PROGRAMACIÓN IOS

En Swift existen dos maneras de encapsular la información, por un lado están las clases y por otro lado las estructuras. Este lenguaje de programación crea automáticamente los métodos accesores y modificadores(GETTER/SETTER)

Características Iguales:

-Definen propiedades y almacenan valores.

-Definen métodos que aportan funcionalidad

-Definen constructores (Clases)

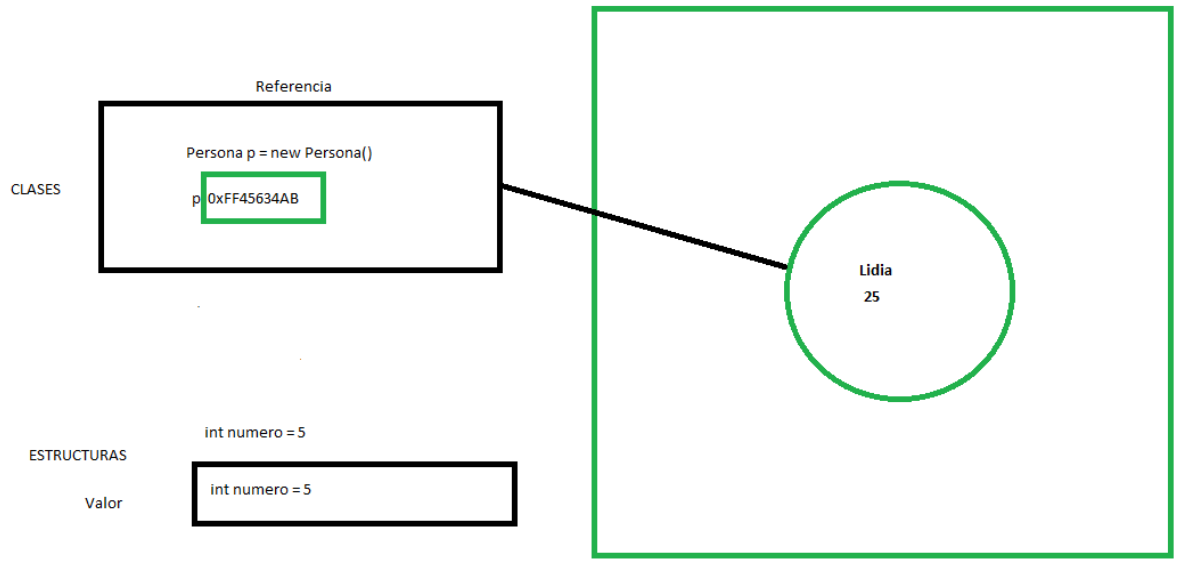
-Pueden ser expandidas para aumentar su funcionalidad (Herencia)

Características – Estructuras:

Son variables normales, en el que se guardan los valores del dato y no la posición de memoria donde se encuentran. NO son referencias como en las Clases. Definen constructores para todas sus propiedades, ya que Swift los crear automáticamente.

Características – Clases:

Son variables de referencias, que guardan la posición de memoria de donde se encuentra el valor. Se puede utilizar la herencia, es decir, siguen las reglas de la programación orientada a objetos(POO). Tienen métodos cuando se libera de memoria (deinit), como en JAVA, pero la diferencia está en que no se usan mucho.



ESTRUCTURAS

-Se definen con la palabra “struct”.

-Dentro las estructuras se definen las propiedades.

-Se crean constructores por defecto basándose en las propiedades.

-Fácil de imprimir en la consola, los valores por defecto, sin necesidad de toString.

-No se pueden comparar

resolution1 == resolution2 – (Esto no se puede hacer) ERROR

-Las instancias de las estructuras siempre se pasan por valor(se hace una copia de variable cuando se pasa por parámetro a una función).

```
struct Resolution {
    var width = 0
    var height = 0
}
```

CONSTRUCTORES:

```
Let resolution1 = Resolution()
Let resolution2 = Resolution(width: 640, height: 480)
Let resolutionH = Resolution (height: 480)
var resolutionW = Resolution (width: 640)
```

Para acceder a las propiedades, se pueden hacer mediante JAVA. En este caso se está accediendo a la propiedad de WIDTH pero para cambiar el valor inicial. Esto sólo se puede hacer cuando la variable no sea constante, es decir, sea var.

```
resolutionW.width = 1080
```

CLASES

-Se definen mediante la palabra “class”

- Las propiedades se definen como en JAVA, mediante la inferencia de equipos.

-Las clases NO tienen varios constructores por defecto. Sólo tienen el constructor por defecto, sin parámetros.

-IMPORTANTE: Cuando se tiene una propiedad de tipo LET(constante), no se puede cambiar la referencia a donde se esté apuntando al objeto, pero las clases sí se puede cambiar el valor. No se puede cambiar la referencia, pero si sus propiedades.

-Las instancias de las variables de las clases se pasan por referencia, es decir, se pasa el mismo objeto(no se hace copia del objeto).

-El objeto se crea sin la palabra "new"

Let someVideo = VideoMode()

class VideoMode{

//Crear objeto resolution1

var resolution1 = Resolution() - (Propiedad estructura)

var interlaced = false – (Propiedad Boolean)

var frameRate = 0.0 – (Propiedad Double)

*var name: String? – (Propiedad String) = *Optional: el valor pueda también ser nil(Nulo)*

}

PROPIEDADES

-Acceder a un método GET

print(resolution1.width) / print(resolution1.resolution.width)

resolution1.getResolution(); - (GET JAVA)

-Acceder a un método SET

resolution1.resolution.width = 1280 / resolution1.width = 1280

resolution1.setResolution(new Resolution(1920, 1080); - (SET JAVA)

-Crear un nuevo objeto: (Aunque sea la variable LET, se permite modificar los valores)

Let resolutionHD = VideoMode()

resolutionHD.resolution = Resolution(width: 1920, height: 1080)

resolutionHD.interlaced = true

resolutionHD.name = "Full HD"

resolutionHD.frameRate = 30.0

print(resolutionHD)/(No sale bien escrito por defecto en la consola) = dump(resolutionHD)

VARIABLES

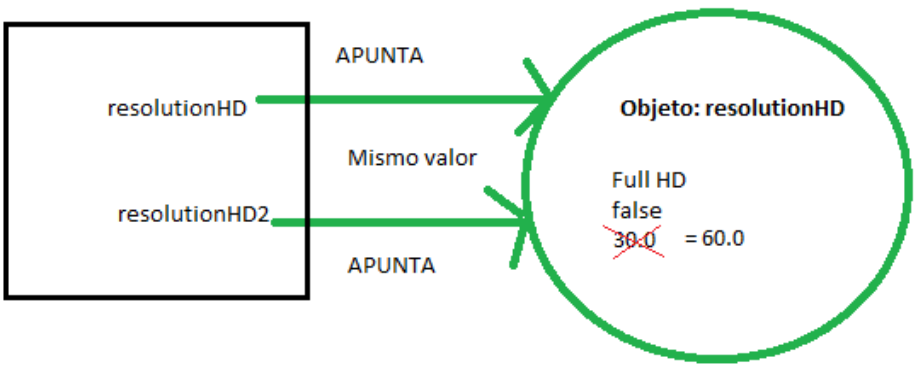
Las variables de Clases son referencias, es decir, se puede crear una variable haciendo referencia a otra variable previamente definida. Por ejemplo:

Let resolutionHD2 = resolutionHD

print(resolutionHD2.frameRate)

print(resolutionHD2.frameRate = 60.0)

print(resolutionHD.frameRate)



Las variables de las Estructuras son variables normales que se pasan mediante un valor

print(resolution1)

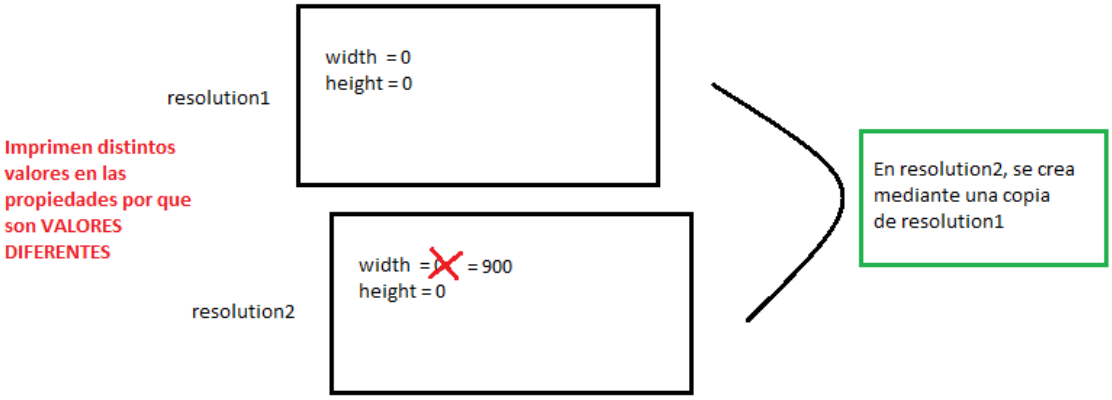
var resolution2 = resolution1

print(resolution2)

resolution2.width = 900

print(resolution1)

print(resolution2)



//Diferencias:

-

INDENTITY OPERATORS(Operadores de identidad) – '==='

-Sólo se puede utilizar con Clases, en Estructuras o Enumerados no se pueden utilizar estos operadores, si se usa pueden dar fallos al compilar y los valores que se le asignan en las constantes o variables definidas anteriormente, resultan copiadas. Por lo que para hacer una comparación se usa ===, esto equivale a la asignación == en JAVA y esto significa la comparación de si ambas variables apuntan al mismo objeto.

If resolutionHD=== resolutionHD2 {

print(resolution1 y resolution2 provienen de la misma instancia VideoMode)

}

resolution1 === resolution2 – ERROR

Función

-Todos los parámetros pasados en una función son variables de tipo LET, no se pueden cambiar. Excepto si se coloca la palabra “inout” sería variable.

Func probandoInstancias(_a:Resolution(E:ENTRADA), _b:VideoMode(C:ENTRADA), _c:inout Resolution(ENTRADA/SALIDA)){

a.width = 34 – ERROR (No se puede modificar por ser variable de tipo LET)

-Se hace una copia de ‘a’ para luego poder trabajar con los valores de la estructura Resolution.

var a2 = a

a2.width = 34

-Como VideoMode es una clase que tiene variables de referencia, permite cambiar el valor de las propiedades de la constante “let” de ‘b’

b.name = “CAMBIAR”

b = VideoMode() – ERROR (No se puede instanciar ‘b’ a un nuevo Objeto(VideoMode)

-En cambio ‘c’, como es una variable de tipo VAR, si se puede modificar.

c.width = 50

}

Var video = VideoMode()

Video.name = “video”

Var resolution = Resolution(width:320, height: 480)

Var resolution2 = Resolution(width:987, height:630)

probandoInstancias(resolution(LET), video(LET), &resolution2(VAR)) ***inout(&)**

print(video.name ?? “no hay nombre”) = “CAMBIAR”

print(resolution.width) = 320

print(resolution2.width) = 50

Consideraciones

ESTRUCTURA

-Si se quiere encapsular algunos valores, se consigue fácil con una Estructura.

-Cuando se instancia la estructura, los valores encapsulados son copiados y no referenciados.

-Las propiedades almacenadas en las Estructuras son de tipo simple, es decir, que no haya referencias dentro de las propiedades de éstas.

-No se necesita herencia

-*OJO!*Los Strings, Arrays y Diccionarios, mientras que en JAVA los Arrays y Diccionarios son objetos de referencias, son ESTRUCTURAS. Cuando se igualan los Arrays y los Diccionarios en las Estructuras se crea una copia de esto, pero en JAVA se hace una referencia al mismo objeto, es decir, al mismo Array o el mismo Diccionario.

var nombres = [“Lidia”, “Juan”, “Rosa”]

var nombresCopia = nombres

-Si se borra un Array, no tiene porque borrarse el otro Array

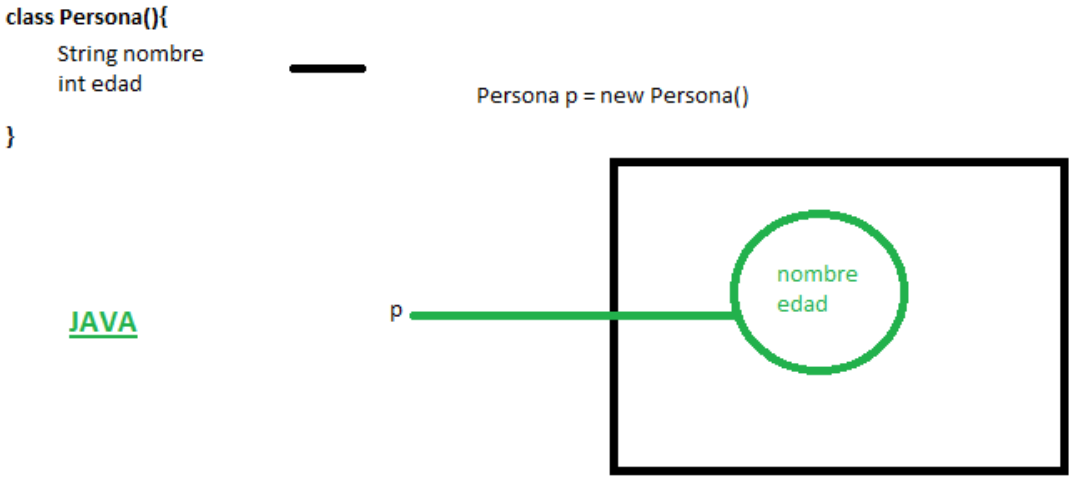
nombresCopia,remove(at: 0) – (se quita la posición 0 = Lidia)

print(nombres) = Lidia, Juan, Rosa

print(nombresCopia) = Juan, Rosa

STORED PROPERTIES

Una stored properties es una constante o una variable, que se almacena como parte de una instancia de una Clase o una Estructura. Esto equivale a las propiedades NO estáticas de JAVA, es decir, las dinámicas.



struct rango {

var primero : Int

let segundo : Int

}

Var rango2 = rango(primero: 5, segundo: 0)

print(rango2.primero) = 6 – Se modifica el valor 'primero' y no el valor 'segundo'

let rango3 = rango(primero: 0, segundo: 2)

print(rango3.segundo) > LET = 2

print(rango3.primero) > LET = 0

*Se pueden acceder a las propiedades pero no modificarlas.

Cuando una propiedad no pertenece a la instancia, y esto en JAVA es equivalente, a una propiedad 'static'

Class Persona{

static var edadMaxima = 100

var nombre : String?

var edad: Int = 0

}

-Crear un objeto persona

var persona = Persona()

-Para acceder a las propiedades de objetos se hace mediante el objeto

persona.nombre = "Lidia"

-Para acceder a las propiedades de la clase se hace a través de la clase

persona.edad = Persona.edadMaxima

dump(persona)

COMPUTED PROPERTIES

Este tipo de propiedades no almacenan ningún valor, sino que proporcionan los métodos accesores y modificadores(GETTER/SETTER) a otras propiedades. Se pueden utilizar tanto en las Clases como en las Estructuras.

struct Usuario{

-Estas dos variables de tipo VAR, almacenan información.

var nombre : String

var apellido : String

-Mientras que en esta variable, no almacena información pero tiene su propio GET y SET. *COMPUTED PROPERTIE*

var nombreComp : String{

get{

return nombre + " " + apellido // usuario.nombreComp

}

-Si al acceder al método SET no se le asigna ningún parámetro de entrada, Swift lo crea automáticamente.

set(nombreUsuario) {

let values = nombreUsuario.split(separator: " ")

nombre= String(values[0])

apellido= String(values[1])

}

}

}

-También se puede hacer solo el GET en una computed propertie. El GET es opcional.

var nombreComp : String{

return "Hola mundo"

}

-**OJO!**Se puede hacer un método GET, sin un método SET. Pero NO se puede hacer un método SET, sin un método GET.

LAZY STORED PROPERTIES

Son propiedades que su valor inicial no es calculado, hasta que no se utiliza por primera vez. Es decir, siempre se declaran como variables de tipo VAR. Sólo se van crear las propiedades del objeto cuando se accedan a ellos.

En este tipo de propiedades son útiles cuando su valor inicial depende de factores externos cuyos valores no se conocen hasta después de que se instancie la clase, y no quiere crear memoria hasta entonces.

```
Class DataImporter{
    var filename = "data.txt"
}
Class DataManager{
    lazy var importer = DataImporter() (*)
    var data = [String]()
}
let manager = DataManager()
```

-*La instancia de DataImporter para la propiedad de importer no se ha creado todavía ya que aún no se ha utilizado.

PROPERTY OBSERVERS(Propiedades Observadoras)

Se utilizan para detectar cambios en las propiedades, se acude a ellas siempre que se precise cambiar un valor. Se pueden usar en cualquier propiedad, excepto en las LAZY PROPERTIES.

```
Class ContadorPasos{
```

-Cuando se ejecute el método, éste quiere modificar el valor de totalPasos

```
    var totalPasos : Int  = 0{
```

-Ambos métodos se ejecutan cuando haya un cambio en la propiedad totalPasos de la Clase

```
        willSet(nuevoTotalPasos) {
```

-willSet: Este método se ejecutará antes de que se invoque a la propiedad, es decir, antes de que se cambie el valor de ésta.

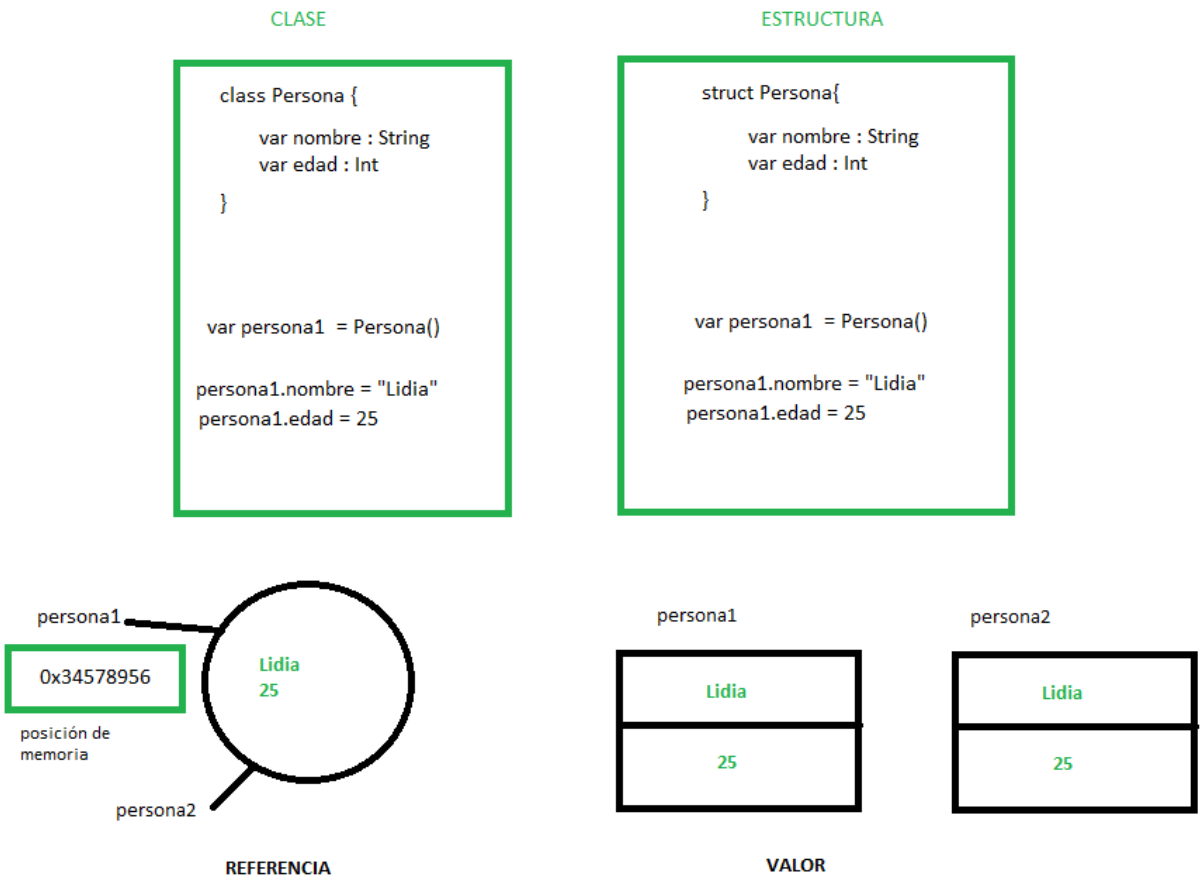
```
        }
        didSet(viejoTotalPasos){
```

-didSet: Este método se ejecutará después de que se invoque a la propiedad, es decir, justo después de que el valor de ésta ya se haya cambiado.

```
        }
}
```

PROPERTIES	DEFINICIÓN	EJEMPLOS
Stored Properties	Una stored properties es una constante o una variable, que se almacena como parte de una instancia de una clase o una estructura.	*struct rango {* *var primero : Int* *let segundo : Int* *}* *var rango2 = rango(primero: 5, segundo: 0)* *print(rango2.primero) = 6* *let rango3 = rango(primero: 0, segundo: 2)* *print(rango3.segundo) > LET = 2* *print(rango3.primero) > LET = 0*
Computed Properties	Este tipo de propiedades no almacenan ningún valor, sino que proporcionan los métodos accesores y modificadores(GETTER/SETTER) a otras propiedades. Se pueden utilizar tanto en las Clases como en las Estructuras.	*struct Usuario{.* *var nombre : String* *var apellido : String* *var nombreComp : String{* *get{* *return nombre + " " + apellido* *}* *set(nombreUsuario) {* *let values = nombreUsuario.split(separator: " ")* *nombre= String(values[0])* *apellido= String(values[1])* *}* *}* *}*
Lazy Stored Properties	Son propiedades que su valor inicial no es calculado, hasta que no se utiliza por primera vez. Es decir, siempre se declaran como variables de tipo VAR. Sólo se van crear las propiedades del objeto cuando se accedan a ellos.	*Class DataImporter{* *var filename = "data.txt"* *}* *Class DataManager{* *lazy var importer = DataImporter()* *var data = [String]()* *}* *let manager = DataManager()*

Properties Observers	Se utilizan para detectar cambios en las propiedades, se acude a ellas siempre que se precise cambiar un valor. Se pueden usar en cualquier propiedad, excepto en las lazy properties.	*Class ContadorPasos{* *var totalPasos : Int = 0{* *willSet(nuevoTotalPasos){}* *didSet(viejoTotalPasos){ }* *}*



MÉTODOS DE INSTANCIA

-Sirven para reutilizar código, la diferencia entre funciones y métodos, es que estos últimos son funciones que están definidos dentro de una Clase o de una Estructura.

-Declarar método dentro de una clase

```
class Contador{
    var count = 0
```

-Método de instancia(incrementar())

-Existe la sobrecarga de métodos, es decir, son métodos con igual nombre pero con distinto número y tipo de parámetros.

```
    Func incrementar(){
        count += 1
    }
    Func incrementar(by amount: Int){
        count += amount
    }
}
```

-Se pueden modificar los valores de los atributos de una clase con estos métodos, realizando la modificación dentro de éste.

```
class Point2{
    var x = 0.0
    var y = 0.0
    func change(x x1: Double, y y1 : Double){
        x += x1
        y += y1
    }
}
```

-En cambio en Estructuras o en los Enumerados, esta función se realiza de distinta forma. Sino que tenemos que invocar al método 'mutating'. Esta función permite mutar los valores.

```
struct Point3{
    var x = 0.0
    var y = 0.0
    mutatin change(x x1: Double, y y1 : Double){
        x += x1
        y += y1
    }
```

```
}
```

MÉTODOS DE CLASES

-Estos métodos equivalen a los métodos estáticos en JAVA. Se declara un método estático mediante la expresión 'class', no 'static' como en las propiedades.

-Para usar este tipo de métodos no es necesario crear el objeto, sino que directamente se usa en la propia clase.

```
class Clase1{
        class func tipoMetodo(){
                print("método")
        }
}
clase1.tipoMetodo()
```

SELF

-Este método es equivalente, a THIS en JAVA. Es una referencia de si misma, es decir, de la propia clase.

```
struct Point{
        var x = 0.0
        var y = 0.0
        func isToTheRightOf(x: Double) -> Bool){
                return self.x(atributo) > x(parámetro)
        }
```

-Esta función tiene un parámetro llamado 'x' de tipo Double, pero este parámetro oculta la visibilidad del atributo llamado 'x'. Para poder acceder al atributo 'x' de la clase se puede usar la palabra 'self'.

```
}
let punto1 = Point(x: 4.0, y: 5.0)
if(punto1.isToTheRightof(x: 1.0){
```

print(“El punto está a la derecha de la línea donde x = 1.0”)

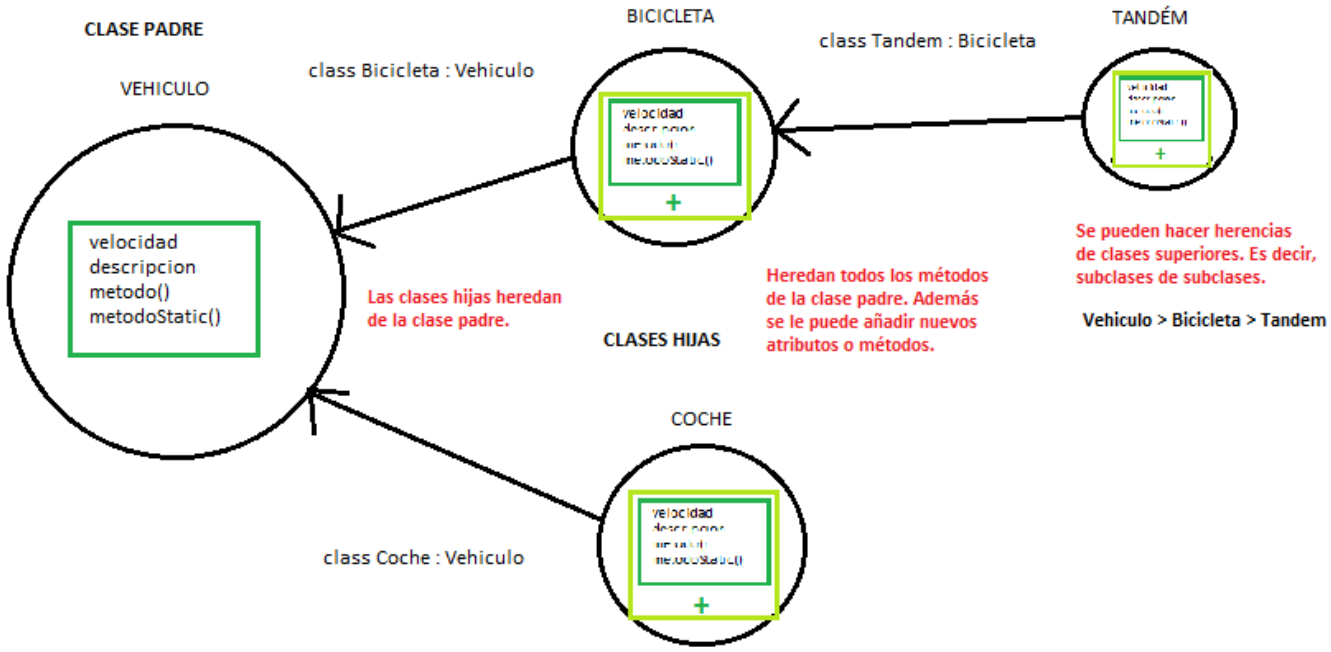
}

HERENCIA

-Las clases en Swift no heredan de la clase universal, si no se especifica una clase padre, ésta no heredará de ninguna otra.

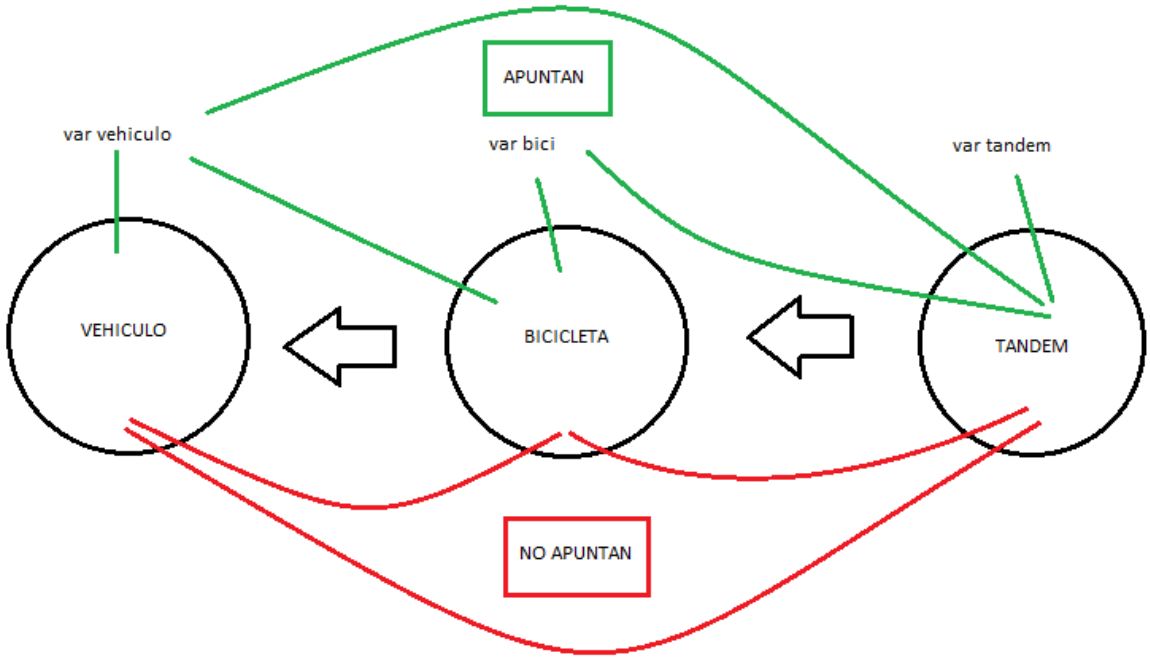
-Se pueden heredar tanto métodos como atributos de una clase.

-Heredan todos los métodos de la clase padre. Además se le puede añadir nuevos atributos o métodos.



-*OJO!*, En Swift existe la inferencia de tipos(var tandem = Tandem()), es decir, tandem se elige en tiempo de ejecución de tipo Tandem.

-Las propiedades de la herencia, con una referencia de tipo superior se puede apuntar a cualquier objeto de tipo inferior, pero a la inversa no se puede.



tandem = bicicleta – ERROR

Porque tandem es de tipo Tandem

vehiculo = bicicleta

SOBREESCRITURA

La sobreescritura es cuando se tiene un método HIJO que se llama exactamente igual que el método PADRE y tiene los mismo parámetros de entrada. Para indicar que se esta sobreescribiendo el método hay incluir la palabra reservada 'override', en cambio en JAVA se hace de manera automática.

```
class Tractor : Vehículo{
    override var descripción : String{
        print("Soy un tractor amarillo")
    }
    override func método(){
        print("Chu chu!")
    }
```

*Se pueden sobreescribir las propiedades observadoras.

```
override var velocidad: Double {
```

```
                willSet{
                        print("ahora será un valor nuevo")
                }
                didSet{
                        print("valor viejo")
                }
        }
}
```

-Por lo contrario, si queremos conseguir que un método o una propiedad no se sobreescriba se coloca la palabra reservada 'final'

JAVASCRIPT

Es un lenguaje de programación de entorno cliente, es decir, se ejecuta en los navegadores. Existen 3 maneras de poner JS en una página web:

1) Dentro de las etiquetas de HTML <p>
2) Dentro de las etiquetas <script>. Si el <script> se encuentra dentro del HEAD, se ejecutará cuando se carga la página web.
3) Importando un fichero js en el HEAD con <script>

-Si en el BODY se quiere escribir directamente en el HTML, se hace mediante:

```
document.write();
```

VARIABLES

-Interpretado

-Secuencial

-Orientado a objetos. No tipado, es decir puede apuntar a cualquier referencia.

-Funcional, orientado a funciones.

-Existen 5 tipos de variables básicos: enteros, decimales, cadenas de texto, booleanos y arrays.

-Para imprimir valores se puede usar, pulsando F12 en el navegadores se puede ver la consola:

console.log();

-Se puede concatenar un String

console.log("Hola"+ nombre);

-Se puede expandir variables

console.log('Expandir nombre: $(nombre)');

Crear un formulario en JS:

```
<!DOCTYPE html>
<html lang = "en">
<head>
  <meta charset ="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge, chorme=1">
  <meta name="viewport" content="width=device-widht, initial-scale=1.0">
  <title>Formulario</title>
  <script>
function escribirEnCaja(){
```

-Control + F5 Refresca el explorador

```
    console.log("Comprobar error")
```

-Manera clásica de acceder a un objeto del árbol DOM

-Desde las últimas versiones JS nos crea una variable automáticamente con cada id encuentre en la página.

```
    var valor = document.getElementById("procesar");
    var resultado = document.getElementById("resultado");
```

-El atributo "value" sirve para acceder al valor

```
    resultado.value = valor.value  >  resultado.value = nombre.value;
}
function add(valor1,valor2){
```

-Si la función tiene return devuelve un valor, sino es como si fuera un NULL

```
    return valor1 + " " + valor2;
  }
```

```
        function concantenarElementos(){
            resultado = add(nombre.value, apellidos.value);

        }

        function cambiarColor(){
          -Calcular aleatoriamente los colores
              var colorRojo = Math.random() * (255);
              var colorAzul = Math.random() * (255);
              var colorVerde = Math.random() * (255);

            -Función rgb
            resultado.style.backgroundColor = "rgb(" + colorRojo + "," +  colorVerde
              + "," +  colorAzul + ")";
                  }
    </script>
</head>
<body>
    <div style="text-align: center;">
        Nombre: <input type="text" id="nombre" value="lidia"/>
        <br/>
        Apellidos: <input type="text" id="apellidos"/>
        <br/>
        <button  id = “procesar” onclick="escribirEnCaja()">Procesar</button>
        <br/>
        <button onclick="concantenarElementos()">Concatenar</button>
        <br/>
        <button onclick="cambiarColor()">Cambiar de color el texto:</button>
        <br/>
        <input type="text" id="resultado"/>
```

```
    </div>


</body>
</html>
```

-Para acceder a valores de tipo RADIO se hace mediante: *document.getElementByName()* , y con el método *.checked* para comprobar si el botón radio se ha clickado.

-Para hacer un SELECT:

var lista = document.getElementById("opcionesLista");

-Obtener el índice de la opción seleccionada

var indice = lista.selectedIndex;