- **Propósito de creación:** Se creó con el **propósito de reemplazar** Objective-C **por un lenguaje más moderno y seguro** que era el lenguaje principal que se utilizaba para el desarrollo de aplicaciones nativas (**fue creado en 1980**).

- **Open source:** Posibilidad de usarlo en distintos SO gracias a la liberación del código fuente, hoy en día no sólo puede usarse para el desarrollo de aplicaciones nativas de Apple, sino que **es posible usarla en Linux y Windows**. Antes sólo podía usarse con XCode (IDE para macOS), hoy en día puede usarse también con Visual Studio Code.

- **Historia de por qué Apple usaba Objective-C:**
    - **Objective-C** es un lenguaje de programación orientado a objetos creado como un superconjunto de C para que implementase un modelo de objetos parecido al de Smalltalk.
    - Originalmente fue **creado** por Brad Cox y la corporación **StepStone** en **1980**.
    - En **1988**, **NeXT licenció** el **Objective-C** de StepStone y extendió el compilador GCC para dar **soporte a Objective-C**.
    - En **1996 Apple adquiere NeXT** y **adopta OpenStep** en su **nuevo sistema operativo**, **Mac OS X**. OpenStep **incluía Objective-C** y la herramienta de desarrollo basada en Objective-C de NeXT, Project Builder (que ahora se conoce por Xcode), así como la herramienta de diseño de interfaz, Interface Builder.
      Así la mayoría de la actual Cocoa *(Framework más usado para el desarrollo nativo de apps en macOS X)* está basada en objetos de interfaz de OneStep.

- **Curva de aprendizaje:** Es relativamente fácil de aprender, luego vamos a ver ejemplos de sintaxis, para una persona que ya hizo algo de programación es fácil de leer y entender.

- **Fuertemente tipado:** Una vez que se asigna un **tipo de dato** a una variable/constante, **no se puede modificar** su tipo de dato a no ser que se haga un **casteo explícito**.
  Tampoco permite expresiones con operandos de diferente tipo.

- **ARC (mecanismo de seguimiento de uso):** El ARC es un mecanismo de seguimiento de uso que cuantifica el nro de referencias de variables y libera la memoria de las menos referenciadas.

- **Interoperabilidad con Objective-C:** permite a los desarrolladores **utilizar librerías o módulos desarrollados en Objective-C** dentro de Swift. Esto **facilita la transición (migración)** de proyectos existentes **a Swift** (motivo por el cual ya fueron migradas muchas de las aplicaciones que existían desarrolladas con Objective-C).
  La interoperabilidad **se logra mediante** un proceso llamado "puente" (**bridging**). El

puente se encarga de traducir el código de un lenguaje al otro.

- **Compilador:**
  - **Verificación de tipos**: realiza verificación de tipos en tiempo de compilación y realiza la inferencia de tipos de ser necesario.
  - **Eliminación de "código muerto"**: utiliza análisis estáticos para determinar **qué partes** del código **nunca se ejecutarán** y las **elimina** del programa final.
  - **Propagación de constantes**: **identifica las referencias** a variables cuyo valor es **constante y las sustituye por su valor** correspondiente (en el código) en tiempo de compilación. **(Evita** tener que **acceder a la memoria** para recuperar su valor).
  - **Optimizar expresiones constantes**: Cuando se propagan constantes en expresiones matemáticas u operaciones lógicas, el compilador puede **evaluar** estas **expresiones en tiempo de compilación y reemplazarlas por su resultado constante**. (Esto **reduce** la **carga** del **procesador**).

- **Operadores especiales:**

Operator	Description	Example
`? :`	Ternary Operator - returns value based on the condition	`(5 > 2) ? "Success" : "Error" // Success`
`??`	Nil-Coalescing Operator - checks whether an optional contains a value or not	`number ?? 5`
`...`	Range Operator - defines range containing values	`1...3 // range containing values 1,2,3`

- **Variables:**
  - Usa **"let"** para declarar: constantes y **"var"** para declarar: variables.
  - Se pueden declarar múlitples variables con distintos tipos en una sola linea (en el ejemplo podrían ser <> tipos).
  - **Inferencia:** cuando no se define un tipo explícito, el **compilador** puede inferir el tipo.
    En los **ejemplos** de las filminas, las variables adquieren el tipo de dato **entero (Int)** justamente porque son números enteros.
- **Estructuras de datos:** todas las estructuras de datos tienen métodos propios como append(), remove(), isEmpty(), etc.
  - **Arreglos:**
    - Se **infieren a tipos de datos heterogéneos** (todos int, float, string) a no ser que se explicite que el arreglo es de tipo **[Any]** (este permite

distintos tipos de datos para el arreglo).

- **Set:**
  - Conjunto de datos de un único tipo y valores únicos.
- **Diccionarios:**
  - Clave valor, idénticos a los vistos en python.

- **Estructuras de control:** contiene estructuras como **if** (con expresiones delimitadas con llaves **{}**), **switch-case**, **for in** loop (es un for each), **while** loop, **repeat...while** loop.
  - **For in:** (particular de Swift, puede usar una clausura **where** como las de las consultas sql).

    También tiene **For in Range...** `for i in 1...3 {`

    For más convencional: `for i in stride(from: 1, to: 10, by: 2)`
  - **Break:** permite salir de flujo de control donde está actualmente, si es dentro de un if, del if, si es dentro de un for, del for. En el ejemplo sale del if, no de todo del flujo for que lo contiene.
  - **Guard:** es una especie de if pero que ejecuta su bloque de código si se no se cumple la condición. Se suele usar para cortar el flujo del programa, ejemplo en la filmina el Output sería:

    Output

    ```
    Eligible to vote
    ```

    lo que significa que sale de la función.

- **Casteo e Interpolación:** no es posible concatenar variables de diferente tipo.
  Para hacer esto se puede o castear el tipo de variable o usar interpolación de cadenas tipo f-strings.

- **Opcionales:** ejemplos de uso: como cuando se **recibe datos de una fuente** externa (donde no controlamos lo que vamos a recibir).
  - También se puede usar en: `let name = person?.name` en donde si el valor de **person** es nulo, la expresión completa devuelve nil sin generar una excepción.
  - El operador **!** es otro tipo de opcional, se utiliza para desempaquetar el valor del opcional, si hicieramos print(optionalTitle) sin usarlo dentro del if, retornaría: Optional("Mi titulo") en cambio, String! → "Mi titulo": **El problema**

si el valor no existe, cuando se intenta de acceder

- **Funciones:**
  - **2.**
    - Los parámetros son pasados como constantes, es decir que no pueden ser modificados dentro del ámbito de la función, si intentás hacerlo da un error en tiempo de compilación:
      ```
      Error: cannot assign to value: 'name' is a 'let' constant
      ```
    - El **pasaje de parámetros** siempre debe cumplir con la misma estructura declarada en la función.
      - **No es posible** hacer: **findSquare(num)** ni aún habiendo declarado el valor de num antes de enviárselo como param.
      - **Ni tampoco** se puede pasar solamente el valor **findSquare(3)**.
    - Siempre **debe cumplir** que tenga el **mismo orden, cantidad, nombre** (label) **y tipo de datos** de c/ **parámetro**. **No es posible** hacer: **findSquare(numerito: 3).**
      Ni: `greet(b: "mundo",a: "hola");`
      en caso de tener: `func greet(a: String, b: String)`
  - **3.**
    - Es posible definir valores por defecto para los param. de una función. Esto nos brinda la libertad de llamar a la función con distinta cantidad de param. ej:
      func addNumbers(a: 1, b: 2) //sobreescriben los valores por defecto
      func addNumbers(a: 5) //sobreescribe el valor por defecto de "a"
      func addNumbers() //utiliza los valores definidos por defecto
  - **4.**
    - Se pueden utilizar "Arguments label" para expresar las llamadas a las funciones de manera más expresiva, como si fuera una oración.
      Llamada **sum(of: 2, and: 3)**
    - Luego dentro del ámbito de la función, **no es posible** usar los arguments label para trabajar, sólo se podría operar con **"a"** y **"b"**.
  - **5.**
    - **inout** sirve para declarar que el parámetro que se va a recibir será pasado **por referencia** (referencia directa a la dirección de memoria) // *(por valor era cuando se creaba una copia del valor de la variable)*.
      Esto implica tener que cambiar la forma de llamar a la función:
      changeName(name: &userName) // teniendo que agregar el ampersand &
  - **6.** return de múltiples variables. El valor retornado almacena una **tupla de**

valores result.1 = message, result.2 = marks, result.n = otros valores que podría tener si retornara más de dos la función.

- **Programación funcional:**
  - Además de las funciones vistas anteriormente con las cuales se podían hacer:
    - **Funciones puras** (funciones que siempre retornan los mismos valores para los mismos argumentos de entrada) y...
    - **Funciones Inmutables** (que no cambian su valor)

    también se pueden crear funciones de **orden superior** (descripción en la filminas).

- **Closures:** //Para la filmina de más adelante que hay como ejemplo
  - **parameters** - any value passed to closure
  - **returnType** - specifies the type of value returned by the closure
  - **in** (optional) - used to separate **parameters/returnType** from closure body

---

- **Orientado a objetos:**
  - Existe otra estructura similar a la de las clases llamada "**Struct**", pero estás no proveen Herencia (como las clases).
  - Se puede hacer **override** de cualquier función (sea de clase, interfaz, o agregada en extension).
  - El **init** funciona como constructor de la clase.
    Es posible liberar la memoria ocupada por la instancia de la clase asignandole "**obj = nil**", Swift permite añadir funcionalidad extra a la "desinicialización" de la variable, mediante la escritura del método "**deinit**". Si deinit no está definido, no hay problema, simplemente sirve para añadir funcionalidad extra al momento de.

  - **Atributos computados:** son atributos de clase a los que se le pueden asignar un valor calculado.
    Convencionalmente se utilizan para definir los setters y getters de los atributos.

  - **Atributos estáticos:** son los llamados atributos de clase

---

- **Sobrecarga de métodos:** Swift permite sobrecarga de métodos (mismo nombre de función, con distintas implementaciones → **distinta cantidad y tipo de datos** de parámetros, esto es la sobrecarga).
  En init permite dejar un "constructor" por default.

- **Sobrescritura de métodos:** con "**final"** podemos prevenir la sobresescritura, bloquéandola. En el ejemplo, si se quisiese sobreescribir una función declarada como "**final"**, daría un **error en tiempo de compilación**, con la alerta de que no es posible sobreescribirla por la definición "final".

- **Scope de variables:** El "scope" se refiere a la región de código donde una variable es visible y accesible.
  - **Global scope:** tipo programación imperativa, se puede escribir código sin que esté dentro de ninguna estructura (clase, protocolo, struct, etc). Tipo Pascal, C, C++, ...
  - **Local scope:** lo mismo sucedería si intentara declarar una función dentro de "**printLocalVar()**" y luego invocarla desde fuera, la llamada a la inner function (**o nested function** le suelen llamar también) daría un error en tiempo de compilación.

```
func outerFunction() {
    var outerVar = "I'm an outer variable"

    func innerFunction() {
        var innerVar = "I'm an inner variable"
        print(outerVar)
        print(innerVar)
    }

    innerFunction() // Output: I'm an outer variable
                    //         I'm an inner variable
}
```

  - **Scopes en Clases y Structs:** por defecto, si no se agrega "static", la propiedad es de instancia, por el contrario con static, es de clase (atributo/método compartido y accesible por todas las instancias).

- **Conclusiones:**
  - **Asentado como más usado vs Objective-C:** Desde su salida, Swift ganó popularidad entre los desarrolladores debido a su **facilidad de uso**, **seguridad y rendimiento**, habiéndose **asentado como el lenguaje más** usado en aplicaciones actuales **en comparación con Objective-C**.

ToDo:

Comments:

- **Manejo de punteros:** no encontré información completa sobre el manejo de punteros por default en Swift. Encontré uso de punteros mediante el uso de librerías de C/C++, pero no de manera “nativa” en el lenguaje.
  Ejemplos: MemoryLayout<genericType> y UnsafePointer
  - https://developer.apple.com/documentation/swift/unsafepointer
  - https://www.kodeco.com/7181017-unsafe-swift-using-pointers-and-interacting-with-c#toc-anchor-002
  - https://prafullkumar77.medium.com/pointers-in-swift-f8d651d0e724
- **Manejo de errores:** no llegué a darlo: buena referencia de lectura:
  - https://www.programiz.com/swift-programming/error-handling

```swift
// create an enum with error values
enum DivisionError: Error {

  case dividedByZero
}
```

```swift
// create a throwing function using throws keyword
func division(numerator: Int, denominator: Int) throws {

  // throw error if divide by 0
  if denominator == 0 {
    throw DivisionError.dividedByZero
  }

  else {
    let result = numerator / denominator
    print(result)
  }
}

// call throwing function from do block
do {
  try division(numerator: 10, denominator: 0)
  print("Valid Division")
}

// catch error if function throws an error
catch DivisionError.dividedByZero {
  print("Error: Denominator cannot be 0")
}
```

- **Enums:**

```
enum Season {
  case spring, summer, autumn, winter
}
```