POLITÉCNICA

ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Persistencia

IWEB 2019-2020
Santiago Pavón

ver: 2019.12.02

Persistencia

- Conservar datos:
  - Aunque se pare y relance la aplicación.
  - Aunque apague y encienda el terminal.
- Existen varias formas de guardar datos:
  - User Defaults (preferencias de usuario)
  - Sistema de Ficheros
  - Core Data
  - Cloud
  - ...

Preferencias de Usuario

Preferencias de Usuario

- Las preferencias de usuario son valores persistentes usados por la aplicación.
- Pueden modificarse:
  - desde la propia aplicación.
  - desde la aplicación **Ajustes** (**Settings**).
    - Para poder modificar las preferencias desde **Ajustes**:
      - La aplicación debe tener un **settings bundle**.
        - conjunto de ficheros describiendo los datos de preferencias.
      - **Ajustes** crea un GUI para editar los datos.
- **UserDefaults**
  - Es la clase usada para almacenar/recuperar los valores de las preferencias.
    - Cada valor está asociado a una clave.

Acceder desde nuestra aplicación

- **UserDefaults** implementa un singleton

```
var def: UserDefaults = UserDefaults.standard
```

- Pueden guardarse combinaciones de:
  - Data, String, Int, Double, Float, Date, Array, Dictionary, URL, Bool.
  - NSData, NSString, NSNumber, NSDate, NSArray, NSDictionary, NSURL.
  - Otros tipos de datos pueden guardarse si se serializan, por ejemplo en un Data.
- Se usa como un diccionario:
  - para obtener datos:

```
func object(forKey defaultName: String) -> Any?
func integer(forKey defaultName: String) -> Int
func bool(forKey defaultName: String) -> Bool
...
```

  - para salvar datos:

```
func set(_ value: Any?, forKey defaultName: String)
func set(_ value: Int, forKey defaultName: String)
func set(_ value: Bool, forKey defaultName: String)
...
```

  - Invocar `synchronize() -> Bool` para salvar los datos de la cache que no se hayan salvado aun.
    - Este método se llama automáticamente periódicamente

Cuando cargar / salvar datos

- Dependiendo de si estamos desarrollando con UIKit o SwiftUI , se hará en sitios y de forma diferente.
  - En SwiftUI:
    - El UI simplemente muestra el estado de la app, luego solo hay que preocuparse del acceso a las preferencias cuando se cambie o se consulte el estado.
  - En UIKit:
    - Hay que preocuparse de más detalles de implementación.
      - Acceder a las preferencias cuando se cambia algún dato en algún sitio, o cuando se consulta.
      - Cuando la pantalla va a mostrarse o ocultarse (viewWillAppear viewWillDisappear).
      - Al cargar una pantalla en memoria (viewDidLoad).
      - En los métodos del delegado de la aplicación.
        - Se llaman al pasar a segundo plano, cuando va a terminar la app, …
      - …

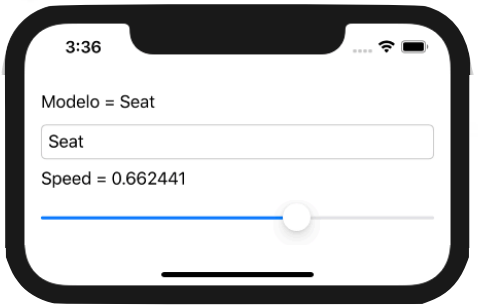
Ejemplo

```
private let defaults = UserDefaults.standard

struct ContentView: View {

    @State var model: String = ""
    @State var speed: Double = 0

    var body: some View {

        let bindingModel = Binding(
            get: { self.model },
            set: { defaults.set($0, forKey: "model")
                   self.model = $0 })
        let bindingSpeed = Binding(
            get: { self.speed },
            set: { defaults.set($0, forKey: "speed")
                   self.speed = $0 })

        return VStack(alignment: .leading) {
            Text("Modelo = \(model)")
            TextField("Model", text: bindingModel)
                .textFieldStyle(RoundedBorderTextFieldStyle())
            Text("Speed = \(speed)")
            Slider(value: bindingSpeed)
        }
        .padding(.horizontal)
        .onAppear(perform: {
            self.model = defaults.object(forKey: "model") as? String ?? "Ferrari"
            self.speed = defaults.double(forKey: "speed")
        })
    }
}
```

Preferencias del usuario

Estado

Bindings personalizados que guardan los valores asignados en las preferencias

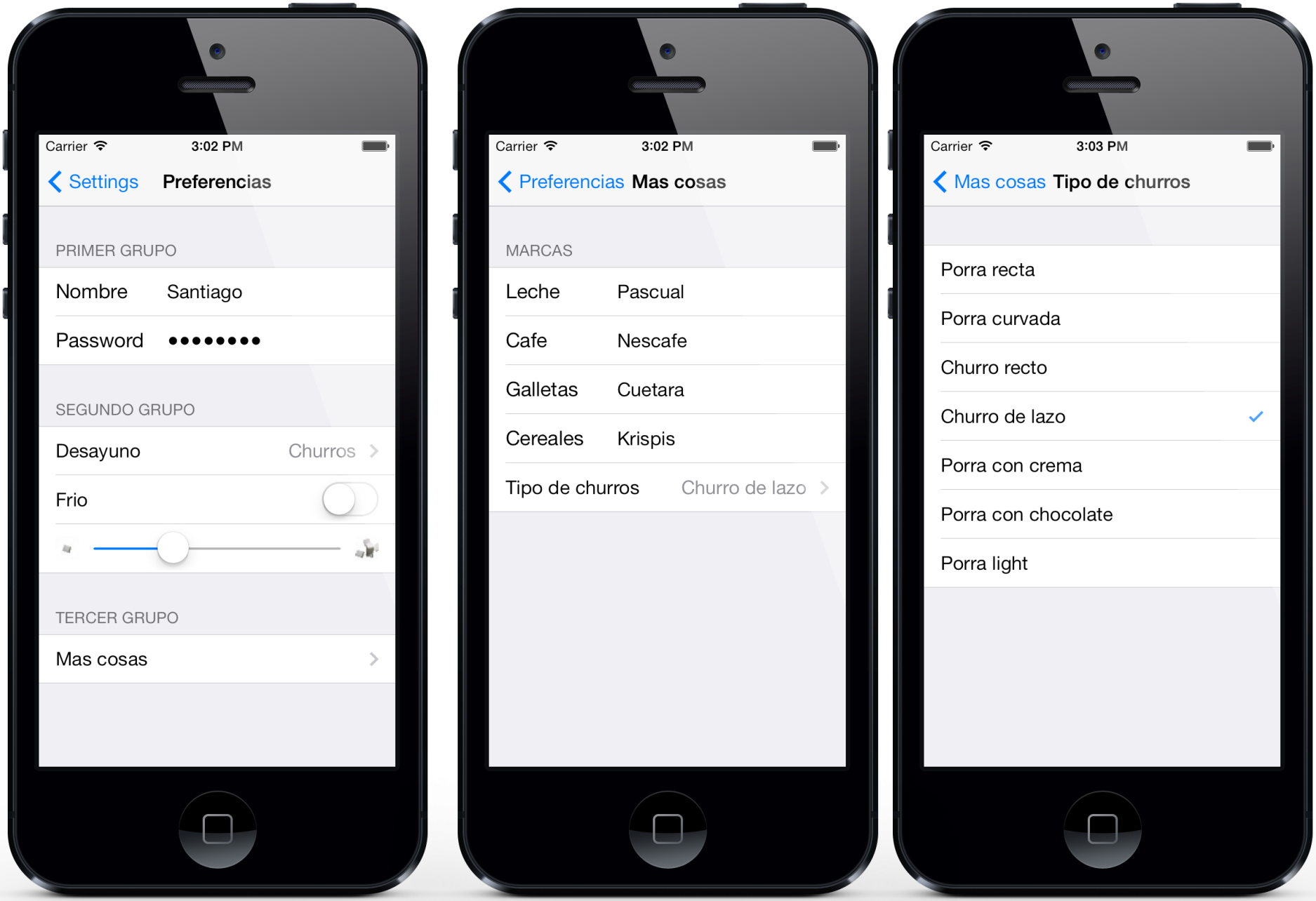


Cargar las preferencias cuando aparece la pantalla.

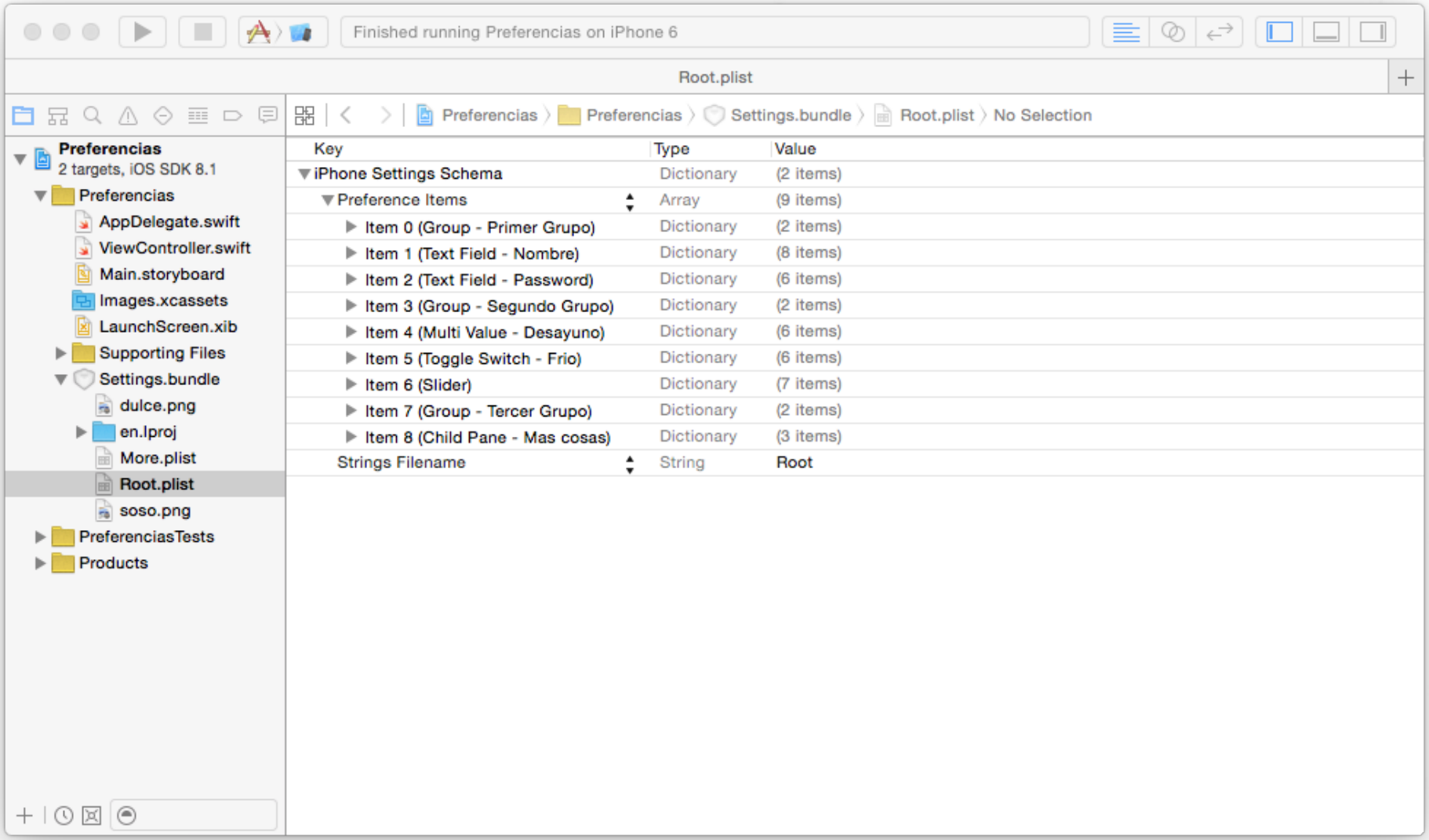
Acceso desde la Aplicación **Ajustes**

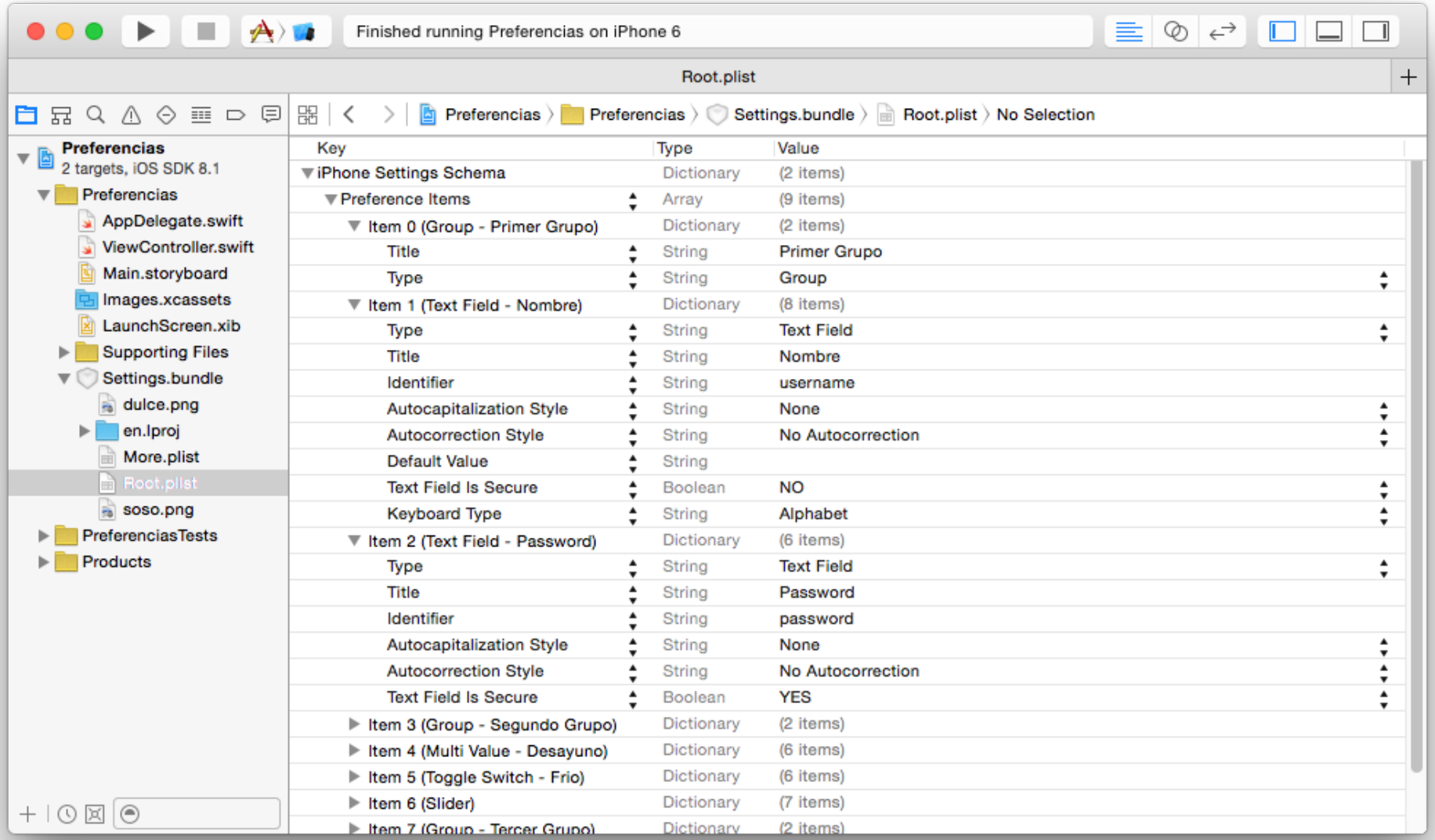
- Las preferencias de usuario pueden editarse desde la aplicación **Ajustes** (**Settings**).
  - Solo pueden editarse algunos tipos de valores.
- Para ello, hay que crear en nuestra aplicación un **Settings Bundle**:

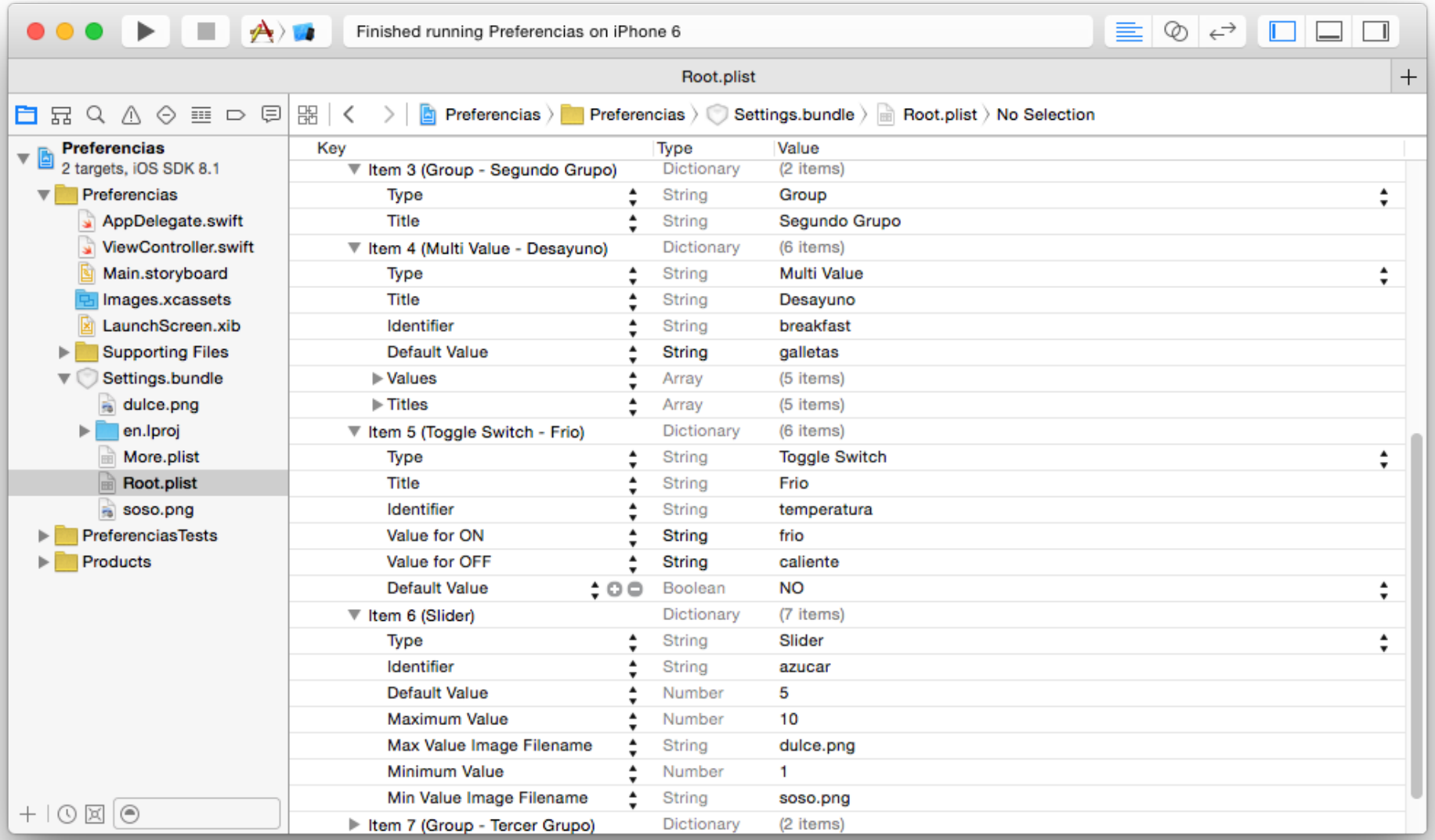
  New File > iOS > Resource > Settings Bundle
  - **Para manipular el contenido de este fichero hay que usar Finder.**
    - **Por ejemplo, para añadir un imagen.**
- **`Root.plist`**
  - define la primera vista de las preferencias.
- Para crear subvistas adicionales (nuevas pantallas) hay que crear nuevos ficheros de listas de propiedades.
  - Un fichero **plist** para cada pantalla adicional.
- Consultar la guía:
  - **Preferences and Settings Programming Guide: About Preferences and Settings**.

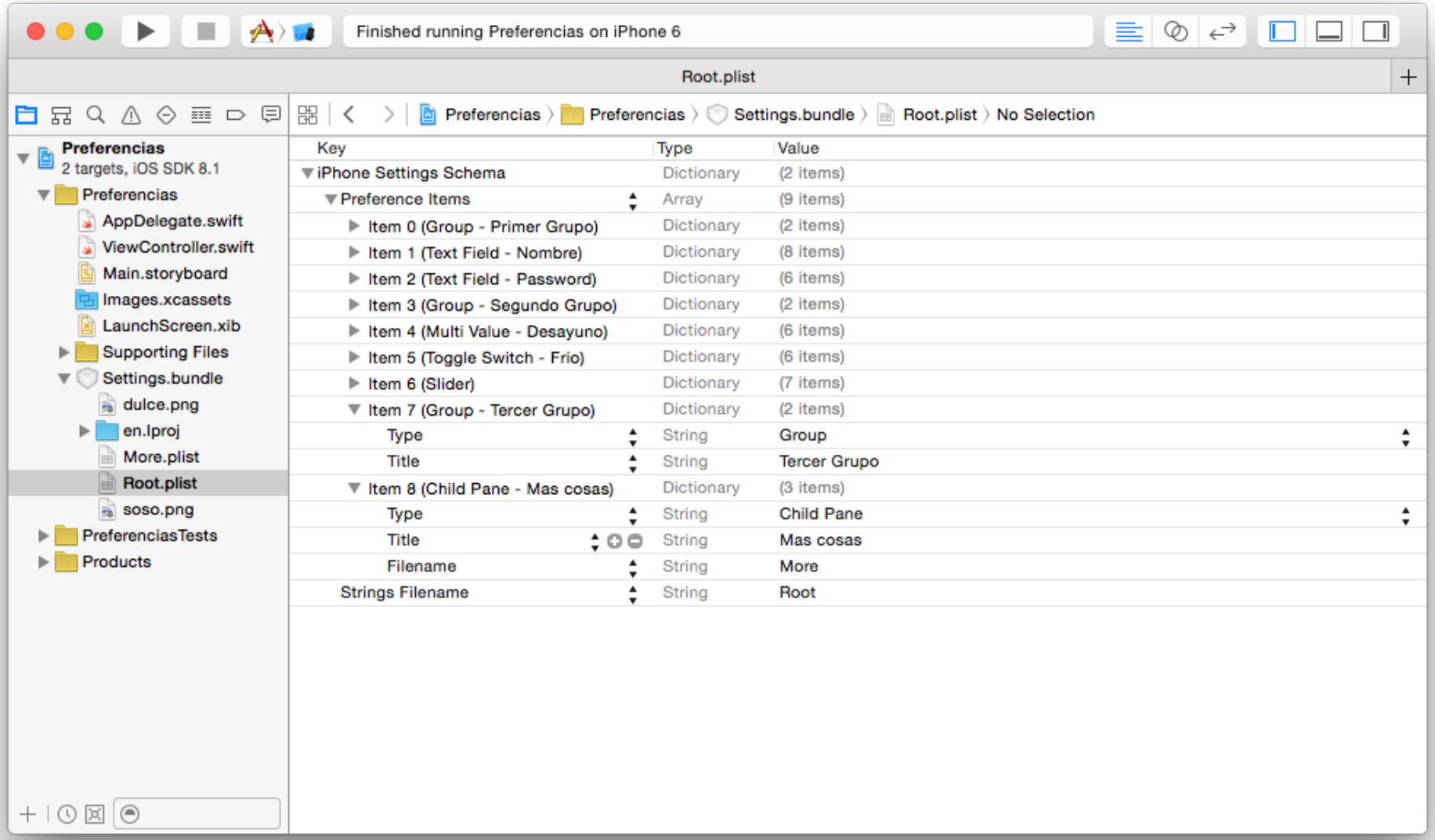


Root.plist

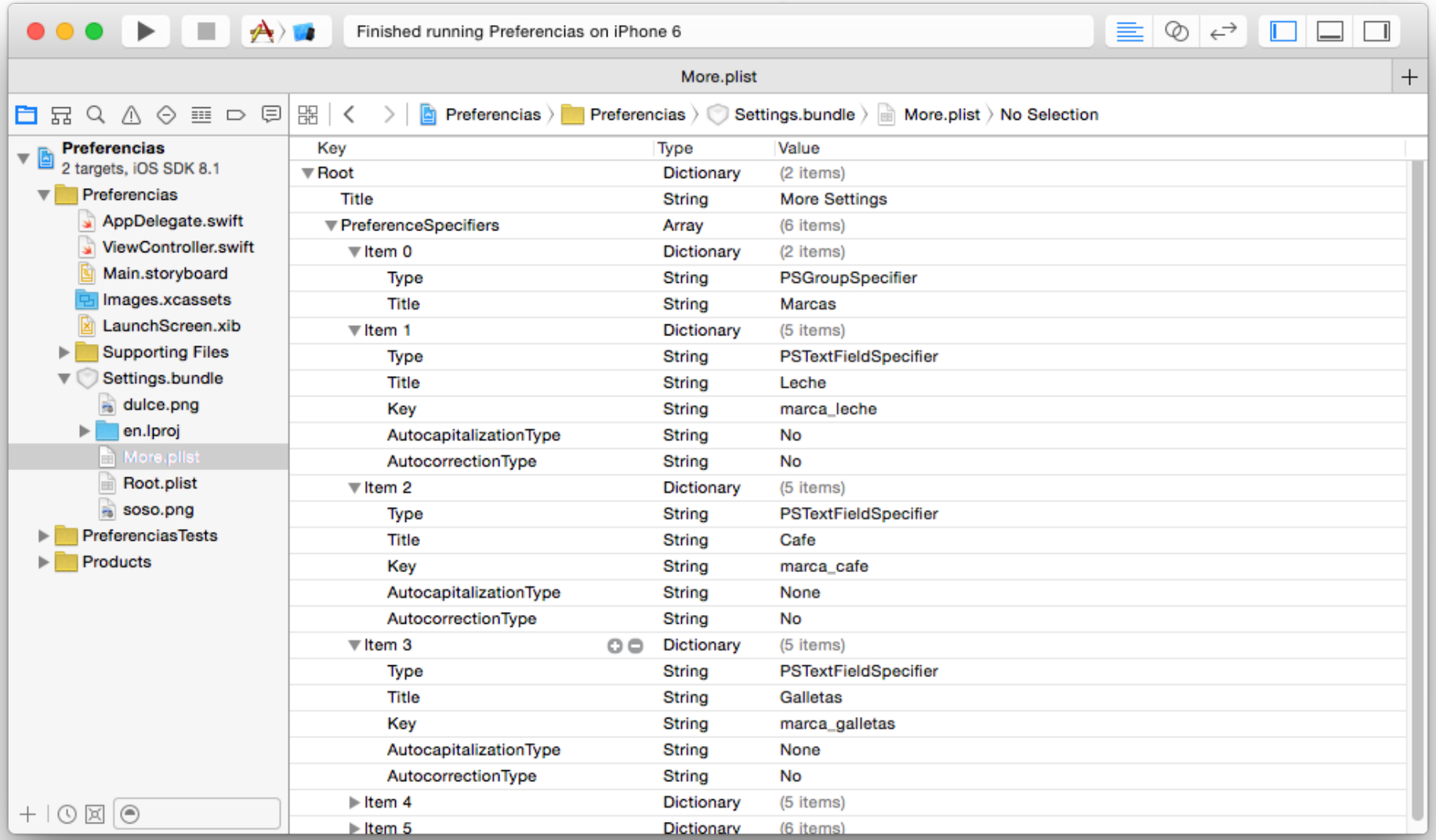


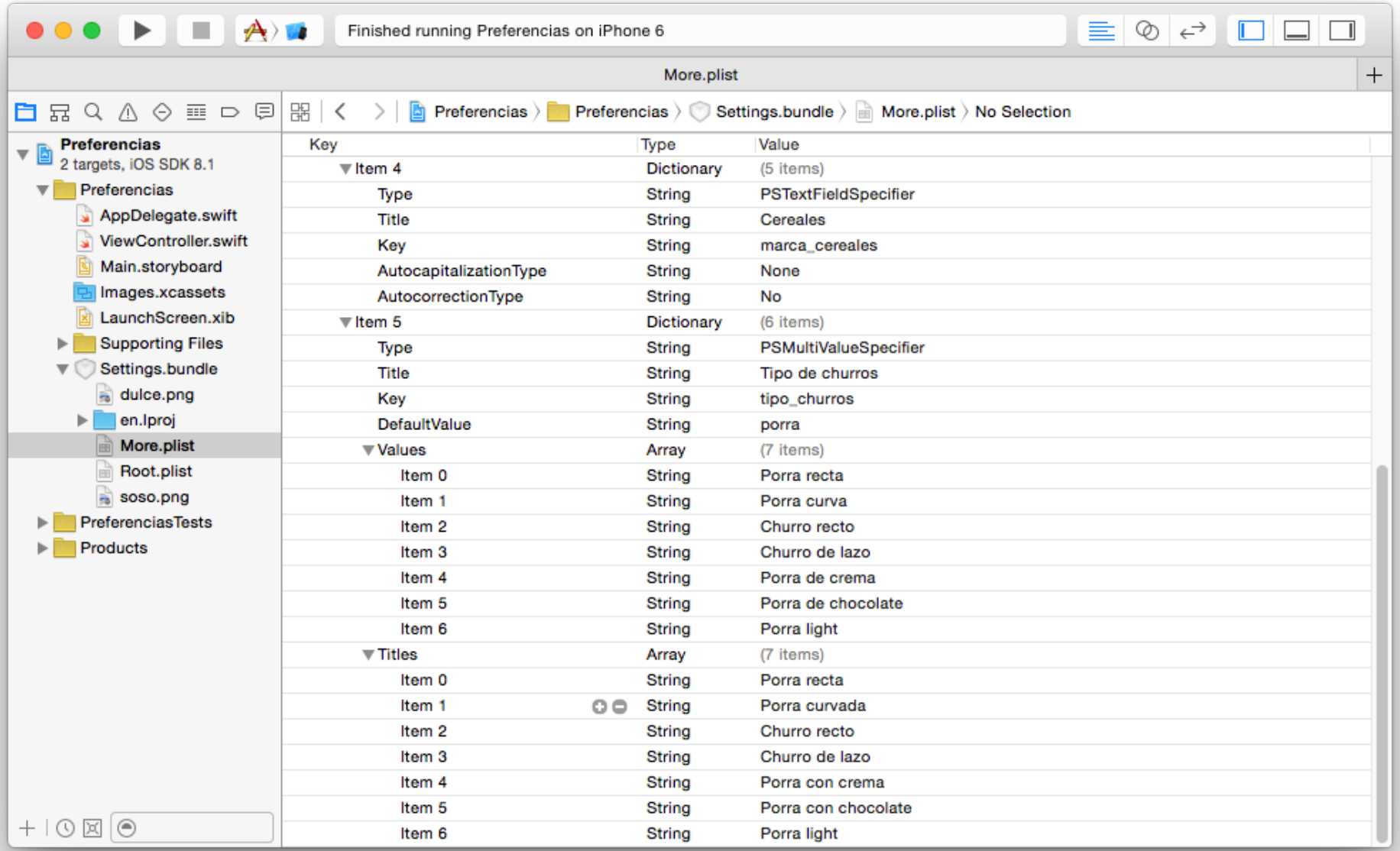






More.plist





Main Bundle

Bundle

- Un objeto **Bundle** representa un lugar del sistema de ficheros.
  - Carpeta donde se guardan recursos, código, etc.
- Las aplicaciones y frameworks son bundles.
- El **main bundle** de una aplicación permite acceder a los recursos que se añadieron en el proyecto.
- Estos objetos están firmados,
  - no pueden modificarse.

Usar el Main Bundle

```
let bundle: Bundle = Bundle.main

let path: String? = bundle.path(forResource: "pokemons",
                                ofType: "plist")

let url: URL? = bundle.url(forResource: "pokemons",
                           withExtension: "plist")

let img: UIImage? = UIImage(named: "foto.jpg")
```

Sistema de Ficheros

Sistema de Ficheros

- Las aplicaciones ven un sistema de ficheros UNIX.
- Las aplicaciones corren en un Sandbox.
  - La ejecución de un programa no daña a otros.
  - Proteger acceso a los datos de una aplicación.
  - Fácil borrar datos al desinstalar una aplicación.
- Contenido del sandbox:
  - directorio del bundle de aplicación. (sólo lectura).
  - directorio Documents. (donde salvar los datos permanentes).
  - directorio de caches. (temporales sin backup de iTunes).
  - . . .

Obtener Rutas a Directorios

- Directorio Home:

```
let home: String = NSHomeDirectory()
```

- Directorio Temporal:

```
let tmp: String = NSTemporaryDirectory()
```

- Directorio Documents:

```
let paths = NSSearchPathForDirectoriesInDomains(
                    .documentDirectory,
                    .userDomainMask,
                    true)
let docsPath: String = paths[0]
```

Manipular Rutas

- Consultar la documentación de las clases **NSString** y **NSURL**.

```
// Directorio de documentos:
let paths = NSSearchPathForDirectoriesInDomains(.DocumentDirectory,
                                                .UserDomainMask, true)
let docsPath = paths[0] as NSString

// Añadir al path:
let datosPath: NSString = docsPath.appendingPathComponent("g.dat") as NSString

// La extension de un fichero:
let ext = datosPath.pathExtension

// Nombre del fichero:
let fileName: NSString = datosPath.lastPathComponent as NSString

// Nombre del fichero sin extension:
let fileBasename = fileName.deletingPathExtension

// Directorio del fichero:
let basedir = datosPath.deletingLastPathComponent
```

```
// Directorio de documentos:
let paths = NSSearchPathForDirectoriesInDomains(.DocumentDirectory,
                                                .UserDomainMask, true)
let docsPath = paths[0]

// URL del directorio de Documentos:
let docsURL: URL? = URL(fileURLWithPath: docsPath)

// Añadir al URL:
let datosURL: NSURL = docsURL!.appendingPathComponent("g.dat")

// Absolute string:
let absPath: String? = datosURL.absoluteString

// Escapar queryURL: URL encoding
let esc: String? = "a=b c".addingPercentEscapes(
                                withAllowedCharacters: .urlQueryAllowed)
                            // a=b%20c
```

FileManager

- Proporciona métodos para:
  - ver si un fichero existe.
  - crear y examinar directorios.
  - manipular ficheros: copiar, mover, borrar.
  - comparar ficheros.
  - obtener URL de directorios del sistema.
  - ...
- Consultar la documentación para ver todos los métodos disponibles.

Ejemplo

- Copiar un fichero desde el Bundle de la aplicación al directorio de documentos:

```
// Crear un File Manager
let fm = FileManager()

// Fichero origen
let bundle = Bundle.main
guard let origenURL = bundle.url(forResource: "pokemons", withExtension:"plist")
      else { return }

// Fichero destino (en el directorio de documentos)
let docsURLs = fm.urls(for: .DocumentDirectory, in: .UserDomainMask)
let docsURL = docsURLs[0]
let destinoURL = docsURL.appendingPathComponent("pokemons.plist")

// Copiar
do {
   try fm.copyItem(at: origenURL, to: destinoURL)

   // Usar el fichero copiado
   // . . .
} catch let error {
   print(error)
}
```

Lista de Propiedades

Lista de Propiedades

- Es un **dato** formado por cualquier combinación de los tipos:
  - **NSArray**, **NSDictionary**, **NSData**, **NSString**, **NSNumber**, **NSDate**.
  - y los relacionados de Swift: **Array**, **Dictionary**, **Data**, **String**, **Int**, **Double**, **Float**, **Bool**.
- Se pueden guardar y recuperar de ficheros **.plist**.

```
var dic = NSDictionary(contentsOfFile:path1) as! [String:String]
dic["clave"] = "valor"
(dic as NSDictionary).writeToFile(path1, atomically:true)


var arr = NSArray(contentsOfFile:path2) as! [Int]
arr += 100
(arr as NSArray).writeToFile(path2, atomically:true)
```

PropertyListSerialization

- La clase **PropertyListSerialization** proporciona métodos de tipo para:
  - serializar una lista de propiedades en un **Data**.

```
class func data(fromPropertyList plist: Any,
                format: PropertyListSerialization PropertyListFormat,
                options opt: PropertyListSerialization.WriteOptions)
                                    throws -> Data
```

  - y crear una lista de propiedades desde un **Data**.

```
class func propertyList(from data: Data,
                options opt: PropertyListSerialization.WriteOptions = [],
                format: UnsafeMutablePointer<
                        PropertyListSerialization.PropertyListFormat>?)
                                    throws -> Any
```

- Los objetos **Data** se pueden leer y escribir en ficheros usando:

```
func write(to url: URL, options: Data.WritingOptions = default) throws
init(contentsOf url: URL, options: Data.ReadingOptions = default) throws
. . .
```

- La clase **NSData** tiene más métodos para leer y escribir en ficheros.

Protocolo **`Codable`**

Codificación/Decodificación

- Conversión de los datos entre los tipos del lenguaje Swift y otros formatos externos de representación (ej: JSON, listas de propiedades, XML, ...).
  - Para guardar/recuperar datos en ficheros, para transmitirlos/recibirlos por un socket, descargas/subidas a un sitio web, etc.
- Swift es un lenguaje fuertemente tipado, y los formatos externos no lo son (normalmente).
  - Estas conversiones adaptan el tipo de los datos adecuadamente.
    - Ejemplo: Las fechas se adaptan entre su representación como un String en JSON y como el tipo Date en Swift.
- El lenguaje Swift, el compilador, la librería estándar, Foundation, etc. permiten programar estas conversiones de manera fácil, y también permiten configurar/personalizar estas conversiones.

Protocolo Codable

- Los tipos Swift (enumerados, structs, clases) deben adoptar el protocolo **Codable** para que puedan ser codificados/decodificados.
- Muchos tipos predefinidos en los frameworks existentes ya son conformes a Codable.
  - Foundation -> Date, Data, URL, IndexPath, NSRange, Decimal, CGFloat, AffineTransform, Calendar, Locale, ...
- El protocolo **Codable** es la unión de dos protocolos: **Encodable** y **Decodable**.

```
typealias Codable = Encodable & Decodable
```

  - El protocolo **Encodable** declara una función para codificar los tipos Swift a un formato externo.

```
protocol Encodable {
    func encode(to encoder: Encoder)
}
```

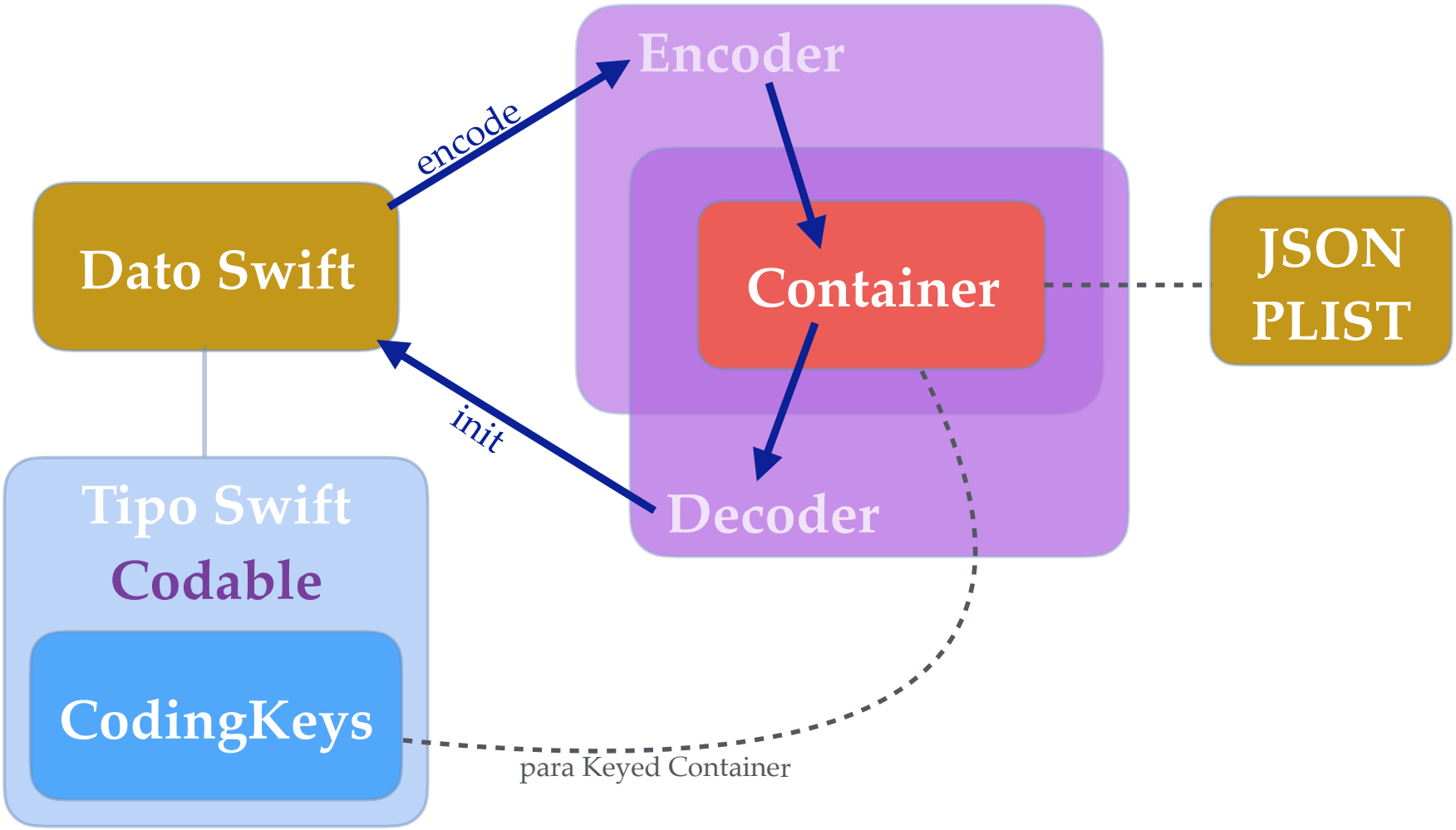
    - Este método guarda las propiedades en el objeto pasado como parámetro.
    - Encoder es un protocolo.
  - El protocolo **Decodable** declara un inicializador para reconstruir tipos Swift desde un formato externo:

```
protocol Decodable {
   init(from decoder: Decoder) throws
}
```

    - Este método inicializa un objeto con las propiedades recuperadas del parámetro Decoder.
    - Decoder es un protocolo.
- El compilador genera implementaciones por defecto de estos protocolos para nuestros tipos,
  - Debemos modificar solo las partes en las que queramos personalizar algo.

Foundation

- El framework Foundation proporciona varias clases y protocolos usados para archivar datos Swift.
  - **JSONEncoder** y **JSONDecoder**
    - Las instancias de estas clases codifican los datos Swift en el formato externo JSON, e inversa.
  - **PropertyListEncoder** y **PropertyListDecoder**
    - Las instancias de estas clases codifican los datos Swift como listas de propiedades, e inversa.
  - **JSONSerialization** y **PropertyListSerialization**
    - Serializar listas de propiedades en buffers de bytes (Data), conversión entre JSON y listas de propiedades, etc.
  - Otros:
    - **NSCoding**, **NSCoder**, **NSKeyedArchiver**, **NSKeyedUnarchiver**, ...



Ejemplo 1.1 JSON

```
let str = """
        { "name": "Pepe",
          "address": "Rue del Percebe, 13",
          "age": 33
        }
        """

struct Person : Codable {
        let name: String
        let address: String
        let age: Int
}

let jsonData = str.data(using: .utf8)!

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p.name)    // Pepe
    print(p.age)     // 33
}
```

Crear JSONDecoder para transformar un string con un JSON en un tipo Swift. En el JSON solo hay strings y números.

Ejemplo 1.2 JSON

```
let str = """
    { "name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33,
      "birthday": "2017-07-26T16:00:49Z"
    }
    """

struct Person : Codable {
    let name: String
    let address: String
    let age: Int
    let birthday: Date
}

let jsonData = str.data(using: .utf8)!

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601

if let p = try? decoder.decode(Person.self, from: jsonData) {
   print(p.birthday)   // 2017-07-26 16:00:49 +0000
}
```

JSON no tiene un tipo para representar fechas. Puede usarse un String, un número con los segundos desde una fecha de referencia, ... En este ejemplo, el JSON usa un String en formato ISO8601 para la fecha.

Ejemplo 1.3 JSON

```
let str = """
    { "name": "Pepe", "address": "Rue del Percebe, 13", "age": 33,
      "birthday": "2017/07/26"
    }
    """
struct Person : Codable {
    let name: String
    let address: String
    let age: Int
    let birthday: Date
}
let jsonData = str.data(using: .utf8)!

let decoder = JSONDecoder()
let f = DateFormatter()
f.dateFormat = "yyyy-MM-dd"
f.timeZone = TimeZone(secondsFromGMT: 0)
decoder.dateDecodingStrategy = .formatted(f)

if let p = try? decoder.decode(Person.self, from: jsonData) {
   print(p.birthday)   // 2017-07-26 00:00:00 +0000
}
```

En el JSON hay un string con una fecha en un formato personalizado.

Otros formatos:

- .deferredToDate
- .millisecondsSince1970
- .secondsSince1970
- ...

Ejemplo 1.4 JSON

```
let str = """
          { "name": "Pepe",
            "genre": "male"
          }
          """

struct Person : Codable {
    let name: String
    let genre: Genre
}

enum Genre: String, Codable {
    case male
    case female
}

let jsonData = str.data(using: .utf8)!

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p.name)   // Pepe
    print(p.genre)  // male
}
```

En el JSON hay un campo string que decodificamos como un enum. Solo hay dos posibles valores para ese campo.

Ejemplo 1.5 JSON

```
let jsonData = """
               { "a": "NaN",
                 "b": "+Infinity",
                 "c": "-Infinity",
                 "d": 2.5
               }
               """.data(using: .utf8)!
struct Values : Codable {
    let a: Float
    let b: Float
    let c: Float
    let d: Float
}
let decoder = JSONDecoder()
decoder.nonConformingFloatDecodingStrategy =
                .convertFromString(positiveInfinity: "+Infinity",
                                   negativeInfinity: "-Infinity",
                                   nan: "NaN")
if let p = try? decoder.decode(Values.self, from: jsonData) {
    print(p.a)   // nan
    print(p.b)   // inf
    print(p.c)   // -inf
    print(p.d)   // 2.5
}
```

Configurar la decodificación de números Float y Double inválidos.
Proporcionamos una representación como Strings.
Es igual para codificar.

Ejemplo 1.6 JSON

Configurar la decodificación de datos binarios (Data) representados en Base 64. Es igual para codificar.

```
let jsonData = """
               { "data": "Y2FtacOzbg=="
               }
               """.data(using: .utf8)!

struct Values : Codable {
    let data: Data
}

let decoder = JSONDecoder()

decoder.dataDecodingStrategy = .base64

if let p = try? decoder.decode(Values.self, from: jsonData),
    let s = String(data: p.data, encoding: .utf8) {
    print(s)   // camión
}
```

Ejemplo 1.7 JSON

```
let jsonData = """
    [ { "name": "Pepe", "age": 33 },
      { "name": "Ana",  "age": 32 },
      { "name": "Luis", "age": 37 }
    ]
   """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let age: Int
}

let decoder = JSONDecoder()

if let p = try? decoder.decode([Person].self, from: jsonData) {
    print(p) // [__lldb_expr_426.Person(name: "Pepe", age: 33),
             //  __lldb_expr_426.Person(name: "Ana", age: 32),
             //  __lldb_expr_426.Person(name: "Luis", age: 37)]
}
```

Decodificar un array de instancias de un tipo Swift.

Ejemplo 1.8 JSON

```
let jsonData = """
    { "padre": { "name": "Pepe", "age": 33 },
      "madre": { "name": "Ana",  "age": 32 },
      "hijo":  { "name": "Luis", "age": 37 }
    }
   """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let age: Int
}

let decoder = JSONDecoder()

if let p = try? decoder.decode([String:Person].self, from: jsonData) {
    print(p) // [ "hijo": __lldb_expr_500.Person(name: "Luis", age: 37),
             //   "padre": __lldb_expr_500.Person(name: "Pepe", age: 33),
             //   "madre": __lldb_expr_500.Person(name: "Ana", age: 32) ]
}
```

Decodificar un diccionario Dictionary<String,Person>

Ejemplo 1.9 JSON

```
let jsonData = """
    { "name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

// No se ha definido el atributo address
struct Person : Codable {
    let name: String
    let age: Int
}

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p)  // Person(name: "Pepe", age: 33)
}
```

Se ignoran los campos del JSON que no se definen en el tipo Swift.

Ejemplo 1.10 JSON

```
let jsonData = """
    { "person": { "name": "Pepe", "address": "Rue del Percebe, 13",
                  "age": 33, "birthday": "2017-07-26T16:00:49Z"
                },
      "email": "pepe@dominio.es",
      "web": "https://dominio.es/pepe"
    }
    """.data(using: .utf8)!

struct Contact : Codable {
    struct Person : Codable {
        let name: String
        let address: String
        let age: Int
        let birthday: Date
    }
    let person: Person
    let email: String
    let web: URL
}

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601

if let contact = try? decoder.decode(Contact.self, from: jsonData) {
    print(contact.email)          // pepe@dominio.es
    print(contact.person.name)    // Pepe
}
```

JSON con tipos anidados.

Un campo es una URL.

Ejemplo 1.11 JSON

```
let jsonData = """
    { "person": { "name": "Pepe", "address": "Rue del Percebe, 13",
                  "age": 33, "birthday": "2017-07-26T16:00:49Z"
                },
      "email": "pepe@dominio.es",
      "web": "https://dominio.es/pepe"
    }
    """.data(using: .utf8)!
struct Contact : Codable {
    struct Person : Codable {
        let name: String
        let address: String
        let age: Int
        let birthday: Date
    }
    let person: Person
    let email: String
    let web: URL
}
let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601
```

Decodificar y Codificar en JSON.

```
if let contact = try? decoder.decode(Contact.self, from: jsonData) {
      print(contact.email)         // pepe@dominio.es
      print(contact.person.name)   // Pepe

     let encoder = JSONEncoder()
     encoder.dateEncodingStrategy = .iso8601

     if let data = try? encoder.encode(contact),
        let str = String(data: data, encoding: .utf8) {
        print(str) // {"email":"pepe@dominio.es","web":"https:\/\/dominio.es\/pepe",
                   // "person":{"age":33,"name":"Pepe","birthday":
                   // "2017-07-26T16:00:49Z", "address":"Rue del Percebe, 13"}}
     }
}
```

Ejemplo 1.12 JSON

```
let jsonData = """
    { "person": { "name": "Pepe", "address": "Rue del Percebe, 13",
                  "age": 33, "birthday": "2017-07-26T16:00:49Z"
                },
      "email": "pepe@dominio.es",
      "web": "https://dominio.es/pepe"
    }
    """.data(using: .utf8)!
struct Contact : Codable {
    struct Person : Codable {
        let name: String
        let address: String
        let age: Int
        let birthday: Date
    }
    let person: Person
    let email: String
    let web: URL
}
let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601
```

La salida de la codificación es muy compacta y muy difícil de leer.
Configuramos el encoder para que use un formato de salida más fácil de leer.

```
if let contact = try? decoder.decode(Contact.self, from: jsonData) {
      print(contact.email)        // pepe@dominio.es
      print(contact.person.name)  // Pepe

      let encoder = JSONEncoder()
      encoder.dateEncodingStrategy = .iso8601
      encoder.outputFormatting = .prettyPrinted

      if let data = try? encoder.encode(contact),
         let str = String(data: data, encoding: .utf8) {
         print(str) // {
                    //    "email" : "pepe@dominio.es",
                    //     "web" : "https:\/\/dominio.es\/pepe",
                    //     "person" : {
                    //       "age" : 33,
                    //       "name" : "Pepe",
                    //       "birthday" : "2017-07-26T16:00:49Z",
                    //       "address" : "Rue del Percebe, 13"
                    //     }
                    //    }
      }
}
```

Ejemplo 1.13 JSON

```
let jsonData = """
    { "name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

struct Person : Codable {
    let name: String              // HEMOS OMITIDO address
    let age: Int
}

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {

    print(p)   // Person(name: "Pepe", age: 33)

    let encoder = JSONEncoder()

    if let data = try? encoder.encode(p),
        let str = String(data: data, encoding: .utf8) {

        print(str)  // {"name":"Pepe","age":33}
    }
}
```

Para no decodificar un atributo, lo omitimos en el tipo Swift.

Ejemplo 1.14 JSON

Los campos que pueden faltar ocasionalmente en el JSON deber ser opcionales en el tipo Swift.

```
let jsonData = """
    { "name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let address: String?  // Opcional, pero si esta en el JSON de este ejemplo
    let age: Int
    let other: Int?       // Opcional y no esta en el JSON de este ejemplo
}

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p)   // Person(name: "Pepe", address: Optional("Rue del Percebe, 13"),
               //        age: 33, other: nil)

    let encoder = JSONEncoder()

    if let data = try? encoder.encode(p),
       let str = String(data: data, encoding: .utf8) {
        print(str)  // {"name":"Pepe","address":"Rue del Percebe, 13","age":33}
    }
}
```

Ejemplo 1.1 PropList

```
let plist: Dictionary<String,Any> = [
    "name": "Pepe", "address": "Rue del Percebe, 13", "age": 33
]
let plistData = try! PropertyListSerialization.data(fromPropertyList: plist,
                                                    format: .xml,
                                                    options: 0)

struct Person : Codable {
    let name: String
    let address: String
    let age: Int
}

let decoder = PropertyListDecoder()
if let p = try? decoder.decode(Person.self, from: plistData) {
    print(p) // Person(name: "Pepe", address: "Rue del Percebe, 13", age: 33)

    let encoder = PropertyListEncoder()
    if let data = try? encoder.encode(p),
        let dic = try? PropertyListSerialization.propertyList(from: data,
                                                              options: [], format: nil) {
        print(dic) // {
    }              //   address = "Rue del Percebe, 13";
}                  //   age = 33;
                   //   name = Pepe;
                   // }
```

Decodificar y Codificar una Lista de Propiedades.

CodingKeys

- Esto solo aplica cuando los **Encoder**s y **Decoder**s usan Keyed contenedores.
  - En el tipo Swift a archivar hay que crear el enum **CodingKeys** con las claves.
- Por defecto, el compilador genera automáticamente un enum llamado **CodingKeys** cuyos los valores son los nombres de los atributos a codificar y decodificar.
  - Por defecto, el nombre de los atributos es el mismo para los tipos Swift y el formato externo.
  - Si el nombre de algún atributo es distinto en el tipo Swift y en el formato externo, entonces hay que crear una versión personalizada de CodingKeys con la correspondencia de nombres.
  - Las propiedades no incluidas en CodingKeys no se codifican, ni se decodifican.
    - Deben tener un valor por defecto para que el tipo Swift sea conforme con Decodable, y evitar así un error de compilación.
- El tipo de los valores asociados a los cases puede ser String o Int.
  - Los tipos de valores que pueden ser usados como claves deben ser conformes con el protocolo **CodingKey**, y el enum CodingKeys lo es. Este protocolo requiere que los valores usados como claves tengan asociado un string que lo represente, y opcionalmente también un entero.
  - Ese tipo lo usan los Encoders y Decoders para decidir que tipo de clave usan al guardar/recuperar los valores en los *keyed* contenedores que usan internamente.

Ejemplo 2.1

```
let jsonData = """
    { "name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let address: String
    let age: Int

    private enum CodingKeys : String, CodingKey {
        case name
        case address
        case age
    }
}

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p)  // Person(name: "Pepe", address: "Rue del Percebe, 13", age: 33)
}
```

Este es el enum CodingKeys que se genera por defecto. Por defecto, el valor asociado a cada case del enum es un String con el mismo nombre del case.

Ejemplo 2.2

```
let jsonData = """
    { "full_name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let address: String
    let age: Int

    private enum CodingKeys : String, CodingKey {
        case name = "full_name"
        case address
        case age
    }
}

let decoder = JSONDecoder()

if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p)  // Person(name: "Pepe", address: "Rue del Percebe, 13", age: 33)
}
```

Hay que crearse un CodingKeys personalizado si no coinciden los nombres de los campos. Cambiando el valor asociado de los cases.

Ejemplo 2.3

Las propiedades del tipo Swift no declaradas en CodingKeys:

- No se codifican, ni se decodifican.
- Deben tener un valor por defecto para que el tipo Swift sea conforme con Decodable, y evitar que se genere un error de compilación.

```
let jsonData = """
    { "name": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let address: String
    let age: Int = 66
    // Dado que age no existe en CodingKeys,
    // entonces no se decodifica y se crea con el valor por defecto.
    private enum CodingKeys : String, CodingKey {
        case name
        case address
    }
}

let decoder = JSONDecoder()
if let p = try? decoder.decode(Person.self, from: jsonData) {
    print(p)  // Person(name: "Pepe", address: "Rue del Percebe, 13", age: 66)

    let encoder = JSONEncoder()
    if let data = try? encoder.encode(p),
        let str = String(data: data, encoding: .utf8) {
        print(str) // {"name":"Pepe","address":"Rue del Percebe, 13"}
    }
}
```

Ejemplo 2.4 PropList

```
let plist: Dictionary<String,Any> = [
    "fullName": "Pepe", "address": "Rue del Percebe, 13", "age": 33
]
let plistData = try! PropertyListSerialization.data(fromPropertyList: plist,
                                                    format: .xml,
                                                    options: 0)

struct Person : Codable {
    let name: String
    let address: String
    let age: Int
    private enum CodingKeys : String, CodingKey {
        case name = "fullName"
        case address
        case age
    }
}

let decoder = PropertyListDecoder()
if let p = try? decoder.decode(Person.self, from: plistData) {
    print(p) // Person(name: "Pepe", address: "Rue del Percebe, 13", age: 33)

    let encoder = PropertyListEncoder()
    if let data = try? encoder.encode(p),
        let dic = try? PropertyListSerialization.propertyList(from: data,
                                                              options: [], format: nil) {
        print(dic) // {
    }              //   address = "Rue del Percebe, 13";
}                  //   age = 33;
                   //   fullName = Pepe;
                   // }
```

Con Listas de Propiedades en vez de JSON.

Errores de Codificación

- Una codificación puede fallar si hay alguno de los valores a codificar no puede representarse en el formato externo.
- El enum **EncodingError** enumera las causas de error *(solo hay una causa)* que pueden aparecer al codificar un dato.
  - `.invalidValue(Any, EncodingError.Context)`
    - No se puede codificar el valor en la representación externa.

Errores de Decodificación

- El enum **DecodingError** enumera las causas de error que pueden aparecer al decodificar un dato.
  - **`.typeMismatch`**`(Any.Type, DecodingError.Context)`
    - En la representación externa del dato se usa algún tipo que no coincide con el definido en el tipo Swift.
  - **`.keyNotFound`**`(CodingKey, DecodingError.Context)`
    - Falta una propiedad en la representación externa.
  - **`.valueNotFound`**`(Any.Type, DecodingError.Context)`
    - Falta un valor en la representación externa.
  - **`.dataCorrupted`**`(DecodingError.Context)`
    - El dato a decodificar está corrupto o es inválido por alguna otra razón.

- **DecodingError** define internamente una estructura llamada **`DecodingError.Context`** para describir el contexto en el que se produjo el error.
  - Su propiedad **`codingPath`** es una ruta de CodingKeys hasta el punto donde se produjo el error.
  - Su propiedad **`debugDescription`** describe el error producido.
  - ...
- DecodingError también define varios métodos de tipo para crear instancias de este tipo de error.

Ejemplo 3.1

```
let jsonData = """
    { "XXXXXXname": "Pepe",
      "address": "Rue del Percebe, 13",
      "age": 33
    }
    """.data(using: .utf8)!

struct Person : Codable {
    let name: String
    let address: String
    let age: Int
}


let decoder = JSONDecoder()

do {
    let p = try decoder.decode(Person.self, from: jsonData)
    print(p)
}
catch {
    print("Error: \(error.localizedDescription)")
    // Error: The data couldn't be read because it is missing.
}
```

Un único catch captura todos los tipos de error, sin importarnos cuál ha sido exactamente el error que se ha producido.

Ejemplo 3.2

```
...
do {
    let p = try decoder.decode(Person.self, from: jsonData)
    print(p)
}
catch DecodingError.typeMismatch(let type, let context) {
    print("typeMismatch =>")
    print(type)
    print(context.codingPath)
    print(context.debugDescription)
}
catch DecodingError.keyNotFound(let codingKey, let context) {
    print("keyNotFound =>")
    print(codingKey)
    print(context.codingPath)
    print(context.debugDescription)
}
catch DecodingError.valueNotFound(let type, let context) {
    print("valueNotFound =>")
    print(type)
    print(context.codingPath)
    print(context.debugDescription)
}
catch DecodingError.dataCorrupted(let context) {
    print("dataCorrupted =>")
    print(context.codingPath)
    print(context.debugDescription)
}
catch {
    print("Error: \(error.localizedDescription)")
}
```

Un catch diferente para cada tipo de error.

Implementar Codable

- El compilador proporciona implementaciones por defecto para los tipos que implementan el protocolo Codable (Decodable y/o Encodable).
- También podemos proporcionar una implementación personalizada cuando queramos modificar algo respecto de la implementación generada por defecto.
  - Por ejemplo: Al decodificar queremos validar que ciertos valores cumplen una condición, o que se satisfacen algunas relaciones entre varios elementos de la app, ...

Ejemplo 4.1

```
let jsonData = "{ \"name\": \"Pepe\", \"age\": 133 }".data(using: .utf8)!
struct Person : Codable {
    let name: String
    let age: Int
    private enum CodingKeys : String, CodingKey {
        case name
        case age
    }
    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        name = try container.decode(String.self, forKey: .name)
        age = try container.decode(Int.self, forKey: .age)
        guard age < 100 else {
            throw DecodingError.dataCorruptedError(forKey: .age, in: container,
                                                   debugDescription: "La edad es muy grande")
        }
    }
}
do { let p = try JSONDecoder().decode(Person.self, from: jsonData)
}
catch DecodingError.dataCorrupted(let context) {
    print(context.codingPath)       // [???.Person.(CodingKeys in ???).age]
    print(context.debugDescription) // La edad es muy grande
}
catch { print("Error: \(error.localizedDescription)") }
```

Usa un Keyed Container

Creamos nuestra propia implementación de init para comprobar que la edad es menor de 100.

Containers

- Los contenedores son almacenes donde se guardan los valores codificados.
- Los contenedores los usan:
  - Los objetos codificadores, conformes con **Encoder**, para guardar los valores que codifican dentro del contenedor.
  - Los objetos decodificadores, conformes con **Decoder**, para reconstruir los datos extrayendo los valores que están guardados en un contenedor.
- Esta diseñado para que las aplicaciones no dependen de los detalles del contenedor (tipo de contenedor, claves y valores usados, ...) usado por un tipo Swift para archivar sus valores. Estos detalles son privados del tipo Swift.
- Existen varios tipos de contenedores:
  - **KeyedDecodingContainer** y **KeyedEncodingContainer**
    - Los valores guardados en el contenedor están asociados a una clave.
    - Estos son los más usados por temas de compatibilidad y mantenimiento entre versiones (ej: solo hay que añadir una nueva clave para que una nueva versión de la app siga funcionando).
  - **UnkeyedDecodingContainer** y **UnkeyedEncodingContainer**
    - El contenedor guarda una secuencia de valores en orden. Los valores no están asociados a una clave, sino que se guardan en un determinado orden.
  - **SingleValueDecodingContainer** y **SingleValueEncodingContainer**
    - El contenedor almacena un único valor (no asociado a una clave).

Ejemplo 4.2

Reimplementar Codable para usar un contenedor Unkeyed. El JSON es un array de números

```
let jsonData = "[2.5, 5.0]".data(using: .utf8)!

struct Point : Codable {
    var x: Double
    var y: Double
    init(from decoder: Decoder) throws {
        var container = try decoder.unkeyedContainer()
        x = try container.decode(Double.self)
        y = try container.decode(Double.self)
    }
    public func encode(to encoder: Encoder) throws {
        var container = encoder.unkeyedContainer()
        try container.encode(x)
        try container.encode(y)
    }
}
do {
    let decoder = JSONDecoder()
    let p = try decoder.decode(Point.self, from: jsonData)
    print(p)
    let encoder = JSONEncoder()
    let data = try encoder.encode(p)
    if let str = String(data: data, encoding: .utf8) { print(str) }
}
catch { print("Error: \(error)") }
```

Contenedores Anidados

- Los contenedores pueden anidarse.
  - Por ejemplo, un KeyedContainer puede tener como valor un UnkeyedContainer, u otro KeyedContainer, etc.

- Caso de Uso 1: Los contenedores anidados pueden usarse con clases para gestionar la herencia, es decir, para encapsular los datos de las clase padre en su propio contenedor, aislándolos de los datos de la subclase.
  - Los contenedores usados para codificar tienen un método, llamado **superEncoder**, que crea un contenedor anidado para guardar los valores de la clase padre. Este método devuelve el Encoder que hay que usar para codificar la clase padre en el contenedor creado.
  - Los contenedores usados para decodificar tienen un método, llamado **superDecoder**, que devuelve el Decoder que hay que usar para reconstruir los valores de la clase padre. Estos valores se sacan del contenedor anidado.
  - Consultar en la documentación los diferentes métodos superEncoder y superDecoder que existen.

- Caso de Uso 2: Los contenedores anidados pueden usarse para aplanar objetos.
  - Por ejemplo, la representación JSON de los datos presenta cierto nivel de anidamiento que no queremos tener en el tipo Swift.
    - Se definen otros enumerados XXXXCodingKeys para los datos anidados.
    - Los contenedores tienen un método, llamado **nestedContainer**, que devuelve el contenedor a usar para los datos anidados.

Ejemplo 4.3 (*para el caso de uso 1)*

```
class Circle : Codable {
    var radius: Double

    private enum CodingKeys : String, CodingKey {
        case radius
    }

    init(radius: Double) {
        self.radius = radius
    }

    required init(from decoder: Decoder) throws {
        let container = try decoder.singleValueContainer()
        radius = try container.decode(Double.self)
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        try container.encode(radius)
    }}
```

La clase Circle se ha configurado para archivar usando un SingleValue???Container

```
class BorderedCircle : Circle {
    var border: Double

    private enum CodingKeys : String, CodingKey {
        case border
    }

    init(radius: Double, border: Double) {
        self.border = border
        super.init(radius: radius)
    }

    required init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        border = try container.decode(Double.self, forKey: .border)

        let superDecoder = try container.superDecoder()
        try super.init(from: superDecoder)
    }

    override func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(border, forKey: .border)

        let superEncoder = container.superEncoder()
        try super.encode(to: superEncoder)
    }
}
```

BorderedCircle usa contenedores anidados

Contenedor para el padre (super).

Contenedor para el padre (super).

```swift
let c1 = BorderedCircle(radius: 2.5, border: 3.3)

print("C1: r=\(c1.radius) b=\(c1.border)") // C1: r=2.5 b=3.3

let encoder = JSONEncoder()
do {
    let data = try encoder.encode(c1)

    let decoder = JSONDecoder()
    let c2 = try decoder.decode(BorderedCircle.self, from: data)

    print("C2: r=\(c2.radius) b=\(c2.border)") // C2: r=2.5 b=3.3
}
catch {
    print("Error: \(error)")
}
```

Codificar y decodificar un objeto BorderedCircle

Ejemplo 4.4 (*para el caso de uso 2)*

```
// JSON anidado:
// { "type": "cuadrado",
//   "info": { "sides":4,
//             "color":"rojo"
//           }
// }

struct Figure : Codable {
    var type: String
    var sides: Int
    var color: String

    init(type: String, sides: Int, color: String) {
        self.type = type
        self.sides = sides
        self.color = color
    }
```

Tenemos un JSON con anidamiento y el tipo Swift es plano.

```
    private enum CodingKeys : String, CodingKey {
        case type
        case info
    }

    private enum InfoCodingKeys : String, CodingKey {
        case sides
        case color
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        type = try container.decode(String.self, forKey: .type)

        let infoContainer = try container.nestedContainer(
                                    keyedBy: InfoCodingKeys.self, forKey: .info)
        sides = try infoContainer.decode(Int.self, forKey: .sides)
        color = try infoContainer.decode(String.self, forKey: .color)
    }
    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(type, forKey: .type)

        var infoContainer = container.nestedContainer(
                                   keyedBy: InfoCodingKeys.self, forKey: .info)
        try infoContainer.encode(sides, forKey: .sides)
        try infoContainer.encode(color, forKey: .color)
    }

}
```

Campos exteriores

Campos anidados

```
let f1 = Figure(type: "cuadrado", sides: 4, color: "rojo")

print("F1: t=\(f1.type) s=\(f1.sides) c=\(f1.color)")
// F1: t=cuadrado s=4 c=rojo

let encoder = JSONEncoder()
do {
    let data = try encoder.encode(f1)

    if let str = String(data: data, encoding: .utf8) {
        print(str)
        // {"type":"cuadrado","info":{"sides":4,"color":"rojo"}}
    }

    let decoder = JSONDecoder()
    let f2 = try decoder.decode(Figure.self, from: data)

    print("F2: t=\(f2.type) s=\(f2.sides) c=\(f2.color)")
    // F2: t=cuadrado s=4 c=rojo
}
catch {
    print("Error: \(error)")
}
```

Formato de las Representaciones Externas

- Todo está diseñado para que el proceso de codificación y decodificación sea independiente del formato en el que se representan los datos externamente.
  - Los Encoders/Decoders y los contenedores se abstraen del formato externo (JSON, Plist, formato de las fechas, bytes en base64 o binario, ...).
  - Pueden usarse otros formatos externos, tener diferentes configuraciones, sin necesidad de cambiar los tipos existentes.
- Tenemos los codificadores para JSON y para Listas de Propiedades, con varias opciones de configuración.

Core Data

Core Data

- Aplicación para el diseño visual de modelos de datos.
- Los datos se almacenan por defecto usando una base de datos SQLite.
  - Alternativas: ficheros binarios, memoria.
- Manejamos ese almacén de datos usando un contexto.
  - No vemos como se almacena.

- Con el editor creamos entities
  - Son los tipos de datos que creamos.
- En la aplicación creamos "**managed objects**"
  - Son las instancias de las entities definidas.

  - Las entities definen propiedades
  - Los managed objects usan KVC
    - para acceder al valor de una propiedad se usa su nombre como clave.