by initializating the superclass
        super.init(size: size)
    }

    // this gets triggered automtically when the scene is presented by the view
    // similar to Event.ADDED_TO_STAGE
    override func didMoveToView(view: SKView) {

        // let's add it to the display list
        self.addChild(spriteNode)
    }

    // we override update, which is like an Event.ENTER_FRAME or advanceTime in
        Starling
    override func update(currentTime: CFTimeInterval) {
        i += 0.1
        // oscillation with sin, like Math.sin
```

{GameScene spriteNode SKSpriteNode i 0.0}

Timeline 〉 MyPlayground.playground (Timeline)

var osc = 1.5 + sin(CDouble(i))



40 sec

Style guide

- https://github.com/raywenderlich/swift-style-guide

```
AppDelegate
  window
  application(_:didFinishLaunchingWithOptions:)
  applicationWillResignActive(_:)
  applicationDidEnterBackground(_:)
  applicationWillEnterForeground(_:)
  applicationDidBecomeActive(_:)
  applicationWillTerminate(_:)
```

Class Prefixes

Swift types are automatically namespaced by the module that contains them and you should not add a class prefix such as RW. If two names from different modules collide you can disambiguate by prefixing the type name with the module name. However, only specify the module name when there is possibility for confusion which should be rare.

```
import SomeModule

let myClass = MyModule.UsefulClass()
```

Delegates

When creating custom delegate methods, an unnamed first parameter should be the delegate source. (UIKit contains numerous examples of this.)

Preferred:

```
func namePickerView(_ namePickerView: NamePickerView, didSelectName name: String)
func namePickerViewShouldReload(_ namePickerView: NamePickerView) -> Bool
```

Not Preferred:

```
func didSelectName(namePicker: NamePickerViewController, name: String)
func namePickerShouldReload() -> Bool
```

Use Type Inferred Context

Use compiler inferred context to write shorter, clear code. (Also see Type Inference.)

Preferred:

```
let selector = #selector(viewDidLoad)
view.backgroundColor = .red
let toView = context.view(forKey: .to)
let view = UIView(frame: .zero)
```

Not Preferred:

```
let selector = #selector(ViewController.viewDidLoad)
view.backgroundColor = UIColor.red
let toView = context.view(forKey: UITransitionContextViewKey.to)
let view = UIView(frame: CGRect.zero)
```

Generics

Generic type parameters should be descriptive, upper camel case names. When a type name doesn't have a meaningful relationship or role, use a traditional single uppercase letter such as `T`, `U`, or `V`.

Preferred:

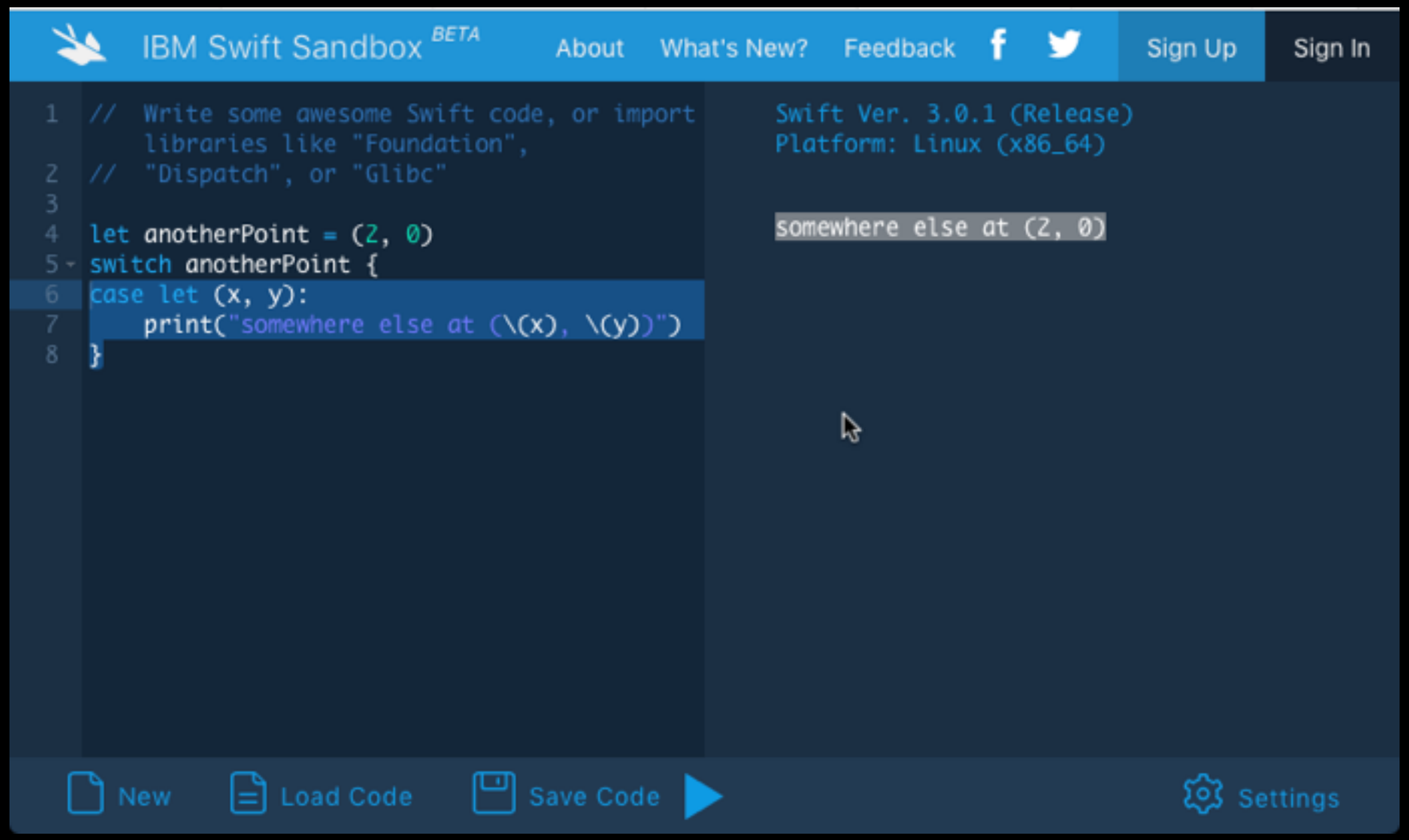
```
struct Stack<Element> { ... }
func write<Target: OutputStream>(to target: inout Target)
func swap<T>(_ a: inout T, _ b: inout T)
```

Not Preferred:

```
struct Stack<T> { ... }
func write<target: OutputStream>(to target: inout target)
func swap<Thing>(_ a: inout Thing, _ b: inout Thing)
```

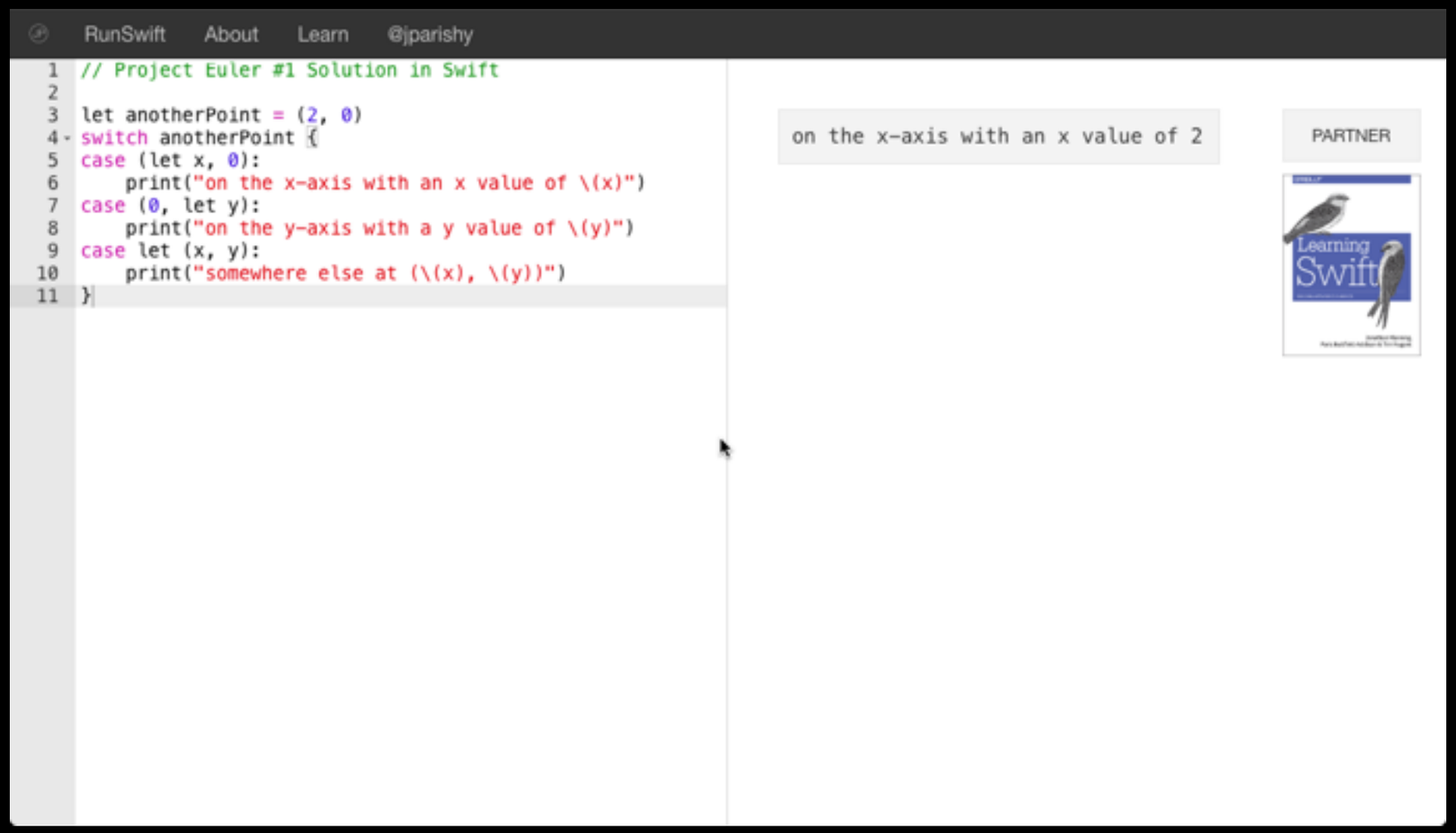
Coder du Swift sans Xcode ?

Swifter sans Xcode
(et sans IHM…)



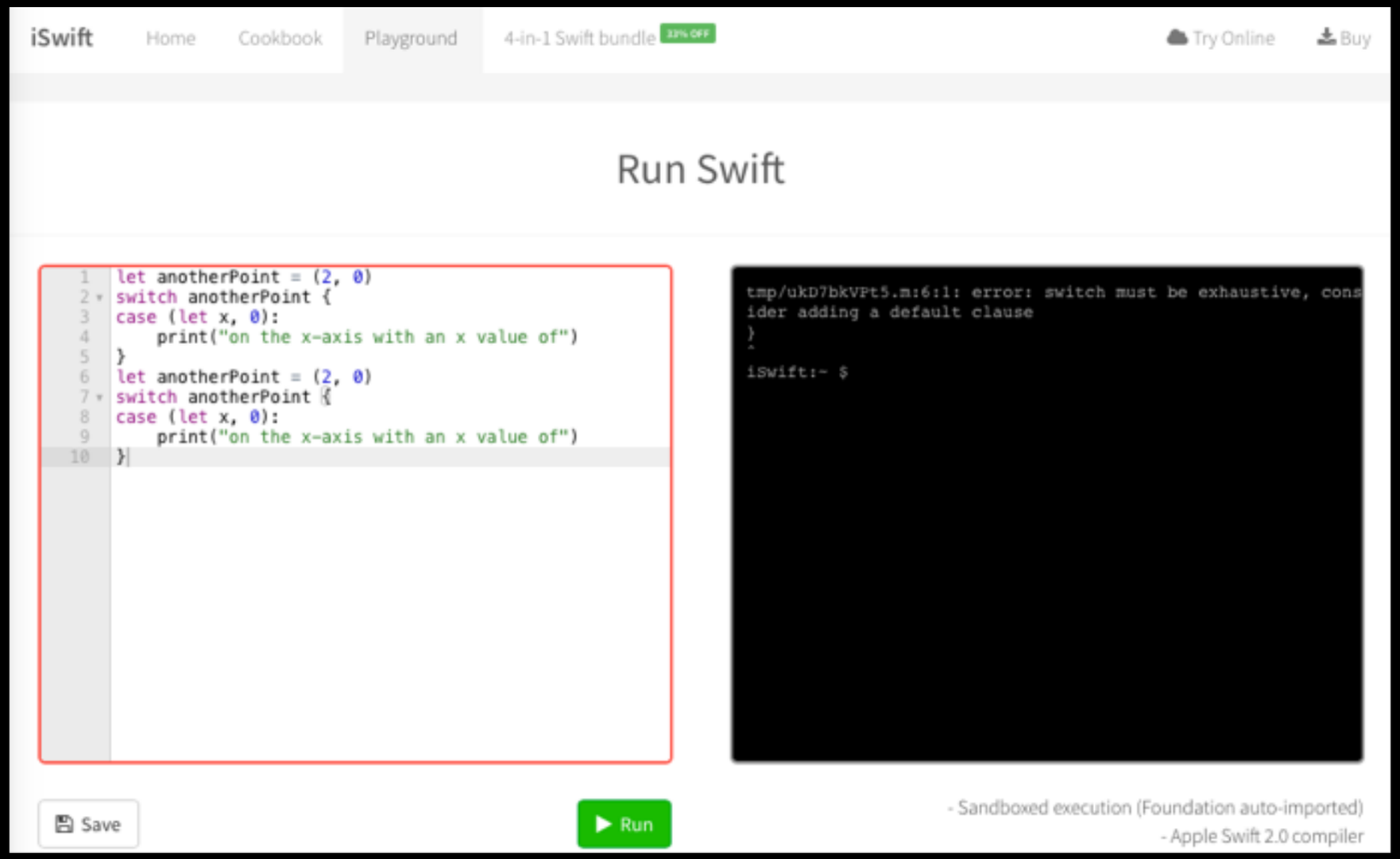
- https://swiftlang.ng.bluemix.net/#/repl (IBM) : avec import

Swifter sans Xcode



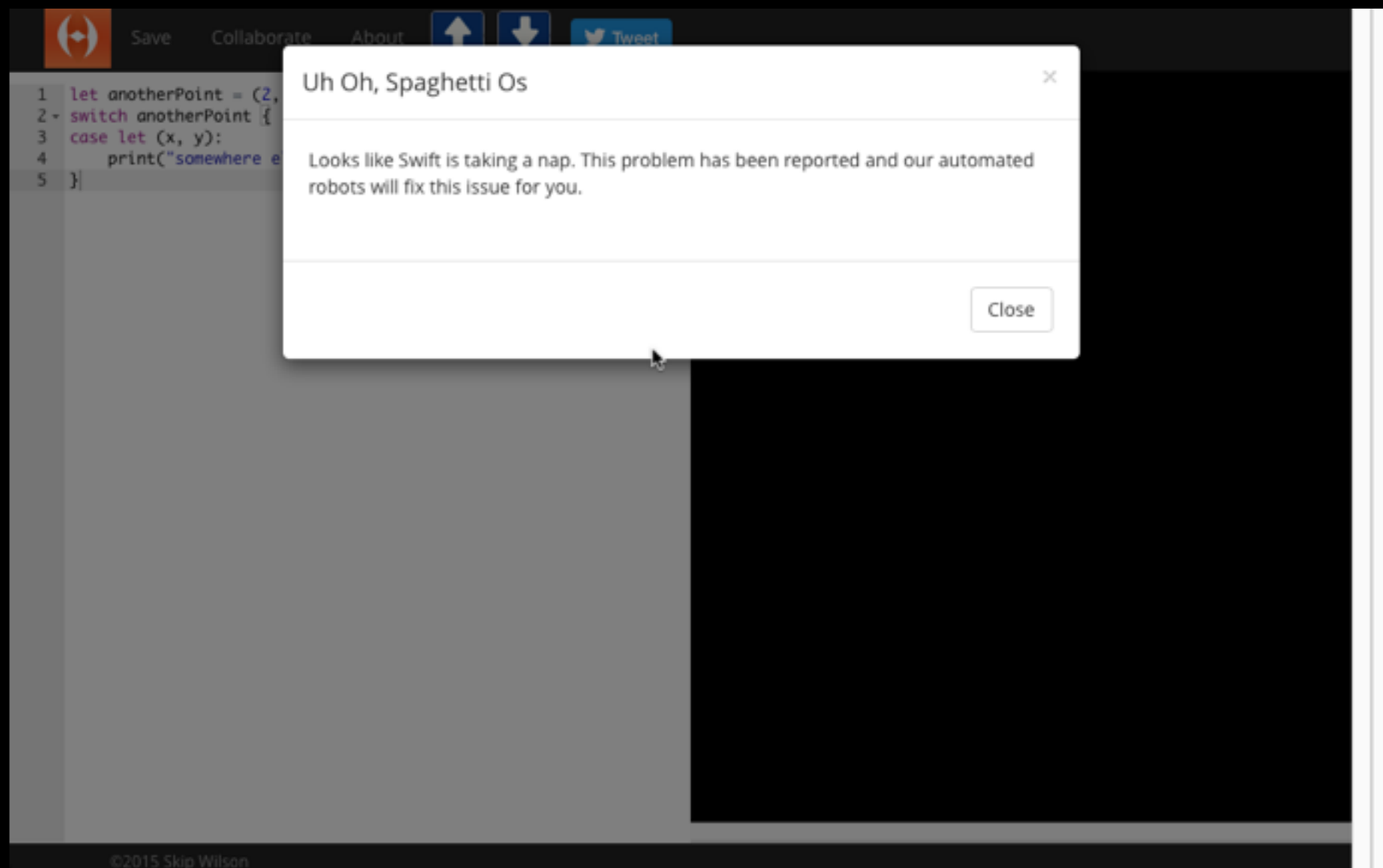
- http://www.runswiftlang.com/ (inclut Swift 3.0 !!!)

Swifter sans Xcode



- http://iswift.org/playground : idem mais Swift V2.0 seulement

Swifter sans Xcode



- http://swiftstub.com/ (Swift 2.0 et uh uh.. hier)

Part I.

Before Objects

A swift tour

Basics

Comments

```
/* This is the start of the first multiline
comment.
    /* This is the second, nested multiline
comment. */
    This is the end of the first multiline
comment. */
```

Valeurs

- **Constantes** : Valeur unique à un moment (pas forcément lors de la compilation) : utiliser pour nommer une valeur utilisée plusieurs fois

```
let myConstant = 42
```

- **Variables**

```
var myVariable = 42
myVariable = 50
```

- Les versions Mutable et Immutable des containers en Objective-C ne sont plus nécessaires

Nommage

- N'importe quel caractère unicode, sauf :
  - espace, caractère mathématiques, flèches, caractères unicode réservés ou points (:, ; , etc), lignes et contours de boites

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

- Redéclaration ou renommage impossibles

NOTE : If you need to give a constant or variable the same name as a reserved Swift keyword, surround the keyword with back ticks (`) when using it as a name. However, avoid using keywords as names unless you have absolutely no choice.

Variables

- Constantes et variables doivent avoir le même type que la valeur attribuée
- Déclarations multiples

```
var x = 0.0, y = 0.0, z = 0.0
```

Affichage des var/ctes

- print (ancien println):

```
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"

print(friendlyWelcome)
// prints "Bonjour!"

let cat = "🐱"; print(cat)
```

Types

- **Typage inféré** à partir de la valeur

```
let implicitInteger = 70
let implicitDouble = 70.0
```

- **Typage explicite** (annotation de type) si nécessaire (rarement utile)

```
let explicitDouble: Double = 70
```

Type safety
String and **Int**

```
var name: String
name = "Tim McGraw"

var age: Int
```

- Type safety

```
age = 25
name = 25
// compile-time error : error: cannot assign
value of type 'Int' to type 'String'
age = "Time McGraw"
// compile-time error : error: cannot assign
value of type 'String' to type 'Int'
```

Data types
Int

- **Int**: Type safety Integer Bounds
  You can access the minimum and maximum values of each integer type with its min and max properties:

```
let minValue = UInt8.min  // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max  // maxValue is equal to 255, and is of type UInt8
```

- In most cases, you don't need to pick a specific size of integer to use in your code. Swift provides an additional integer type, Int, which has the same size as the current platform's native word size:

- **Int** : On a 32-bit platform, Int is the same size as Int32, on a 64-bit platform, Int is the same size as Int64.
  Unless you need to work with a specific size of integer, always use Int for integer values in your code. This aids code consistency and interoperability. Even on 32-bit platforms, Int can store any value between -2,147,483,648 and 2,147,483,647, and is large enough for many integer ranges.

- The same for **UInt**…

Data types
Float and **Double**

- Float and Double

```
var latitude: Double
latitude = 36.166667

var longitude: Float
longitude = -86.783333
```

- Précision

```
longitude = -86.783333
longitude = -186.783333
longitude = -1286.783333
longitude = -12386.783333
longitude = -123486.783333
longitude = -1234586.783333
```

```
-86.78333
-186.7833
-1286.783
-12386.78
-123486.8
-1234587
```

Data types
Float and **Double**

- Précision (suite)

```
var longitude: Double
longitude = -86.783333
longitude = -186.783333
longitude = -1286.783333
longitude = -12386.783333
longitude = -123486.783333
longitude = -1234586.783333
```

```
-86.783333
-186.783333
-1286.783333
-12386.783333
-123486.783333
-1234586.783333
```

Data types
Boolean

- true or false :

```
var stayOutTooLate: Bool
stayOutTooLate = true

var nothingInBrain: Bool
nothingInBrain = true

var missABeat: Bool
missABeat = false

if missABeat {
    print("Mmm, tasty turnips!")
} else {
    print("Eww, turnips are horrible.")
}
// prints "Eww, turnips are horrible."
```

Data types
Boolean

- Swift's type safety prevents non-Boolean values from being substituted for Bool. The following example reports a compile-time error:

```
let i = 1
if i {
    // this example will not compile, and will report an
error
}
```

- However, the alternative example below is valid:

```
let i = 1
if i == 1 {
    // this example will compile successfully
}
```

Strings

- Lourd en Objective-C

```
NSMutableString string = [NSMutableString
alloc] initWithString:@"Hello"];
[string appendString:@" world"];
NSString greeting = [NSString
stringByAppendingString:@"my name is Kevin"];
```

- Swift : script-like et fun (comme langages actuels)

```
var string = "Hello" + " world"
var greeting = string + "my name is Kevin"
```

String interpolation

There’s an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (\) before the parentheses. For example:

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

Numeric Literals

- Integer literals can be written as:
  - A decimal number, with no prefix
  - A binary number, with a 0b prefix
  - An octal number, with a 0o prefix
  - A hexadecimal number, with a 0x prefix

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 17 in binary notation
let octalInteger = 0o21           // 17 in octal notation
let hexadecimalInteger = 0x11     // 17 in hexadecimal
notation
```

Numeric Literals

- For decimal numbers with an exponent of exp, the base number is multiplied by 10^{exp}:
  - 1.25e2 means 1.25×10^{2}, or 125.0.
  - 1.25e-2 means 1.25×10^{-2}, or 0.0125.
- For hexadecimal numbers with an exponent of exp, the base number is multiplied by 2exp:
  - 0xFp2 means 15×2^{2}, or 60.0.
  - 0xFp-2 means 15×2^{-2}, or 3.75.

Numeric Literals

- All of these floating-point literals have a decimal value of 12.1875:

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

- Numeric literals can contain extra formatting to make them easier to read. Both integers and floats can be padded with extra zeros and can contain underscores to help with readability. Neither type of formatting affects the underlying value of the literal:

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

Arrays

- Create **arrays** and **dictionaries** using brackets ([]), and access their elements by writing the index or key in brackets.
- A comma is allowed after the last element.

```
var shoppingList = ["catfish", "water", "tulips", "blue
paint"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
 ]
occupations["Jayne"] = "Public Relations"
```

Arrays

- To create an empty array or dictionary, use the initializer syntax.
- A comma is allowed after the last element.

```
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

Casting: Integer and Floating-Point Conversion

- Conversions between integer and floating-point numeric types must be made explicit:

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) +
pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be
of type Double
```

Casting

- Le **casting** n’est jamais automatique
- If you need to convert a value to a different type, explicitly make an instance of the desired type

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

Type Aliases

- Type aliases define an alternative name for an existing type. You define type aliases with the typealias keyword. Type aliases are useful when you want to refer to an existing type by a name that is contextually more appropriate, such as when working with data of a specific size from an external source:

```
typealias AudioSample = UInt16
```

- Once you define a type alias, you can use the alias anywhere you might use the original name:

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

- Here, AudioSample is defined as an alias for UInt16. Because it is an alias, the call to AudioSample.min actually calls UInt16.min, which provides an initial value of 0 for the maxAmplitudeFound variable.

Tuples

- Tuples group multiple values into a single compound value.
- The values within a tuple can be of any type and do not have to be of the same type as each other.

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String),
and equals (404, "Not Found")
```

Tuples

- You can decompose a tuple's contents into separate constants or variables, which you then access as usual:

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// prints "The status code is 404"
print("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

- If you only need some of the tuple's values, ignore parts of the tuple with an underscore (_) when you decompose the tuple:

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// prints "The status code is 404"
```

Tuples

- Alternatively, access the individual element values in a tuple using index numbers starting at zero:

```
print("The status code is \(http404Error.
0)")
// prints "The status code is 404"
print("The status message is \(http404Error.
1)")
// prints "The status message is Not Found"
```

Tuples

- You can name the individual elements in a tuple when the tuple is defined:

```
let http200Status = (statusCode: 200, description:
"OK")
```

- If you name the elements in a tuple, you can use the element names to access the values of those elements:

```
print("The status code is \
(http200Status.statusCode)")
// prints "The status code is 200"
print("The status message is \
(http200Status.description)")
// prints "The status message is OK"
```

Tuples*

- You can name the individual elements in a tuple when the tuple is defined:

```
let http200Status = (statusCode: 200, description: "OK")
```

- If you name the elements in a tuple, you can use the element names to access the values of those elements:

```
var result = (statusCode: 200, statusText: "OK", true)
var result2 = (statusCode: 404, "Not Found", hasBody: true)
result = result2                             // OK
print(result)
// print (.0 404, .1 "Not Found", .2 true)
result2 = result                             // OK
result = (code: 200, statusText: "OK", true) // ERROR
// print error: cannot assign value of type '(code: Int, statusText:
String, Bool)' to type '(statusCode: Int, statusText: String, Bool)'
//  result = (code: 200, statusText: "OK", true) // ERROR
//           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Important: **Tuples**

- Tuples are particularly useful as the return values of functions.
- A function that tries to retrieve a web page might return the (Int, String) tuple type to describe the success or failure of the page retrieval. By returning a tuple with two distinct values, each of a different type, the function provides more useful information about its outcome than if it could only return a single value of a single type.

Optionals

You use optionals in situations where a value may be absent. An ***optional*** says:

- There **is** a value, and it **equals x**

or

- There **isn't** a value at all!
  - NOTE: The concept of optionals doesn't exist in C or Objective-C. The nearest thing in Objective-C is the ability to return nil from a method that would otherwise return an object, with nil meaning "the absence of a valid object." However, this only works for objects—**it doesn't work for structures, basic C types, or enumeration values. For these types, Objective-C methods typically return a special value (such as NSNotFound) to indicate the absence of a value. This approach assumes that the method's caller knows there is a special value to test against and remembers to check for it. Swift's optionals let you indicate the absence of a value for any type at all, without the need for special constants**.

Optionals

- Here's an example of how optionals can be used to cope with the absence of a value.
- Swift's ***Int*** type has an initializer which tries to convert a ***String*** value into an Int value.
- However, not every string can be converted into an integer.
  - The string "123" can be converted into the numeric value 123,
  - but the string "hello, world" does not have an obvious numeric value to convert to.

Optionals

- The example below uses the initializer to try to convert a String into an Int:

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type
"Int?", or "optional Int"
```

- Because the initializer might fail, it returns an **optional Int, rather than an Int**. An optional Int is written as **Int?**, not Int. The question mark indicates that the value it contains is optional, meaning that it might contain some Int value, or it might contain no value at all. (It can't contain anything else, such as a Bool value or a String value. It's either an Int, or it's nothing at all.)

nil & optionals

- You set an optional variable to a valueless state by assigning it the special value **nil**:

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int
value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

- NOTE: **nil cannot be used with nonoptional** constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.)

nil

- If you define an optional variable without providing a default value, the variable is automatically set to **nil** for you:

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

- NOTE: **Swift's nil is not the same as nil in Objective-C**. In Objective-C, nil is a pointer to a nonexistent object. **<u>In Swift, nil is not a pointer</u>**—it is the absence of a value of a certain type. Optionals of any type can be set to nil, not just object types.)

Optional Typing

- Objective-C:

```
NSString *x = @"hello"; // great
NSString *y = nil; // fine
```

- Swift:

```
var x : String = "hello"; // great
var y : String? = nil; // ok
var z : String = nil; // COMPILER ERROR:
Nil cannot initialize specified type
‘String’
```

If Statements

- You can use an if statement to find out whether an optional contains a value by comparing the optional against nil.
- You perform this comparison with the “equal to” operator (==) or the “not equal to” operator (!=).
- If an optional has a value, it is considered to be “not equal to” nil:

```
if convertedNumber != nil {
    print("convertedNumber contains some integer
value.")
}
// prints "convertedNumber contains some integer value."
```

Forced Unwrapping Optionals

- Once you're sure that the optional does contain a value, you can access its underlying value by **adding an exclamation mark (!)** to the end of the optional's name.

- The exclamation mark effectively says, "I (the developer) know that this optional definitely has a value; please use it."

- This is known as **forced unwrapping** of the optional's value:

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \
(convertedNumber!).")
}
// prints "convertedNumber has an integer value of 123."
```

Optional Binding

- You use optional binding to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable.

- Optional binding can be used with if and while statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action.

- Write an optional binding for an if statement as follows:

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

Optional Binding

- You can rewrite the possibleNumber example from the previous Optionals slides to use optional binding rather than forced unwrapping:

```
if let actualNumber = Int(possibleNumber) {
    print("\'\(possibleNumber)\' has an integer value of \
(actualNumber)")
} else {
    print("\'\(possibleNumber)\' could not be converted to an integer")
}
```

- "If the optional Int returned by Int(possibleNumber) contains a value, set a new constant called actualNumber to the value contained in the optional."

- If the conversion is successful, the actualNumber constant becomes available for use within the first branch of the if statement.

- **It has already been initialized with the value contained within the optional, and so there is no need to use the ! suffix to access its value. In this example, actualNumber is simply used to print the result of the conversion.**

Optional Binding

- You can use both constants and variables with optional binding. If you wanted to manipulate the value of actualNumber within the first branch of the if statement, you could write `if var actualNumber` instead, and the value contained within the optional would be made available as a variable rather than a constant.

- You can include multiple optional bindings in a single if statement and use a where clause to check for a Boolean condition:

```
if let  firstNumber = Int("4"),
        secondNumber = Int("42")
   where firstNumber < secondNumber {
        print("\(firstNumber) < \(secondNumber)")
}
// prints "4 < 42"
```

Implicitly Unwrapped Optionals

- Sometimes it is clear from a program's structure that an optional will always have a value, after that value is first set.
  - In these cases, it is useful to remove the need to check and unwrap the optional's value every time it is accessed, because it can be safely assumed to have a value all of the time.
- These kinds of optionals are defined as implicitly unwrapped optionals.
  - You write an implicitly unwrapped optional by placing an exclamation mark (**String!**) rather than a question mark (**String?**) after the type that you want to make optional.

Implicitly Unwrapped Optionals

- Implicitly unwrapped optionals are useful when an optional's value is confirmed to exist immediately after the optional is first defined and can definitely be assumed to exist at every point thereafter.
  - The primary use of implicitly unwrapped optionals in Swift is during class initialization.
- An implicitly unwrapped optional is a normal optional behind the scenes, but can also be used like a nonoptional value, without the need to unwrap the optional value each time it is accessed.

Implicitly Unwrapped Optionals

- The following example shows the difference in behavior between an optional string and an implicitly unwrapped optional string when accessing their wrapped value as an explicit String:

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! //
requires an exclamation mark

let assumedString: String! = "An implicitly
unwrapped optional string."
let implicitString: String = assumedString // no
need for an exclamation mark
```

Implicitly Unwrapped Optionals

- You can think of an implicitly unwrapped optional as giving permission for the optional **to be unwrapped automatically whenever it is used**. Rather than placing an exclamation mark after the optional's name each time you use it, you place an exclamation mark after the optional's type when you declare it.

```
let forcedString: String = possibleString!
//versus
let assumedString: String! = "An implicitly unwrapped
optional string."
```

- NOTE: If an implicitly unwrapped optional is nil and you try to access its wrapped value, **you'll trigger a runtime error**. The result is exactly the same as if you place an exclamation mark after a normal optional that does not contain a value.

Implicitly Unwrapped Optionals

- You can still treat an implicitly unwrapped optional like a normal optional, to check if it contains a value:

```
if assumedString != nil {
    print(assumedString)
}
// prints "An implicitly unwrapped optional
string."
```

Implicitly Unwrapped Optionals

- You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

```
if let definiteString = assumedString {
    print(definiteString)
}
// prints "An implicitly unwrapped optional string."
```

- NOTE: **Do not use an implicitly unwrapped optional when there is a possibility of a variable becoming nil at a later point**. Always use a normal optional type if you need to check for a nil value during the lifetime of a variable.

Implicitly Unwrapped Optionals

- You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

```
if let definiteString = assumedString {
    print(definiteString)
}
// prints "An implicitly unwrapped optional string."
```

- NOTE: **Do not use an implicitly unwrapped optional when there is a possibility of a variable becoming nil at a later point**. Always use a normal optional type if you need to check for a nil value during the lifetime of a variable.

A swift tour

Operators

Arithmetic Operators

- Swift supports the four standard arithmetic operators for all number types:
  - Addition (+)
  - Subtraction (-)
  - Multiplication (*)
  - Division (/)

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

- NOTE: Unlike the arithmetic operators in C and Objective-C, the Swift arithmetic operators do not allow values to overflow by default.

Remainder Operator (modulo)

```
9 % 4     // equals 1
8 % 2.5    // equals 0.5
```

Increment and Decrement Operators

```
var i = 0
++i      // i now equals 1
var a = 0
let b = ++a
// a and b are now both equal to 1
let c = a++
// a is now equal to 2, but c has been set
to the pre-increment value of 1
```

Unary Operator

- Unary Minus Operator

```
let three = 3
let minusThree = -three
// minusThree equals -3
let plusThree = -minusThree
// plusThree equals 3, or "minus minus
three"
```

- Unary Plus Operator

```
let minusSix = -6
let alsoMinusSix = +minusSix
// alsoMinusSix equals -6
```

Ternary Conditional Operator

- The ternary conditional operator is a special operator with three parts, which takes the form **question ? answer1 : answer2**. It is a shortcut for evaluating one of two expressions based on whether question is true or false. If question is true, it evaluates answer1 and returns its value; otherwise, it evaluates answer2 and returns its value.

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

- The code above is shorthand for the one below:

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
} else {
    rowHeight = rowHeight + 20
}
// rowHeight is equal to 90
```

Nil Coalescing Operator

- The nil coalescing operator:

```
a ?? b
```

- unwraps an optional a if it contains a value, or returns a default value b if a is nil. The expression a is always of an optional type. The expression b must match the type that is stored inside a. It is shorthand for the code below:

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

- The code above uses the ternary conditional operator and forced unwrapping (**a!**) to access the value wrapped inside a when a is not nil, and to return b otherwise. The nil coalescing operator provides a more elegant way to encapsulate this conditional checking and unwrapping in a concise and readable form.

- NOTE: If the value of a is non-nil, the value of b is not evaluated. This is known as short-circuit evaluation.

Nil Coalescing Operator

- The example below uses the nil coalescing operator to choose between a default color name and an optional user-defined color name:

```
let defaultColorName = "red"
var userDefinedColorName: String?   //
defaults to nil

var colorNameToUse = userDefinedColorName ??
defaultColorName
// userDefinedColorName is nil, so
colorNameToUse is set to the default of "red"
```

Range Operators

- Swift includes two range operators, which are shortcuts for expressing a range of values.

- Closed Range Operator

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

Range Operators

- Half-Open Range Operator

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
    print("Person \(i + 1) is called \
(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

Others operators and overloading

- http://nshipster.com/swift-operators/

A swift tour

Strings

Counting Characters

- To retrieve a count of the Character values in a string, use the count property of the string's characters property:

```
let unusualMenagerie = "Koala 🐨, Snail 🐌,
Penguin 🐧, Dromedary 🐪"
print("unusualMenagerie has \
(unusualMenagerie.characters.count)
characters")
// prints "unusualMenagerie has 40
characters"
```

Accessing and Modifying a String

- Each String value has an associated index type, `String.Index`, which corresponds to the position of each Character in the string.

- As mentioned above, different characters can require different amounts of memory to store, so in order to determine which `Character` is at a particular position, you must iterate over each Unicode scalar from the start or end of that `String`. For this reason, Swift `strings` cannot be indexed by integer values.

- Use the **`startIndex`** property to access the position of the first Character of a String. The **`endIndex`** property is the position after the last character in a String. As a result, the `endIndex` property isn't a valid argument to a string's subscript. If a String is empty, `startIndex` and `endIndex` are equal.

Accessing and Modifying a String

- A `String.Index` value can access its immediately preceding index by calling the **`predecessor()`** method, and its immediately succeeding index by calling the **`successor()`** method.
- Any index in a `String` can be accessed from any other index by chaining these methods together, or by using the **`advancedBy(_:)`** method.
- Attempting to access an index outside of a string's range will trigger a runtime error.

Accessing / Modifying a String

- You can use subscript syntax to access the `Character` at a particular `String` index.

```
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
greeting[greeting.endIndex.predecessor()]
// !
greeting[greeting.startIndex.successor()]
// u
let index = greeting.startIndex.advancedBy(7)
greeting[index]
// a
```

Accessing / Modifying a String

- Attempting to access a **`Character`** at an index outside of a string's range will trigger a runtime error.

```
greeting[greeting.endIndex] // error
// The endIndex property is the position after the last
character in a String.
greeting.endIndex.successor() // error, of course!
```

- Use the indices property of the **characters property** to create a **Range** of all of the indexes used to access individual characters in a **string**.

```
for index in greeting.characters.indices {
    print("\(greeting[index]) ", terminator: "")
}
// prints "G u t e n  T a g !"
```

Inserting and Removing

- To insert a character into a string at a specified index, use the **insert(_:atIndex:)** method.

```
var welcome = "hello"
welcome.insert("!", atIndex: welcome.endIndex)
// welcome now equals "hello!"
```

- To insert the contents of another string at a specified index, use the **insertContentsOf(_:at:)** method.

```
welcome.insertContentsOf(" there".characters,
at: welcome.endIndex.predecessor())
// welcome now equals "hello there!"
```

Inserting and Removing

- To remove a character from a string at a specified index, use the **removeAtIndex(_:)** method.

```
welcome.removeAtIndex(welcome.endIndex.predece
ssor())
// welcome now equals "hello there"
```

- To remove a substring at a specified range, use the **removeRange(_:)** method:

```
let range =
welcome.endIndex.advancedBy(-6)..<welcome.endI
ndex
welcome.removeRange(range)
// welcome now equals "hello"
```

Comparing Strings

- String and Character Equality

```
let quotation = "We're a lot alike, you and
I."
let sameQuotation = "We're a lot alike, you
and I."
if quotation == sameQuotation {
    print("These two strings are considered
equal")
}
// prints "These two strings are considered
equal"
```

Comparing Strings

- Two String values (or two Character values) are considered equal if their extended grapheme clusters are canonically equivalent. **Extended grapheme clusters are canonically equivalent if they have the same linguistic meaning and appearance, even if they are composed from different Unicode scalars behind the scenes**.

- For example, LATIN SMALL LETTER E WITH ACUTE (U+00E9) : '**é**' is canonically equivalent to LATIN SMALL LETTER E (U+0065) followed by COMBINING ACUTE ACCENT (U+0301). Both of these extended grapheme clusters are valid ways to represent the character é, and so they are considered to be canonically equivalent:

```
// "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"

// "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE
ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"

if eAcuteQuestion == combinedEAcuteQuestion {
    print("These two strings are considered equal")
}
// prints "These two strings are considered equal"
```

Comparing Strings hasPrefix(_:)

- You can use the hasPrefix(_:) method with the romeoAndJuliet array to count the number of scenes in Act 1 of the play:

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        ++act1SceneCount
    }
}
print("There are \(act1SceneCount) scenes in Act 1")
// prints "There are 5 scenes in Act 1 »
```

Comparing Strings hasSuffix(_:)

- Similarly, use the hasSuffix(_:) method to count the number of scenes that take place in or around Capulet's mansion and Friar Lawrence's cell:

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

A swift tour

Control Flows

Control Flows

- Conditionnelless : **if** et **switch**
- **for-in**, **for**, **while**, et **repeat-while** pour des boucles.
- Parenthèses autour des conditions et boucles sont optionnelles.
- Accolades obligatoires.

Boucles : **for-in**

- The for-in loop performs a set of statements for each item in a sequence.

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

for-in

- Variable muette : _

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \
(answer)")
// prints "3 to the power of 10 is 59049"
```

for-in

- Itérer sur des items

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

for-in

- Itérer sur un dictionnaire

```
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
```

Boucles : **for**

- **for** classique du C
- forme standard

```
for initialization; condition; increment {
    statements
}
```

- exemple

```
for var index = 0; index < 3; ++index {
    print("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

for

- Portée locale de la variable d’itération

```
for var index = 0; index < 3; ++index {
    print("index is \(index)")
}
print(index)
```

Use of unresolved identifier 'index'

Boucles : **while**

- While

```
while square < finalSquare {
    //statements
}
```

- repeat while

```
repeat {
    //statements
} while square < finalSquare
```

Conditionnelles : **if**

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a
scarf.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// prints "It's not that cold. Wear a t-shirt."
```

if

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a
scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear
sunscreen.")
} else // optional {
    print("It's not that cold. Wear a t-shirt.")
}
// prints "It's really warm. Don't forget to wear
sunscreen."
```

if

- Le else étant optionnel, il peut disparaitre :

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a
scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear
sunscreen.")
}
```

Conditionnelles : **switch**

- sur n'importe quel **type** de base et ***enums***
- Sécuriser les cas (versus le if) :

```
switch some value to consider {
    case value 1:
    respond to value 1
    case value 2,
    value 3:
    respond to value 2 or 3
    default://not optional!!!
    otherwise, do something else
}
```

switch

- Sécuriser les cas - bis :

```
let someCharacter: Character = "a"
switch someCharacter {
    case "a": // empty case
    case "A":
        print("The letter A")
    default:
        print("Not the letter A")
    }
    // empty case will report a compile-time error
```

switch

- Valeurs multiples du case :

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l",
"m",
"n", "p", "q", "r", "s", "t", "v", "w", "x", "y",
"z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a
consonant")
}
// prints "e is a vowel"
```

switch

- Valeurs multiples du case (exemple 2):

```
let someCharacter: Character = "e"
switch someCharacter {
case "a","b","c","d","e","f":
    print("\(someCharacter) : smallcap hexa
character")
case "A","B","C","D","E","F":
    print("\(someCharacter) : bigcap hexa
character")
default:
    print("\(someCharacter) : another char.")
}
// prints "e : smallcap hexa character"
```

switch
pattern matching

- **Intervalles** :

```
let anotherCharacter: Character = "b"
switch anotherCharacter {
case "a"..."f":
    print("\(someCharacter) : smallcap hexa
character")
case "A"..."F":
    print("\(someCharacter) : bigcap hexa
character")
default:
    print("\(someCharacter) : another char.")
}
// prints "e : smallcap hexa character"
```

switch

- Intervalles (exemple 2) :

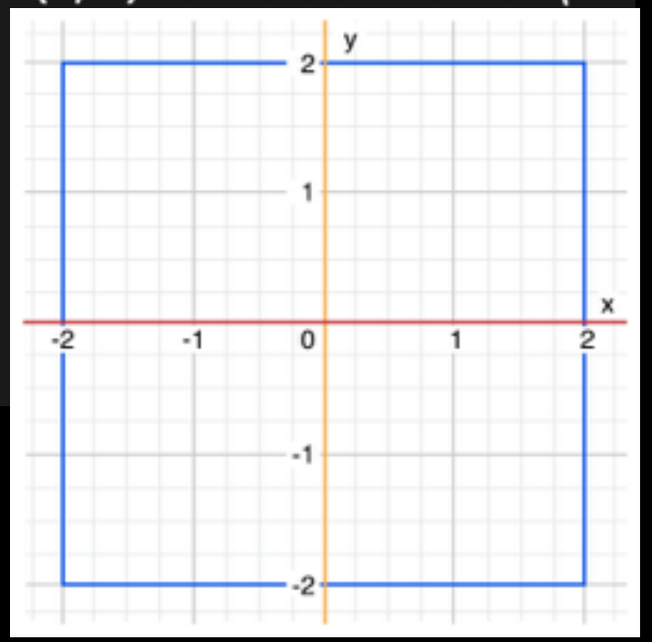
```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
var naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
// prints "There are dozens of moons orbiting Saturn."
```

switch

- Tuples :

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    print("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
    print("(\(somePoint.0), \(somePoint.1)) is
        inside the box")
default:
    print("(\(somePoint.0), \(somePoint.1)) is
        outside of the box")
}
// prints "(1, 1) is inside the box"
```

```
(.0 1, .1 1)
"(1, 1) is inside the box\n"
```



switch

- Multiple matching : **Attention** : les suivants sont ignorés

```
let somePoint = (0, 0)
switch somePoint {
case (_, 0):
    print("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    print("(0, \(somePoint.1)) is on the y-axis")
case (0, 0):
    print("\(somePoint) is at the origin")
case (-2...2, -2...2):
    print("(\(somePoint.0), \(somePoint.1)) is inside the box")
default:
    print("(\(somePoint.0), \(somePoint.1)) is outside of the
box")
}
// prints "(0, 0) is on the x-axis"
print(somePoint)
```

switch

- Value bindings : constantes ou variables

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
// prints "on the x-axis with an x value of 2"
```

switch
must be exhaustive!

- Value bindings : constantes ou variables

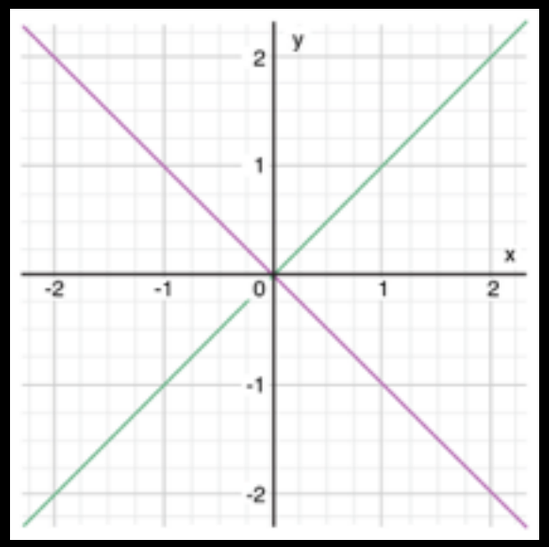
```
let anotherPoint = (2, 0)
switch anotherPoint {
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
// prints "somewhere else at (2, 0) » (OK!)
```

switch
must be exhaustive!

- Value bindings : constantes ou variables

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value
of")
}
//10:1: error: switch must be exhaustive,
consider adding a default clause
//}
//^
// (KO!)
```

switch



- **Where** clause : condition supplémentaire

- *le point est-il sur une diagonale ?*

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary
point")
}
// prints "(1, -1) is on the line x == -y"
```

switch

- Il est possible de surcharger le pattern matching (opérateur ~=)

```
// Example: overload the ~= operator to match a
string with an integer.
func ~=(pattern: String, value: Int) -> Bool {
    return pattern == "\(value)"
}
switch point {
case ("0", "0"):
    print("(0, 0) is at the origin.")
default:
    print("The point is at (\(point.0), \(point.
1)).")
}
// Prints "The point is at (1, 2)."
```

Control Transfer Statements

- `continue (only inside a loop)`
- `break (sort du bloc ou du switch)`
- `fallthrough (only inside a switch )`
- `return`
- `guard`
- `throw`

continue

- break : termine le bloc courant

```
for i in 1..<10 {
    print("i = \(i)")
    for j in 1..<10 {
        print("j = \(j)")
        for k in 1..<10 {
            print("k = \(k)")
            break
        }
        break
    }
    break
}
```

continue

- continue : termine itération ou case courant

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput.characters {
    switch character {
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
        puzzleOutput.append(character)
    }
}
print(puzzleOutput)
// prints "grtmndsthnklk"
```

break

- termine une boucle ou un switch :

```
let numberSymbol: Character = "三"  // Simplified Chinese for the number 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("An integer value could not be found for \(numberSymbol).")
}
// prints "The integer value of 三 is 3."
```

break

- termine une boucle ou un switch :

```
let numberSymbol: Character = "三"  // Simplified Chinese for the number 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    break
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("An integer value could not be found for \(numberSymbol).")
}
// /swift-execution/Sources/main.swift:15:5: warning: code after 'break' will never be executed
    possibleIntegerValue = 3
    ^
// prints "An integer value could not be found for 三."
```

fallthrough

- within a switch (to keep the C behavior):

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// prints "The number 5 is a prime number, and also an integer."
```

fallthrough

- within a switch (to keep the C behavior):

```
let yetAnotherPoint = (0, 1)
switch yetAnotherPoint {
case (0, 1):
    print("(\(0), \(1)) is just some arbitrary point")
    fallthrough
case (_, 1):
    print("(\(1), \(1)) is just some arbitrary point")
    fallthrough
default:
    print("default")
}
/* prints :
 (0, 1) is just some arbitrary point
 (1, 1) is just some arbitrary point
 default
 */
```

blocs à étiquettes

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // diceRoll will move us to the final square, so the
game is over
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // diceRoll will move us beyond the final square, so
roll again
        continue gameLoop
    default:
        // this is a valid move, so find out its effect
        square += diceRoll
        square += board[square]
    }
}
print("Game over!")
```

Early Exit: Guards

- Permet la sortie anticipée sur condition

```
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }

    print("Hello \(name)!")

    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }

    print("I hope the weather is nice in \(location).")
}

greet(["name": "John"])
// prints "Hello John!"
// prints "I hope the weather is nice near you."
greet(["name": "Jane", "location": "Cupertino"])
// prints "Hello Jane!"
// prints "I hope the weather is nice in Cupertino."
```

A swift tour

Functions

Defining and Calling Functions

- The definition describes what the function does, what it expects to receive, and what it returns when it is done. The definition makes it easy for the function to be called unambiguously from elsewhere in your code:

```
print(sayHello("Anna"))
// prints "Hello, Anna!"
print(sayHello("Brian"))
// prints "Hello, Brian!"
```

Defining and Calling Functions

- To simplify the body of this function, combine the message creation and the return statement into one line:

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

Functions Without Parameters

- Functions are not required to define input parameters. Here's a function with no input parameters, which always returns the same String message whenever it is called:

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// prints "hello, world"
```

Functions With Multiple Parameters

- Functions can have multiple input parameters, which are written within the function's parentheses, separated by commas.
- This function takes a person's name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

```
func sayHello(personName: String, alreadyGreeted: Bool) -> String
{
    if alreadyGreeted {
        return sayHelloAgain(personName)
    } else {
        return sayHello(personName)
    }
}
print(sayHello("Tim", alreadyGreeted: true))
// prints "Hello again, Tim!"
```

Functions Without Return Values

- Functions are not required to define a return type. Here's a version of the **sayHello(_:)** function, called **sayGoodbye(_:)**, which prints its own String value rather than returning it:

```
func sayGoodbye(personName: String) {
    print("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, Dave!"
```

Functions with Multiple Return Values

- You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.
- The example below defines a function called **minMax(_:)**, which finds the smallest and largest numbers in an array of Int values:

```
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)
}
```

Functions with Multiple Return Values

- Because the tuple's member values are named as part of the function's return type, they can be accessed with dot syntax to retrieve the minimum and maximum found values:

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

Optional Tuple Return Types

- If the tuple type to be returned from a function has the potential to have “no value” for the entire tuple, you can use an optional tuple return type to reflect the fact that the entire tuple can be **nil**.
  - You write an optional tuple return type by placing a question mark after the tuple type’s closing parenthesis, such as **`(Int, Int)?`** or **`(String, Int, Bool)?`**.
- NOTE: An optional tuple type such as **`(Int, Int)?`** is different from a tuple that contains optional types such as **`(Int?, Int?)`**. With an optional tuple type, the entire tuple is optional, not just each individual value within the tuple.

Optional Tuple Return Types

- The previous **`minMax(_:)`** function returns a tuple containing two **`Int`** values. However, the function does not perform any safety checks on the array it is passed.
  - If the array argument contains an empty array, the **`minMax(_:)`** function, as defined above, will trigger a ***runtime error*** when attempting to access **`array[0]`**.

Optional Tuple Return Types

- To handle this “empty array” scenario safely, write the **`minMax(_:)`** function with an optional tuple return type and return a value of nil when the array is empty:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    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)
}
```

Optional Tuple Return Types

- You can use optional binding to check whether this version of the minMax(_:) function returns an actual tuple value or nil:

```
if let bounds = minMax([8, -6, 2, 109, 3,
71]) {
    print("min is \(bounds.min) and max is \
(bounds.max)")
}
// prints "min is -6 and max is 109"
```

Function Parameter Names

- Function parameters have both an external parameter name and a local parameter name. An external parameter name is used to label arguments passed to a function call.

- A **local (intern) parameter** name is used in the implementation of the function:

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

Specifying External Parameter Names

- You write an external parameter name before the local parameter name it supports, separated by a space:

```
func someFunction(externalParameterName
localParameterName: Int) {
    // function body goes here, and can use
localParameterName
    // to refer to the argument value for
that parameter
}
```

Specifying External Parameter Names

- Here's a version of the sayHello(_:) function that takes the names of two people and returns a greeting for both of them:

```
func sayHello(to person: String, and anotherPerson:
String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}
print(sayHello(to: "Bill", and: "Ted"))
// prints "Hello Bill and Ted!"
```

- Come from ObjectiveC method call notation:

```
- (void) sayHelloTo: (NSString *)person
                 and: (NSString *)anotherPerson;
```

Omitting External Parameter Names

- If you do not want to use an external name for the second or subsequent parameters of a function, write an underscore (_) instead of an explicit external name for that parameter:

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

Exemples*

```
func sayGoodbye(localPersonName: String) {
    print("Goodbye, \(localPersonName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, Dave!"

func sayGoodbye2(externPersonName
localPersonName: String) {
    print("Goodbye, \(localPersonName)!")
}
sayGoodbye2(externPersonName: "David")
// prints "Goodbye, David!"
```

Exemples*

```
// Attention : illogique ? :
func sayGoodbye3(localPersonName : String,
localPersonSurName : String) {
    print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye3("Dave", localPersonSurName:
"Stewart")
// prints "Goodbye, Dave, Stewart!"
// => en l'absence de param externe, on doit
utiliser l'interne (sauf pour le premier)
```

Exemples*

```
func sayGoodbye3_1(localPersonName : String,
localPersonSurName : String) {
    print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye3_1("Dave", "Stewart")
// prints "error: incorrect argument label in
call (have '_:localPersonSurName:', expected
'_:externPersonSurName:')
//  sayGoodbye4("Dave", localPersonSurName:
"Stewart")
//              ^        ~~~~~~~~~~~~~~~~~~~~
//                       externPersonSurName"
```

Exemples*

```
// Mais
func sayGoodbye4(localPersonName : String,
    externPersonSurName localPersonSurName : String)
{
        print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye4("Dave", localPersonSurName: "Stewart")
// you get "error: incorrect argument label in call
(have '_:localPersonSurName:', expected
'_:externPersonSurName:')
// sayGoodbye4("Dave", localPersonSurName: "Stewart")
//             ^       ~~~~~~~~~~~~~~~~~~~~
//                     externPersonSurName
"
```

Exemples*

```
// la solution logique
func sayGoodbye5(localPersonName : String,
                externPersonSurName
localPersonSurName : String) {
    print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye5("Dave", externPersonSurName:
"Stewart")
// prints "Goodbye, Dave, Stewart!"
```

Exemples*

```
// feignant !
func sayGoodbye6(localPersonName : String,
    _ localPersonSurName : String) {
        print("Goodbye, \(localPersonName),
\(localPersonSurName)!")
}
sayGoodbye6("Dave", "Stewart")
// prints "Goodbye, Dave, Stewart!"
```

L’explication*
External vs Local Names

- Careful! The confusion comes from the fact that the first parameter it treated differently from others.

- Default Rules (most of the time you do not need to think too much about external names (at least since Swift 2.1)):

  - The external names for all **except the first parameter** **<u>default to the local name</u>**

  - You omit the **name of the first parameter** when calling the function/method (since it does not by default have an external name)

- This gives us the default Swift behaviour here illustrated with a function that takes three parameters:

```
func logMessage(message: String, prefix: String, suffix: String) {
    print("\(prefix)-\(message)-\(suffix)")
}
logMessage("error message", prefix: ">>>", suffix: "<<<")
// ">>>-error message-<<<"
```

L'explication (2)* Using External Names

- If you want to use an external name for the first parameter or use something other than the local name for the other parameters you can override the **defaults**, preceding the local name with an external name.

```
func logMessage(message message:String,
    withPrefix prefix: String,
    andSuffix suffix: String) {
        print("\(prefix)-\(message)-\(suffix)")
}
```

- Our three parameters now have these external and local names:
  - external: message, local: message
  - external: withPrefix, local: prefix
  - external: andSuffix, local: suffix
- Note that unlike local names external names do not need to be unique (though it is probably a good idea).

Default Parameter Values

- You can define a default value for any parameter in a function by assigning a value to the parameter after that parameter's type. If a default value is defined, you can omit that parameter when calling the function.

```
func someFunction(parameterWithDefault: Int = 12) {
    // function body goes here
    // if no arguments are passed to the function
call,
    // value of parameterWithDefault is 12
}
someFunction(6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12
```

Variadic Parameters

- A variadic parameter accepts zero or more values of a specified type.
  - You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called.
  - Write variadic parameters by inserting three period characters (...) after the parameter's type name:

```
func arithmeticMean(numbers: Double...)
```

- The values passed to a variadic parameter are made available within the function's body **<u>as an array</u>** of the appropriate type.
  - For example, a variadic parameter with a name of numbers and a type of Double... is made available within the function's body as a constant array called numbers of type **[Double]**.

Variadic Parameters

- The example below calculates the arithmetic mean (also known as the average) for a list of numbers of any length:

```
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
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three
numbers
```

- NOTE: A function may have at most one variadic parameter.

In-Out Parameters

- If you really need/want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an ***in-out*** parameter instead.
- You write an **in-out** parameter by placing the **`inout`** keyword at the start of its parameter definition.
  - An in-out parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value.
- You can only pass a variable as the argument for an in-out parameter.
  - You cannot pass a constant or a literal value as the argument, because constants and literals cannot be modified.
- You place an **ampersand (&) directly before a variable's name when you pass it as an argument to an in-out parameter**, to indicate that it can be modified by the function.
- *NOTE: In-out parameters cannot have default values, and variadic parameters cannot be marked as inout. If you mark a parameter as inout, it cannot also be marked as var or let.*

In-Out Parameters

- Here's an example of a function called **swapTwoInts(_:_:)**, which has two in-out integer parameters called a and b:

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

- Note that the names of someInt and anotherInt are prefixed with an ampersand when they are passed to the **swapTwoInts(_:_:)** function:

```
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"
```

- NOTE: In-out parameters are not the same as returning a value from a function. The swapTwoInts example above does not define a return type or return a value, but it still modifies the values of someInt and anotherInt. In-out parameters are an alternative way for a function to have an effect outside of the scope of its function body.

In-Out Parameters

- Bien plus explicite que manipuler des pointeurs en C
- Notez la généricité de type : <T>

```
var x,y: String

(x,y) = ("Some var", "Who cares!")
print((x))

func swapValues<T>(inout a: T,
    inout b: T) {
        let tempA = a
        a = b
        b = tempA
}
swapValues(&x,b:&y)
print((x))
print((y))
```

Function Types

- Function Types as Parameter Types
- Function Types as Return Types

Constant and Variable Parameters

- Function parameters are constants by default.
  - Trying to change the value of a function parameter from within the body of that function results in a compile-time error.
  - **This means that you can't change the value of a parameter by mistake**.
- However, sometimes it is useful for a function to have a **variable <u>copy</u>** of a parameter's value to work with.
  - You can avoid defining a new variable yourself within the function by specifying one or more parameters **<u>as variable parameters instead</u>**.
  - Variable parameters are available as variables rather than as constants, and give **<u>a new modifiable copy</u>** of the parameter's value for your function to work with.

Constant and Variable Parameters

- Define variable parameters by prefixing the parameter name with the var keyword:

```
func alignRight(var string: String, totalLength: Int, pad: Character) ->
String {
    let amountToPad = totalLength - string.characters.count
    if amountToPad < 1 {
        return string
    }
    let padString = String(pad)
    for _ in 1...amountToPad {
        string = padString + string
    }
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, totalLength: 10, pad: "-")
// paddedString is equal to "-----hello"
// originalString is still equal to « hello"!!!
```

- Conclusion :The changes you make to a variable parameter do not persist beyond the end of each call to the function, and are not visible outside the function's body. The variable parameter only exists for the lifetime of that function call.

Nested Functions

Functions can be nested.

Nested Functions*

- Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen() // 15
```

Nested Functions

- Functions are a first-class type: a function can return another function as its value.

```
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7) // 8
```

Nested Functions

- A function can take another function as one of its arguments.

```
func hasAnyMatches(list: [Int], condition: (Int) ->
Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, condition: lessThanTen) // true
```

A swift tour

Dictionaries*

Declare Dictionaries

- To declare a dictionary you can use the square brackets syntax([KeyType:ValueType]).

```
var dictionary: [String:Int]
```

- You can initialize a dictionary with a dictionary literal. **A dictionary literal is a list of key-value pairs, separated by commas, surrounded by a pair of square brackets**. A key-value pair is a combination of a key and a value separate by a colon(:):

```
var dictionary: [String:Int] = [
    "one" : 1,
    "two" : 2,
    "three" : 3
]
```

- Keep in mind that you can create empty dictionary using the empty dictionary literal ([:]).

```
var emptyDictionary: [String:Int] = [:]
```

Getting values

```
var stringsAsInts: [String:Int] = ["zero" : 0, "one" : 1,"two" : 2,
"three" : 3, "four" : 4, "five" : 5, "six" : 6, "seven" : 7,
"eight" : 8,    "nine" : 9]
```

- You can access specific elements from a dictionary using the ***subscript syntax***.
- To do this pass the key of the value you want to retrieve within square brackets immediately after the name of the dictionary.
- Because it's possible not to have a value associated with the provided key the subscript will **return an optional** value of the value type.

```
print(stringsAsInts["zero"])
// print "Optional(0)"
print(stringsAsInts["three"])
// print "Optional(3)"
print(stringsAsInts["ten"])
// print "nil"
```

Getting values

- To unwrap the value returned by the subscript you can do one of two things: use **optional binding** or **force the value** if you know for sure it exists.

```
// Unwraping the optional using optional
binding
if let twoAsInt = stringsAsInts["two"] {
    print(twoAsInt) // 2
}

// Unwraping the optional using the forced
value operator (!)
stringsAsInts["one"]! // 1
```

Getting values

- To get all the values from a dictionary you can use the ***for-in syntax***.

- It's similar to the array for in syntax with the exception that instead of getting only the value in each step you also get the key associated with that value inside of a tuple.

```
var userInfo: [String: String] = [
    "first_name" : "Tim",
    "last_name" : "Cooks",
    "job_title" : "Apple's CEO"
]

for (key, value) in userInfo {
    print("\(key): \(value)")
}
```

- To get the number of elements (key-value pairs) in a dictionary you can use the count property.

```
print(userInfo.count) // 3
```

Updating values

```
var stringsAsInts: [String:Int] = ["zero" : 0,"one" : 1,"two" :
2]
```

- The simplest way to **add a value** to a dictionary is by using the subscript syntax:

```
stringsAsInts["three"] = 3
```

- Using the subscript syntax you can change a the value associated with a key:

```
stringsAsInts["three"] = 10
```

- You can use the updateValue(forKey:) method to update the value associated with a key, if there was no value for that key it will be added. The method will return the old value wrapped in an optional or nil if there was no value before.

```
print(userInfo.count) // 3
```

Updating values

```
var stringsAsInts: [String:Int] = ["zero" : 0,"one" : 1,"two" :
2]
```

- You can use the updateValue(forKey:) method to update the value associated with a key, if there was no value for that key it will be added.

- The method will return the old value wrapped in an optional or nil if there was no value before.

```
print(stringsAsInts["three"])
// print "nil"

stringsAsInts.updateValue(3, forKey: "three")
print(stringsAsInts["three"])
// print "Optional(3)\n"

stringsAsInts.updateValue(10, forKey: "three")
print(stringsAsInts["three"])
//print "Optional(10)\n"
```

Updating values

- To remove a value from the dictionary you can use the subscript syntax to set the value to nil,

```
stringsAsInts["three"] = nil
```

- or theremoveValueForKey() method.

```
stringsAsInts.removeValueForKey("three")
```

Exercice: Encode

- You are given a dictionary code of type [String:String] which has values for all lowercase letters. The code dictionary represents a way to encode a message.
  - For example if code["a"] = "z" and code["b"] = "x" the encoded version if "ababa" will be "zxzxz".
- You are also given a message which contains only lowercase letters and spaces.
- => **Use the code dictionary to <u>encode</u> the message and print it**.

Exercice: Encode

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var message = "hello world"
// output: ifmmp xpsme
var message2 = "wow this problem is hard"
// Output: xpx uijt qspcmfn jt ibse
```

Exercice: Encode

- Hint 1 : If a character doesn’t have a corresponding encoded character leave it unchanged.
- Hint 2 : Build the encoded message step by step by getting the corresponding encoded character from the dictionary.

Exercice: Encode Solution

```
var message = "hello world"

var encodedMessage = ""

for char in message.characters {
    var character = "\(char)"

    if let encodedChar = code[character] {
        // letter
        encodedMessage += encodedChar
    } else {
        // any non-encoded char (eg. space)
        encodedMessage += character
    }
}
```

Exercice: Decode

- You are given a dictionary code of type [String:String] which has values for all lowercase letters. The code dictionary represents a way to encode a message.
  - For example if code["a"] = "z" and code["b"] = "x" the encoded version if "ababa" will be "zxzxz". You are also given a encodedMessage which contains only lowercase letters and spaces.
- => **Use the code dictionary to <u>decode</u> the message and print it**.

Exercice: Decode

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var encodedMessage = "uijt nfttbhf jt ibse up sfbe"
// Output: "this message is hard to read"
```

Exercice: Decode

- Hint 1: You'll have to invert the code dictionary. **Create a new dictionary for this**.

Exercice: Decode Solution

```
var encodedMessage = "uijt nfttbhf jt ibse up sfbe"
var decoder: [String:String] = [:]

// reverse the code
for (key, value) in code {
    decoder[value] = key
}

var decodedMessage = ""

for char in encodedMessage.characters {
    var character = "\(char)"
    if let encodedChar = decoder[character] {
        // letter
        decodedMessage += encodedChar
    } else {
        // space
        decodedMessage += character
    }
}

print(decodedMessage)
//print "this message is hard to read\n"
```

Exercice: Names

- You are given an array of dictionaries. Each dictionary in the array contains exactly 2 keys “firstName” and “lastName”.
- => Create an **array of strings called firstNames** that contains only the values for “firstName” from each dictionary.

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]
```

Exercice: Names

- Expected values:

```
firstNames = ["Calvin", "Garry", "Leah",
"Sonja", "Noel"]
```

- Hint : Keep in mind that persons is an array of dictionaries, you'll have to process this array to get the required data.

Exercice: Names Solution

```
var firstNames: [String] = []

for person in people {
    if let firstName = person["firstName"] {
        firstNames.append(firstName)
    }
}

print(firstNames)
// print "["Calvin", "Garry", "Leah",
"Sonja", "Noel"]\n"
```

Exercice: Full Names

- You are given an array of dictionaries. Each dictionary in the array contains exactly 2 keys “firstName” and “lastName”.
- => Create an **array of strings called fullNames that contains the values for “firstName” and “lastName”** from the dictionary separated by a space.

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
        "age":"27"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]
```

Exercice: Full Names

- Expected values:

```
fullNames = ["Calvin Newton","Garry
Mckenzie","Leah Rivera",
    "Sonja Moreno","Noel Bowen"]
```

- Hint : Keep in mind that persons is an array of dictionaries, you'll have to process this array to get the required data.

Exercice: Full Names Solution 1

```
var fullNames: [String] = []

for person in people {
    print("person: \(person)")
    if let firstName = person["firstName"] {
        if let lastName = person["lastName"] {
            let fullName = "\(firstName) \
(lastName)"
            fullNames.append(fullName)
        }
    }
}
print(fullNames)
// print "["Calvin Newton", "Garry Mckenzie", "Leah
Rivera", "Sonja Moreno", "Noel Bowen"]\n"
```

Exercice: Full Names Solution 1 (suite)

```
/* Safer than the following:
for person in people {
    let firstName = person["firstName"]
    let lastName = person["lastName"]
    let fullName = "\(firstName) \
(lastName)"
    fullNames.append(fullName)
}
*/
```

Exercice: Full Names Solution 2

```
fullNames = []

for person in people {
    var fullName = " "
    for (key, value) in person {
        if key == "lastName" {
            fullName += value
        } else {
            // other fields
            fullName = value + fullName
        }
    }
    fullNames += [fullName]
}

print(fullNames)
// print "["Calvin Newton", "Garry Mckenzie", "Leah Rivera",
"Sonja27 Moreno", "Noel Bowen"]\n"
```

A swift tour

Closures

Closures

```
//
// Closures
//
var numbers = [1, 2, 6]

// Functions are special case closures ({})

// Closure example.
// `->` separates the arguments and return type
// `in` separates the closure header from the closure body
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

Closures

```
// When the type is known, like above, we can
do this
numbers = numbers.map({ number in 3 *
number })
// Or even this
//numbers = numbers.map({ $0 * 3 })

print(numbers) // [3, 6, 18]

// Trailing closure
numbers = numbers.sort { $0 > $1 }

print(numbers) // [18, 6, 3]
```

Part II.

After Objects

A swift tour

Classes (and objects)

Defining a class

- Classes, structures and its members have three levels of access control
- They are: internal (default), public, private

```
public class Shape {
    public func getArea() -> Int {
        return 0
    }
}
```

properties

- All methods and properties of a class are public.
- If you just need to store data in a structured object, you should use a `struct`

```
internal class Rect: Shape {
    var sideLength: Int = 1

    // Custom getter and setter property
    private var perimeter: Int {
        get {
            return 4 * sideLength
        }
        set {
            // `newValue` is an implicit variable available to
setters
            sideLength = newValue / 4
        }
    }
```

properties

- Computed properties must be declared as `var`, you know, cause' they can change

```
var smallestSideLength: Int {
    return self.sideLength - 1
}
```

- Lazily load a property: subShape remains nil (uninitialized) until getter called

```
lazy var subShape = Rect(sideLength: 4)
```

- If you don't need a custom getter and setter, but still want to run code before and after getting or setting a property, you can use `willSet` and `didSet`

```
var identifier: String = "defaultID" {
// the `willSet` arg will be the variable name for the new value
  willSet(someIdentifier) {
    print(someIdentifier)
  }
}
```

methods

- Initializer

```
init(sideLength: Int) {
    self.sideLength = sideLength
    // always super.init last when init custom
properties
    super.init()
}
```

- Override a mother method

```
    override func getArea() -> Int {
        return sideLength * sideLength
    }
} // end of class declaration
```

inheritance

- A simple class `Square` extends `Rect`

```
class Square: Rect {
    convenience init() {
        self.init(sideLength: 5)
    }
}

var mySquare = Square()
print(mySquare.getArea()) // 25
mySquare.shrink()
print(mySquare.sideLength) // 4

// cast instance
let aShape = mySquare as Shape
```

- Compare instances, not the same as == which compares objects (equal to)

```
if mySquare === mySquare {
print("Yep, it's mySquare")
}
```

Optional init

```
class Circle: Shape {
    var radius: Int
    override func getArea() -> Int {
        return 3 * radius * radius
    }

    // Place a question mark postfix after `init` is an
optional init
    // which can return nil
    init?(radius: Int) {
        self.radius = radius
        super.init()

        if radius <= 0 {
            return nil
        }
    }
}
```

Optional init (2)

```
var myCircle = Circle(radius: 1)
print(myCircle?.getArea())    // Optional(3)
print(myCircle!.getArea())    // 3
var myEmptyCircle = Circle(radius: -1)
print(myEmptyCircle?.getArea())    // "nil"
if let circle = myEmptyCircle {
    // will not execute since myEmptyCircle
is nil
    print("circle is not nil")
}
```

A swift tour

Optional Chaining

Optional chaining

- Ne pas oublier le ‘!’ ou le ‘?’ pour le unwrapping
- Swift:

```
var potentialExam: String? = "swift"
if let exam = potentialExam?.uppercaseString
{
    print(exam)
    } else {
    print("il n'y a jamais eu d'exam")
}
```

Optional Chaining

```
let count = object.items!.count // ERROR
```

- Versus

```
let count = object.items?.count // count is
implied to be an optional type and will
contain nothing in this case
```

- NOTE: Optional chaining in Swift is similar to messaging nil in Objective-C, but in a way that works for any type, and that can be checked for success or failure.

Casting

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

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

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

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")
]
```

```
// the type of "library" is inferred to be [MediaItem]
// Swift’s type checker is able to deduce that Movie and Song have
a common superclass of MediaItem, and so it infers a type of
[MediaItem] for the library array
```

Casting

- Checking type : operator **is** :

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        ++movieCount
    } else if item is Song {
        ++songCount
    }
}

print("Media library contains \(movieCount) movies and
\(songCount) songs")
// prints "Media library contains 2 movies and 3 songs"
```

Downcasting

```
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)")
    }
}

// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick Astley
```

A swift tour

Structures

Structures

```
// Structures and classes have very similar
capabilities
struct NamesTable {
    let names: [String]

    // Custom subscript
    subscript(index: Int) -> String {
        return names[index]
    }
}
```

Structures

```
// Structures have an auto-generated
(implicit) designated initializer
let namesTable = NamesTable(names: ["Me",
"Them"])
let name = namesTable[1]
print("Name is \(name)") // Name is Them
```

A swift tour

Enumerations

Enumerations

- Enums can optionally be of a specific type or on their own. They can contain methods like classes

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func getIcon() -> String {
        switch self {
        case .Spades: return "♠"
        case .Hearts: return "♡"
        case .Diamonds: return "♢"
        case .Clubs: return "♣"
        }
    }
}
```

- Enum values allow short hand syntax, no need to type the enum type when the variable is explicitly declared

```
var suitValue: Suit = .Hearts
```

String enums

- String enums can have direct raw value assignments or their raw values will be derived from the Enum field

```
enum BookName: String {
    case John
    case Luke = "Luke"
}
print("Name: \(BookName.John.rawValue)")
```

Enum with associated Values

```
enum Furniture {
    // Associate with Int
    case Desk(height: Int)
    // Associate with String and Int
    case Chair(String, Int)

    func description() -> String {
        switch self {
        case .Desk(let height):
            return "Desk with \(height) cm"
        case .Chair(let brand, let height):
            return "Chair of \(brand) with \(height) cm"
        }
    }
}

var desk: Furniture = .Desk(height: 80)
print(desk.description())     // "Desk with 80 cm"
var chair = Furniture.Chair("Foo", 40)
print(chair.description())    // "Chair of Foo with 40 cm"
```

A swift tour

Advanced

A swift tour

Protocols

Protocols

- `protocol`s can require that conforming types have specific instance properties, instance methods, type methods, operators, and subscripts

```
protocol ShapeGenerator {
    var enabled: Bool { get set }
    func buildShape() -> Shape
}
```

ObjC Protocols

- `Protocols declared with @objc allow optional functions, which allow you to check for conformance

```
@objc protocol TransformShape {
    optional func reshape()
    optional func canReshape() -> Bool
}

class MyShape: Rect {
    var delegate: TransformShape?

    func grow() {
        sideLength += 2

        // Place a question mark after an optional property, method, or
        // subscript to gracefully ignore a nil value and return nil
        // instead of throwing a runtime error ("optional chaining").
        if let reshape = self.delegate?.canReshape?() where reshape {
            // test for delegate then for method
            self.delegate?.reshape?()
        }
    }
}
```

Protocol conformance

- **`Protocols conformance at the class definition level**
- The protocol name is appended beside the superclass name, separated with a comma. If there isn't a superclass, then you just write a colon, followed by the protocol name.
- Both MyClass and AnotherClass conform to the SampleProtocol

```
class MyClass: SampleProtocol
{
    // Conforming to SampleProtocol
    func someMethod() {
    }
}

class AnotherClass: SomeSuperClass, SampleProtocol
{
    // A subclass conforming to SampleProtocol
    func someMethod() {
    }
}
```

Protocol conformance

- **`Protocols conformance at the instance level**
- You may notice the question mark syntax which indicates that it's a property with an optional value (there may or may not be an object assigned to it).

```
@objc protocol TransformShape {
    optional func reshape()
    optional func canReshape() -> Bool
}

class MyShape: Rect {
    var delegate: TransformShape?
…
```

Delegation

- Delegation works hand in hand with protocols because it allows a class to specify a delegate property which conforms to some protocol.
  - Then a second class which actually conforms to that protocol can be assigned to that property. Now the first class can communicate with the second class through the delegate property using the methods and properties as defined by the protocol
- Declaring a delegate property is just like declaring any other property and you specify the protocol name as the type of the property.

```
class FirstClass
{
    var delegate:SampleProtocol?
}
```

- Calling a delegate method.

```
delegate?.someMethod()
```

A swift tour

Extension

Extension

- `extension`s: Add extra functionality to an already existing type
- Square now "conforms" to the `CustomStringConvertible` protocol.

```
extension Square: CustomStringConvertible {
    var description: String {
        return "Area: \(self.getArea()) - ID: \
(self.identifier)"
    }
}

print("Square: \(mySquare)")
```

Extension

- You can also extend built-in types

```
extension Int {
    var customProperty: String {
        return "This is \(self)"
    }

    func multiplyBy(num: Int) -> Int {
        return num * self
    }
}

print(7.customProperty) // "This is 7"
print(14.multiplyBy(3)) // 42
```

A swift tour

Generics

Extension

- Similar to Java and C#. Use the `where` keyword to specify the requirements of the generics

```
func findIndex<T: Equatable>(array: [T], _
valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
let foundAtIndex = findIndex([1, 2, 3, 4], 3)
print(foundAtIndex == 2) // true
```

A swift tour

Codes examples

Shuffle

- Fisher-Yates (fast and uniform) shuffle to any collection type. The naming and behavior follow Swift 2.0's sort() and sortInPlace() methods

```
extension CollectionType {
    /// Return a copy of `self` with its
elements shuffled
    func shuffle() -> [Generator.Element] {
        var list = Array(self)
        list.shuffleInPlace()
        return list
    }
}
```

Shuffle

```
extension MutableCollectionType where Index == Int {
    /// Shuffle the elements of `self` in-place.
    mutating func shuffleInPlace() {
        // empty and single-element collections
don't shuffle
        if count < 2 { return }

        for i in 0..<count - 1 {
            let j =
Int(arc4random_uniform(UInt32(count - i))) + i
            guard i != j else { continue }
            swap(&self[i], &self[j])
        }
    }
}
```

Shuffle

```
[1, 2, 3].shuffle()
// [2, 3, 1]

let fiveStrings = 0.stride(through: 100, by:
5).map(String.init).shuffle()
// ["20", "45", "70", "30", ...]

var numbers = [1, 2, 3, 4]
numbers.shuffleInPlace()
// [3, 2, 1, 4]
```

Sécurité et Sûreté

Les mécanismes du langage

Gestion des erreurs

- « In this example, the makeASandwich() function will throw an error if no clean dishes are available or if any ingredients are missing. Because makeASandwich() can throw an error, the function call is wrapped in a try expression. By wrapping the function call in a do statement, any errors that are thrown will be propagated to the provided catch clauses.

- If no error is thrown, the eatASandwich() function is called. If an error is thrown and it matches the SandwichError.outOfCleanDishes case, then the washDishes() function will be called. If an error is thrown and it matches the SandwichError.missingIngredients case, then the buyGroceries(_:) function is called with the associated [String] value captured by the catch pattern. »

```
func makeASandwich() throws {
    // ...
}

do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

Assertions

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
// this causes the assertion to trigger, because age is not >= 0
```

- In this example, code execution will continue only if age >= 0 evaluates to true, that is, if the value of age is non-negative. If the value of age is negative, as in the code above, then age >= 0 evaluates to false, and the assertion is triggered, terminating the application.
  The assertion message can be omitted if desired, as in the following example:

```
assert(age >= 0)
```

- NOTE : Assertions are disabled when your code is compiled with optimizations, such as when building with an app target's default Release configuration in Xcode.

Assertions

When to Use Assertions

Use an assertion whenever a condition has the potential to be false, but must definitely be true in order for your code to continue execution. Suitable scenarios for an assertion check include:

- An integer subscript index is passed to a custom subscript implementation, but the subscript index value could be too low or too high.
- A value is passed to a function, but an invalid value means that the function cannot fulfill its task.
- An optional value is currently nil, but a non-nil value is essential for subsequent code to execute successfully.

NOTE: Assertions cause your app to terminate and are not a substitute for designing your code in such a way that invalid conditions are unlikely to arise. Nonetheless, in situations where invalid conditions are possible, an assertion is an effective way to ensure that such conditions are highlighted and noticed during development, before your app is published. »

Access control

- Swift supports five access control levels for symbols: open, public, internal, fileprivate, and private. Unlike many object-oriented languages, these access controls ignore inheritance hierarchies: private indicates that a symbol is accessible only in the immediate scope, fileprivate indicates it is accessible only from within the file, internal indicates it is accessible within the containing module, public indicates it is accessible from any module, and open (only for classes and their methods) indicates that the class may be subclassed outside of the module