

# Tutoriel 2 : Persistance des données

**Attention ces notions ne seront revues qu'au cours du prochain TP (aucun CM ne sera consacré à ces notions). Si vous avez des questions ou des difficultés au cours de ce tutoriel, notez les, vous pourrez interroger votre enseignant lors du prochain TP. Par ailleurs, un QCM sera organisé au début du prochain TP pour évaluer ce que vous avez retenu de ce tutoriel.**

Un aspect essentiel du développement d'iOS est la persistance des données. Pratiquement toutes les applications iOS stockent des données pour une utilisation ultérieure. Les données stockées par votre application peuvent être des préférences utilisateur, des caches temporaires ou même de grands ensembles de données relationnelles.

Avant de discuter des stratégies de persistance des données les plus couramment utilisées par les développeurs sur la plateforme iOS, la première partie de ce tutoriel sera consacrée à la présentation du système de fichiers et du concept de sandboxing de l'application. Puis, dans la seconde partie, trois options de persistance de données seront décrites plus en détails avant la présentation des protocoles **Encodable**, **Decodable** et **Codable** dans une troisième partie. Finalement, dans la quatrième partie, vous créerez une application vous permettant d'enregistrer une petite promotion d'étudiants.

## Partie I : Système de fichiers et Sandboxing

La sécurité sur la plate-forme iOS est l'une des principales priorités d'Apple depuis l'introduction de l'iPhone en 2007. Contrairement aux applications OS X, une application iOS est placée dans un sandbox (bac à sable) d'application. Le sandbox d'une application ne fait pas uniquement référence au répertoire sandbox d'une application dans le système de

fichiers. Il comprend également un accès contrôlé et limité aux données utilisateur stockées sur l'appareil, aux services système et au matériel.

## Sandboxing et Répertoires

### Répertoire personnel

Le système d'exploitation installe chaque application iOS dans un répertoire sandbox contenant le répertoire bundle de l'application (cf. prochain paragraphe) et trois autres répertoires : Documents, Library et tmp. Le répertoire sandbox de l'application, souvent appelé son répertoire personnel (home directory), est accessible en appelant une simple fonction du framework Foundation : `NSHomeDirectory()`.

```
print(NSHomeDirectory())
```

Créez un nouveau projet Xcode basé sur le modèle d'application Single View et nommez-le Data Persistence. Ouvrez `AppDelegate.swift` et ajoutez l'extrait de code ci-dessus à la fonction `application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?)`. Si vous exécutez l'application dans le simulateur, la sortie dans la console va ressembler à ceci:

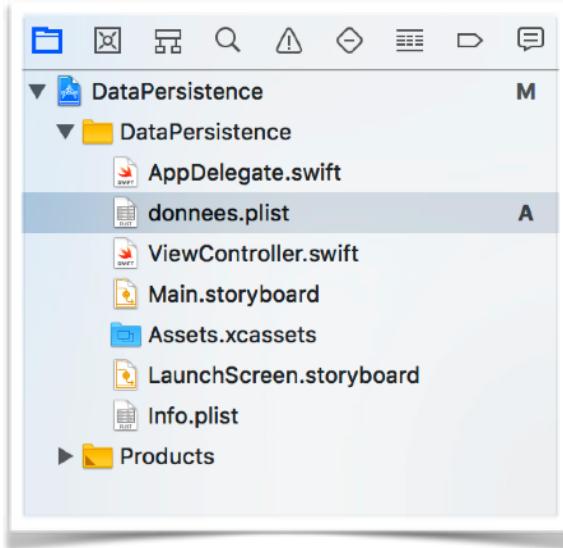
```
/Users/Bart/Library/Developer/CoreSimulator/Devices/14F00EFB-2EAB-438C-B401-7FEFDA1D94AB/data/Containers/Data/Application/81B23594-3BA2-4AF9-B91A-F74A53FD6945
```

Par contre, si vous exécutez l'application sur un périphérique physique, la sortie est un peu différente comme vous pouvez le voir ci-dessous. Le sandbox de l'application et les limitations imposées sont identiques, cependant.

```
/var/mobile/Containers/Data/Application/41E7939B-6A39-4005-9C28-372FD9C7AD99
```

## Répertoire Bundle

Le répertoire **Bundle** de l'application peut être vu comme le répertoire contenant l'exécutable de votre application. Il contient également tous les fichiers de votre projet (c'est à dire ceux que vous voyez apparaître dans le navigateur de votre projet Xcode).



Le chemin vers ce répertoire est accessible grâce aux deux premières lignes de codes suivantes :

```
let mainBundle = Bundle.main  
print(mainBundle.bundlePath)  
  
let path = Bundle.main.path(forResource: "donnees", ofType: "plist")  
print(path!)
```

Les deux dernières lignes de codes permettent de récupérer le chemin vers le fichier `donnees.plist` situé dans le répertoire **Bundle** de votre application.

Attention, ce répertoire est accessible uniquement en lecture. Les fichiers contenus dans ce répertoire peuvent donc être utilisés pour initialiser votre application mais ne pourront être modifiés. Si vous souhaitez enregistrer de nouvelles données, vous devez utiliser le répertoire **Documents**.

## Répertoire Documents

Afin de récupérer le chemin d'accès vers le répertoire **Documents** de l'application, vous devez utiliser l'extrait de code suivant :

```
let directories = NSSearchPathForDirectoriesInDomains(.documentDirectory, FileManager.SearchPathDomainMask.userDomainMask, true)
if let documents = directories.first {
    print(documents)
    let cheminFichier = documents.appending("/donnees.plist")
    print(cheminFichier)
}
```

Dans la fonction `NSSearchPathForDirectoriesInDomains()`, qui est définie dans le framework Foundation, le premier argument appelé `documentDirectory` (de type `NSSearchPathDirectory`) permet d'indiquer que nous sommes uniquement intéressés par le répertoire **Documents** de l'application. La fonction renvoie un tableau de type `[String]`, dont le premier élément correspond au chemin d'accès vers le répertoire **Documents** de l'application. L'avant dernière ligne `let cheminFichier = documents.appending("/donnees.plist")` permet de définir le chemin vers le fichier `donnees.plist` qui est à la fois accessible en lecture et en écriture.

## A quoi sert le Sandboxing ?

Quel est l'avantage du bac à sable ? La raison principale d'utiliser le principe du sandboxing est la sécurité. En confinant les applications à leur propre bac à sable, les applications compromises ne peuvent pas endommager le système d'exploitation ou d'autres applications.

Les applications compromises peuvent être des applications qui ont été piratées, des applications qui sont intentionnellement malveillantes, ainsi que des applications qui contiennent des bugs critiques qui pourraient causer des dommages par inadvertance.

Même si les applications utilisent le principe du sandboxing sur la plate-forme iOS, les applications peuvent demander l'accès à certains fichiers ou éléments situés en dehors de leur bac à sable via un certain nombre d'interfaces système, par exemple la bibliothèque musicale

iTunes stockée sur votre appareil. Sachez cependant que les frameworks système sont en charge de toutes les opérations liées à l'accès aux fichiers.

### **Que stocker où ?**

Même si vous pouvez faire à peu près tout ce que vous voulez dans le bac à sable de votre application, Apple a fourni quelques lignes directrices concernant ce qui devrait être stocké où. Il est important de connaître ces directives. En effet, lorsqu'un périphérique iOS est sauvegardé sur votre ordinateur ou sur iCloud, tous les fichiers du bac à sable ne sont pas inclus dans la sauvegarde.

Le répertoire **tmp**, par exemple, ne devrait être utilisé que pour stocker temporairement des fichiers. Le système d'exploitation est libre de vider ce répertoire à tout moment, par exemple lorsque l'espace disque est insuffisant. Le répertoire **tmp** n'est pas inclus dans les sauvegardes.

Le répertoire **Documents** est destiné aux données utilisateur, tandis que le répertoire **Library** est utilisé pour les données d'application qui ne sont pas strictement liées à l'utilisateur. Le répertoire **Caches** dans le répertoire **Library** est un autre répertoire qui n'est pas sauvegardé.

Gardez également à l'esprit que votre application n'est pas censée modifier le contenu du répertoire **Bundle** de votre application. Le répertoire **Bundle** est associé à une signature lors de l'installation de l'application. En modifiant le contenu du répertoire **Bundle** de quelque manière que ce soit, la signature susmentionnée est modifiée, ce qui signifie que le système d'exploitation ne permet pas à l'application de se lancer à nouveau. C'est une autre mesure de sécurité mise en place par Apple pour protéger ses clients.

## Partie II : Options de persistance des données

Il existe plusieurs stratégies pour stocker les données d'application sur le disque. Dans ce tutoriel, nous présenterons brièvement les trois premières approches très courantes sous iOS:

- ◆ defaults system
- ◆ property lists
- ◆ protocole Codable
- ◆ Core Data
- ◆ iCloud

Si vous souhaitez avoir plus de détails sur les deux dernières options, vous pouvez étudiez les deux articles suivants :

Core Data : <https://www.raywenderlich.com/173972/getting-started-with-core-data-tutorial-2>

iCloud : <https://www.raywenderlich.com/134694/cloudkit-tutorial-getting-started>

### User Defaults

Le **User Defaults** est un élément dont iOS a hérité de OS X. Bien qu'il ait été créé et conçu pour stocker les préférences utilisateur, il peut être utilisé pour stocker tout type de données tant qu'il s'agit d'un type stockable dans un fichier de type **.plist**<sup>1</sup> : **NSString**<sup>2</sup>, **NSDate** , **NSDictionary** et **NSData**, ou l'une de leurs variantes modifiables (**NSMutableDictionary** par exemple).

Qu'en est-il des types de données Swift ? Heureusement, Swift est assez intelligent. Il peut stocker des chaînes de caractères et des nombres en les convertissant en **NSString** et en **NSNumber**. La même chose s'applique aux tableaux et aux dictionnaires en langage Swift.

---

<sup>1</sup> Plus de détails sur les fichiers de propriétés (property list) un peu plus loin dans ce tutoriel.

<sup>2</sup> Les types de données de type **NS...** sont hérités de Objective-C et du système d'exploitation orienté objet NextStep proposés par Steve Jobs après sa démission forcée d'Apple en 1985.

Le système par défaut n'est rien de plus qu'un fichier de type **.plist** (pour *Property List* ou liste de propriétés en français) stocké dans le dossier **Preferences** du dossier **Library** de l'application.

Le code suivant permet de paramétriser de nouvelles valeurs dans le système par défaut (setting values) puis de récupérer ces valeurs (getting values) qui traduisent les préférences utilisateurs ou ses informations de profil.

```
let userDefaults = UserDefaults.standard  
|  
// Setting Values  
userDefaults.set(true, forKey: "musique")  
userDefaults.set(12, forKey: "taillePolice")  
userDefaults.set("Martin", forKey: "nom")  
userDefaults.set([175, 62], forKey: "taille-poids")  
|  
// Getting Values  
let musique = userDefaults.bool(forKey: "musique")  
let tailleP = userDefaults.integer(forKey: "taillePolice")  
let nom = userDefaults.object(forKey: "nom")  
let taille_poids = userDefaults.object(forKey: "taille-poids")  
|  
userDefaults.synchronize()  
print(NSHomeDirectory())
```

Dans la dernière ligne, la fonction **synchronize()** est appelée sur l'objet **UserDefaults** pour écrire les modifications sur le disque. Cependant il est rarement nécessaire d'appeler **synchronize()**, car le système par défaut enregistre les modifications si nécessaire.

Afin de visualiser les données enregistrer, collez l'extrait de code ci-dessus dans la méthode **application(\_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?)** et exécutez l'application dans le simulateur. Ouvrez le terminal et copier coller le chemin qui s'est affiché sur la console puis ajoutez l'instruction « **open .** », une fenêtre Finder va s'ouvrir (correspondant au bac à sable de votre application). Ouvrez le dossier **Preferences**, situé dans le dossier **Library**, et inspectez son contenu.



Vous devriez voir un fichier au format .plist (une liste de propriétés) avec un nom identique à celui de l'application. Cette liste de propriétés est stockée sur disque sous forme de fichier XML, que vous pouvez visualiser facilement avec Xcode.

Key	Type	Value
Root	Dictionary	(4 items)
taillePolice	Number	12
musique	Boolean	YES
nom	String	Martin
taille-poids	Array	(2 items)
Item 0	Number	175
Item 1	Number	62

Vous pouvez également visualiser cette liste de propriétés sous la forme d'un fichier XML.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>musique</key>
    <true/>
    <key>nom</key>
    <string>Martin</string>
    <key>taille-poids</key>
    <array>
        <integer>175</integer>
        <integer>62</integer>
    </array>
    <key>taillePolice</key>
    <integer>12</integer>
</dict>
</plist>

```

## Property Lists

Comme nous l'avons vu précédemment, les fichiers de propriétés (fichier `plist` pour *Property List*) permettent d'enregistrer au format XML certains objets standards d'iOS (`Date`, `String`, `Array`, etc.). La racine du fichier plist est généralement un tableau ou un dictionnaire, comme dans l'exemple ci dessous.

The screenshot shows the Xcode interface with two windows. The top window is a Property List editor titled 'donnees.plist'. It displays a hierarchical structure of data:

Key	Type	Value
Root	Array	(2 items)
Item 0	Dictionary	(3 items)
nom	String	Martin
age	Number	30
dateNaissance	Date	12 mars 1988 à 09:15:51
Item 1	Dictionary	(3 items)
nom	String	Durand
age	Number	28
dateNaissance	Date	28 févr. 1990 à 12:14:08

The bottom window shows the XML code generated by the plist file:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
3 <plist version="1.0">
4 <array>
5   <dict>
6     <key>nom</key>
7     <string>Martin</string>
8     <key>age</key>
9     <integer>30</integer>
10    <key>dateNaissance</key>
11    <date>1988-03-12T08:15:51Z</date>
12  </dict>
13  <dict>
14    <key>nom</key>
15    <string>Durand</string>
16    <key>age</key>
17    <integer>28</integer>
18    <key>dateNaissance</key>
19    <date>1990-02-28T11:14:08Z</date>
20  </dict>
21 </array>
22 </plist>
23

```

Comme je l'ai mentionné précédemment, il est important de garder à l'esprit qu'une liste de propriétés ne peut stocker que des données de liste de propriétés. Cela signifie-t-il qu'il n'est pas possible de stocker des objets de modèle personnalisés à l'aide de listes de propriétés ? C'est possible. Cependant, les objets de modèle personnalisés doivent être archivés avant de pouvoir être stockés dans une liste de propriétés. L'archivage d'un objet signifie simplement

que l'objet doit être converti en un type de données pouvant être stocké dans une liste de propriétés, telle qu'une instance de type **Data**.

## Enregistrer des données dans un fichier

L'extrait de code suivant vous présente comment écrire un tableau ou un dictionnaire sur le disque. En théorie, la structure de données contenue dans une liste de propriétés peut être aussi complexe ou aussi grande que vous le souhaitez. Cependant, gardez à l'esprit que les listes de propriétés ne sont pas destinées à stocker des dizaines ou des centaines de mégaoctets de données. Les utiliser dans ce cadre entraînera probablement des performances dégradées.

```
let directories = _____
    NSSearchPathForDirectoriesInDomains(.documentDirectory,
    FileManager.SearchPathDomainMask.userDomainMask, true)
|
if let documents = directories.first {
    let pathFruits = documents.appending("/fruits.plist")
    let pathDictionary = documents.appending("/dictionary.plist")
|
// Write Array to Disk
let fruits = ["Apple", "Mango", "Pineapple", "Plum", "Apricot"]
    as NSArray
let dictionary = ["anArray" : fruits, "aNumber" : 12345,
    "aBoolean" : true] as NSDictionary
|
    fruits.write(toFile: pathFruits, atomically: true)
    dictionary.write(toFile: pathDictionary, atomically: true)
}
```

Les premières lignes du code consistent à récupérer le chemin vers deux fichiers (**fruits.plist** et **dictionary.plist**) qui se trouvent dans le répertoire **Documents**. Deux structures de données sont ensuite créées : un tableau qui prend le type Objective-C **NSArray** et un dictionnaire (qui contient un tableau) qui prend le type Objective-C **NSDictionary**. L'écriture de ces deux structures de données dans les fichiers se fait en appelant la méthode **writeToFile (\_: atomically :)** sur les structures de données. Le deuxième paramètre de cette méthode (**atomically**) est un booléen qui, s'il prend la valeur vraie, permet d'effectuer l'écriture dans un fichier temporaire distinct, qui sera ensuite renommé, pour remplacer le fichier de destination, puis effacé. Cela fournit l'avantage qu'en cas de crash inopiné, le contenu du fichier de destination restera inchangé. Si le paramètre **atomically** prend la valeur **false**, l'écriture se fait directement dans le fichier de destination.

## Récupérer des données à partir d'un fichier

Dans l'exemple ci dessous, vous pouvez voir comment créer des tableaux ou des dictionnaires à partir d'un fichier .plist.

```
let directories = NSSearchPathForDirectoriesInDomains(.documentDirectory, FileManager.SearchPathDomainMask.userDomainMask, true)

if let documents = directories.first {
    let pathFruits = documents.appending("/fruits.plist")
    let pathDictionary = documents.appending("/dictionary.plist")

    // Load from Disk
    let loadedFruits = NSArray(contentsOfFile:pathFruits)
    if let fruits = loadedFruits {
        print(fruits)
        fruits.add("Strawberry")
        print(fruits)
    }

    let loadedDictionary = NSDictionary(contentsOfFile:pathDictionary)
    if let dictionary = loadedDictionary {
        print(dictionary)
    }
}
```

Comme précédemment, les premières lignes du code consistent à récupérer les chemins vers les fichiers `fruits.plist` et `dictionary.plist` qui se trouvent dans le répertoire `Documents`. La méthode `NSArray(contentsOfFile:)` (resp. `NSDictionary(contentsOfFile:)`) permettent ensuite de créer un tableau de type `NSArray` (resp. un dictionnaire de type `NSDictionary`) contenant le contenu du fichier `fruits.plist` (resp. `dictionary.plist`).

Ces deux types (`NSArray` et `NSDictionary`) sont cependant des types non modifiables, c'est-à-dire que l'on ne peut ni ajouter ni modifier les éléments du tableau `fruits` ou du dictionnaire `dictionary`. Si vous testez cette partie de code, vous verrez que les deux `print(fruits)` affichent la même chose dans la console alors que nous avons ajouté un élément au tableau `fruits`<sup>3</sup>. Pour pallier à cette difficulté, il faut modifier le type des structures de données lors de leur création en appelant les méthodes `NSMutableArray(contentsOfFile:)` et `NSMutableDictionary(contentsOfFile:)` qui créent des types modifiables comme le montre l'extrait de code ci dessous.

---

<sup>3</sup> En réalité, vous aurez même une erreur de compilation précisant que la méthode `add` n'existe pas pour les objets de type `NSArray`.

```

let directories = NSSearchPathForDirectoriesInDomains(.documentDirectory,
    FileManager.SearchPathDomainMask.userDomainMask, true)

if let documents = directories.first {
    let pathFruits = documents.appending("/fruits.plist")
    let pathDictionary = documents.appending("/dictionary.plist")

    // Load from Disk
    let loadedFruits = NSMutableArray(contentsOfFile:pathFruits)
    if let fruits = loadedFruits {
        print(fruits)
        fruits.add("Strawberry")
        print(fruits)
    }

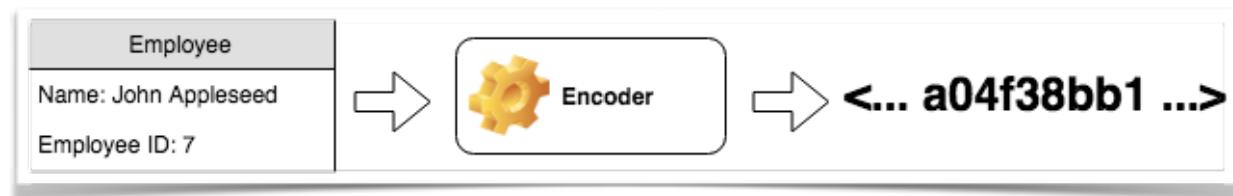
    let loadedDictionary =
        NSMutableDictionary(contentsOfFile:pathDictionary)
    if let dictionary = loadedDictionary {
        print(dictionary)
    }
}

```

## Protocole Codable<sup>4</sup>

La dernière option de persistance des données présentée dans ce tutoriel est le protocole **Codable** qui permet de convertir vos instance d'objets en flux d'octets<sup>5</sup>. En effet, nous avons vu que les fichiers au format .plist ne pouvait stocker que des types issus d'Objective C, vous ne pouvez donc pas enregistrer l'instance d'une classe que vous avez vous même définie.

Lorsque vous voulez écrire une instance dans un fichier dans le cadre du protocole **Codable**, l'instance elle-même ne peut pas être écrite telle quelle dans le fichier, vous devez donc l'encoder dans une autre représentation, comme un flux d'octets:



<sup>4</sup> Le protocole Codable remplace le protocole NSCoding des précédentes versions de Swift.

<sup>5</sup> Ce processus est connu sous le nom d'encodage ou sérialisation. Le processus inverse de transformation des données en instance est appelé décodage ou désérialisation.

Une fois que les données sont codées et enregistrées dans un fichier, vous pouvez les réactiver lorsque vous le souhaitez en utilisant un décodeur:



### Les protocoles Encodable et Decodable

Le protocole **Encodable** est utilisé par les types qui peuvent être codés dans une autre représentation. Ce protocole est composé d'une seule méthode `func encode (to: Encoder) throws` que le compilateur va générer pour vous si tous les attributs de la classe qui se conforme au protocole sont également conformes au protocole **Encodable** (plus de détails un peu plus loin dans le tutoriel).

Le protocole **Decodable** est utilisé par les types qui peuvent être décodés. Ce protocole déclare un seul constructeur: `init (from decoder: Decoder) throws`. Vous saurez quand et comment implémenter ces méthodes à la fin de ce tutoriel.

### Qu'est-ce que le protocole Codable ?

**Codable** est un protocole auquel un type peut se conformer, pour déclarer qu'il peut être encodé et décodé. C'est fondamentalement un alias pour les protocoles **Encodable** et **Decodable**.

`typealias Codable = Encodable & decodable`

Il existe plusieurs types de base du langage Swift qui sont « codables » : **Int**, **String**, **Date** et beaucoup d'autres types de la bibliothèque standard et du framework Foundation. Si vous voulez que votre classe soit « codable », la manière la plus simple de le faire est de faire en sorte qu'elle est conforme au protocole **Codable** et de s'assurer que tous ses attributs le soient également.

Par exemple, supposons que définissiez une classe **Personne** comme ceci:

```
class Personne {
    var nom : String
    var prenom : String
    var age : Int
}
```

Tout ce que vous devez faire pour pouvoir encoder et décoder ce nouveau type de données est de le faire se conformer au protocole **Codable** :

```
class Personne : Codable {  
    var nom : String  
    var prenom : String  
    var age : Int  
}
```

Ceci est possible car les types **String** et **Int** sont eux même « codables ». Mais que se passe-t-il si votre classe inclut d'autres types personnalisés en tant qu'attribut. Par exemple, si votre classe **Personne** possède un attribut adresse de type **Adresse** :

```
class Personne : Codable {  
    var nom : String  
    var prenom : String  
    var age : Int  
    var adresse : Adresse  
}  
  
class Adresse : Codable {  
    var rue : String  
    var numero : Int  
    var ville : String  
    var codePostal : Int  
}
```

En vous assurant que votre classe **Adresse** est également conforme au protocole **Codable**, vous maintenez la conformité générale de votre classe **Personne** au protocole **Codable**. Toutes les structures de données, tels que les tableaux et les dictionnaires, sont également « codables » s'ils contiennent des types « codables ».

### Encodage et décodage des types personnalisés

Vous pouvez utiliser plusieurs représentations pour encoder ou décoder vos classes, comme des fichiers au format XML ou au format .plist. Dans cette section, vous apprendrez à encoder et à décoder à partir de JSON, en utilisant les classes **JSONEncoder** et **JSONDecoder** de Swift.

JSON est l'abréviation de JavaScript Object Notation et est l'un des moyens les plus populaires pour sérialiser les données. Il est facilement lisible par les humains et facile à analyser et à générer par les ordinateurs.

## JSONEncoder et JSONDecoder

Une fois que vous avez un type codable, vous pouvez utiliser la méthode `JSONEncoder` pour convertir votre type en données qui peuvent être écrites dans un fichier. Supposons que vous déclariez cette instance de `Personne` dans votre fichier `ViewController.swift` :

```
let a = Adresse(rue: "rue du paradis", numero: 2, ville: "Orsay",
                codePostal: 91400)
let p = Personne(nom: "Martin", prenom: "Jean", age: 25, adresse: a)
```

Afin de pouvoir encoder la personne `p`, vous devez utiliser la méthode `encode` de la classe `JSONEncoder` comme ceci:

```
let jsonEncoder = JSONEncoder()
do {
    let jsonData = try jsonEncoder.encode(p)
    print(jsonData)
}
catch {
    print("error")
}
```

Dans le code précédent, vous pouvez remarquer que vous devez utiliser `try` car la méthode `encode(_ :)` peut échouer et générer une erreur. Par ailleurs, si vous essayez d'imprimer `jsonData` directement, vous verrez que Xcode omet les données et ne fournit que le nombre d'octets de `jsonData` car `jsonData` contient une représentation illisible de la personne `p`.

Si vous souhaitez créer une version lisible de cet objet JSON sous forme de chaîne de caractères, vous pouvez utiliser le constructeur de la classe `String` suivant :

```
let jsonString = String(data: jsonData, encoding: .utf8)
print(jsonString!)
// {"prenom":"Jean","age":25,"adresse":{"numero":2,"rue":"rue du
paradis","ville":"Orsay","codePostal":91400},"nom":"Martin"}
```

Vous pouvez maintenant enregistrer vos données JSON dans un fichier ou les envoyer sur le réseau.

```

let a = Adresse(rue: "rue du paradis", numero: 2, ville: "Orsay",
codePostal: 91400)
let p = Personne(nom: "Martin", prenom: "Jean", age: 25, adresse: a)

let directories =
    NSSearchPathForDirectoriesInDomains(.documentDirectory,
    FileManager.SearchPathDomainMask.userDomainMask, true)

if let documents = directories.first {
    let fileName = documents.appending("/personne.json")
    let url = URL(fileURLWithPath: fileName)
    let jsonEncoder = JSONEncoder()
    do {
        let jsonData = try jsonEncoder.encode(p)
        try jsonData.write(to: url)
    }
    catch {
        print("error")
    }
}

```

Dans l'exemple précédent, la personne `p` est convertie en donnée JSON appelée `jsonData` puis la méthode `write(to:)` est appelée afin d'enregistrer cette donnée dans le fichier `personne.json` situé dans le répertoire `Documents`.

Finalement, vous pouvez également extraire des données JSON depuis un fichier et décoder ces données dans une instance grâce à la classe `JSONDecoder` :

```

let directories =
    NSSearchPathForDirectoriesInDomains(.documentDirectory,
    FileManager.SearchPathDomainMask.userDomainMask, true)

if let documents = directories.first {
    let fileName = documents.appending("/personne.json")
    let url = URL(fileURLWithPath: fileName)
    let jsonDecoder = JSONDecoder()
    do {
        let jsonDataNew = try Data(contentsOf: url)
        let p2 = try jsonDecoder.decode(Personne.self, from:
jsonDataNew)
        print(p2.nom)
    }
    catch {
        print("error")
    }
}

```

Dans cet exemple, les données JSON sont récupérées depuis le fichier `personne.json` situé dans le répertoire `Documents` puis la méthode `decode` de la classe `JSONDecoder` est appelée. Notez que vous devez indiquer au décodeur le type de décodage du JSON (c'est-à-dire votre classe `Personne`), car le compilateur ne peut pas le comprendre seul.

## Partie III : Enregistrement d'une promotion d'étudiants

Dans cette dernière partie du tutoriel, vous allez manipuler le protocole `Codable` afin d'enregistrer dans un fichier `promotion.json` une promotion d'étudiants.

Pour cela vous allez créer un nouveau projet `Promotion` dans lequel, vous allez :

1. créer deux nouvelles classes : `Promotion` et `Etudiant` (une promotion est représentée par un nom et possède un tableau d'étudiant et un étudiant a un nom, un age et un numéro d'étudiant),
2. créer plusieurs instances de la classe `Etudiant` afin de créer une nouvelle promotion,
3. transformer cette promotion au format `json`,
4. enregistrer votre donnée `json` dans un fichier `promotion.json`,
5. récupérer les données stockées dans votre fichier,
6. recréer un nouvel objet promotion et l'afficher dans la console.

Déposez votre application sur Moodle.

### Sources :

- <https://code.tutsplus.com/tutorials/ios-from-scratch-with-swift-data-persistence-and-sandboxing-on-ios--cms-25505>
- <https://www.raywenderlich.com/172145/encoding-decoding-and-serialization-in-swift-4>