DÉVELOPPEZ VOS APPLICATIONS POUR iPHONE ET iPAD

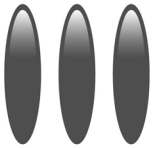
LE GUIDE COMPLET

Jean-Pierre IMBERT

Micro Application

Développez vos applications pour iPhone, iPod Touch, iPad

LE GUIDE COMPLET



1ère Édition - Mai 2010

Auteur

Jean-Pierre IMBERT

ISBN : 978-2-300-028021

ISSN : 1950-0289

MICRO APPLICATION
20-22, rue des Petits-Hôtels
75010 PARIS
Tél. : 01 53 34 20 20
Fax : 01 53 34 20 00
http://www.microapp.com

Support technique :
Également disponible sur
www.microapp.com

Retrouvez des informations sur cet ouvrage !

Rendez-vous sur le site Internet de Micro Application **www.microapp.com**. Dans le module de recherche, sur la page d'accueil du site, entrez la référence à 4 chiffres indiquée sur le présent livre.
Vous accédez directement à sa fiche produit.

Avant-propos

Destinée aussi bien aux débutants qu'aux utilisateurs initiés, la collection *Guide Complet* repose sur une méthode essentiellement pratique. Les explications, données dans un langage clair et précis, s'appuient sur de courts exemples. En fin de chaque chapitre, découvrez, en fonction du sujet, des exercices, une check-list ou une série de FAQ pour répondre à vos questions.

Vous trouverez dans cette collection les principaux thèmes de l'univers informatique : matériel, bureautique, programmation, nouvelles technologies...

Conventions typographiques

Afin de faciliter la compréhension des techniques décrites, nous avons adopté les conventions typographiques suivantes :

- **gras** : menu, commande, boîte de dialogue, bouton, onglet.
- *italique* : zone de texte, liste déroulante, case à cocher, bouton radio.
- `Police bâton` : Instruction, listing, adresse internet, texte à saisir.
- ✂ : indique un retour à la ligne volontaire dû aux contraintes de la mise en page.

Il s'agit d'informations supplémentaires relatives au sujet traité.

Met l'accent sur un point important, souvent d'ordre technique qu'il ne faut négliger à aucun prix.

Propose conseils et trucs pratiques.

Donne en quelques lignes la définition d'un terme technique ou d'une abréviation.

INTRODUCTION

Pour qui est ce livre ?

Ce livre est destiné à tous ceux qui souhaitent développer leur propre application pour iPhone, iPod Touch ou pour iPad. Il vous accompagnera dans l'étude de la programmation pour ces équipements, dans la découverte des outils de développement d'Apple et du langage Objective-C ; et jusqu'à la distribution de votre application sur l'AppStore.

iPhone, iPod Touch, iPad

Nous employons le terme iPhone pour évoquer indistinctement l'iPhone l'iPod Touch ou l'iPad. Lorsqu'une caractéristique est disponible uniquement sur l'un ou l'autre de ces appareils, nous le précisons en indiquant par exemple : "*cette caractéristique n'est pas disponible sur iPod Touch*".

Vous explorerez les techniques permettant d'utiliser les caractéristiques les plus innovantes de l'iPhone (accélérateur, géo-localisation, capacités graphiques, gestes, etc.) et serez certainement séduit par la

désarmante facilité avec laquelle vous mettrez en œuvre ces techniques en utilisant les frameworks et le SDK d'Apple.

Cet ouvrage est destiné à ceux qui ont déjà une connaissance de la programmation d'applications logicielles. Si ce n'est pas le cas, nous vous recommandons la lecture de *Débutez en Programmation* (éditions Micro Application).

Développer pour iPhone ou pour iPad ?

Les applications développées pour iPhone et iPod Touch peuvent être exécutées sur iPad. L'utilisateur a alors la possibilité de visualiser l'interface dans sa taille originale ou dans une taille double. À l'inverse, une application développée pour iPad ne peut généralement pas être exécutée sur iPhone ou iPod Touch, ne serait ce que pour la taille de l'écran. Pour ces trois appareils, il faut utiliser le SDK *iPhone OS*.

Il est donc a priori plus intéressant de développer pour iPhone plutôt que seulement pour iPad. La plupart des chapitres de cet ouvrage traitent donc de l'iPhone et de l'iPod Touch. Le dernier chapitre traite des spécificités de l'iPad et de la réalisation d'applications qui s'adaptent à l'appareil sur lequel elles s'exécutent.

De quoi avez-vous besoin ? Comment l'obtenir ?

Le matériel

SDK Apple
Le SDK Apple ne fonctionne pas sur un PC sous Windows ou Linux.

L'environnement de développement utilisé dans cet ouvrage est le SDK 3.2 qui permet de développer des applications pour iPhone, iPod Touch et iPad. Il ne peut s'exécuter que sur un Macintosh à processeur Intel doté du système d'exploitation Snow Leopard (Mac OS X 10.6) ou ultérieur.

Intel ou PowerPC

Durant de longues années, une particularité des Macintosh d'Apple était de fonctionner sur un processeur de la famille PowerPC, développé en collaboration avec IBM et Motorola, et pas sur un processeur de la famille Pentium ou équivalent comme les PC. À partir du début des années 2000, l'écart de performances entre ces deux types de processeur s'est progressivement accru en faveur d'Intel, si bien que Steve Jobs a annoncé en juin 2005 le changement de processeur. Tous les Macintosh à partir de 2006 sont à processeur Intel et conviennent pour le développement sur iPhone.

Si ce n'est déjà fait, il vous faudra donc impérativement vous procurer un Mac si vous voulez développer des applications pour iPhone.

Le Refurb Store

Apple commercialise des produits reconditionnés sur son site marchand http://store.apple.com/fr. Il est possible d'y faire de bonnes affaires. Si vous êtes étudiant, pensez aussi aux offres spéciales "Éducation" d'Apple.

Figure 1 : Les produits reconditionnés sur l'Apple Store

Plusieurs revendeurs commercialisent également des machines d'occasion, soyez sûr de choisir un Mac à processeur Intel.

Le logiciel

L'environnement de développement est disponible gratuitement sur le site des développeurs d'Apple http://developer.apple.com. Dans la suite

de cet ouvrage, nous emploierons le terme de SDK (*Software Development Kit*) pour désigner l'environnement de développement.

Inscription obligatoire

L'inscription sur le site des développeurs d'Apple est obligatoire pour télécharger le SDK. Cette inscription est gratuite, elle vous permettra également d'accéder aux ressources techniques du site des développeurs (vidéos d'apprentissage, documentation technique, exemples de code source, forum des développeurs).

À l'heure où nous rédigeons ces lignes, le SDK est disponible en version 3.1.2 qui permet de développer des applications pour les versions 2 et 3 d'iPhone OS. Il est fourni au format *.dmg* (format d'image disque standard sur Mac OS X) et sa taille est environ de 2,7 Go ; il faut généralement plusieurs heures pour le télécharger. Vous obtiendrez un SDK complet et d'excellente qualité :

- outils de développement ;
- frameworks iPhone/iPod Touch/iPad et Mac OS X ;
- simulateur d'iPhone et d'iPad pour tester vos applications sur votre Macintosh ;
- outils et instruments divers (mesure de performance, recherche de bogues, ateliers de composition graphique, etc.).

1. Double-cliquez sur le fichier que vous venez de télécharger ; une fenêtre du Finder s'ouvre qui vous permet de visualiser le contenu de l'image disque.
2. Double-cliquez sur le fichier *iPhone SDK* ; le programme d'installation s'exécute et suit le processus standard sur Mac OS X : approbation de la licence d'utilisation des outils de développement puis du kit iPhone, personnalisation de l'installation (laisser les paramètres par défaut) et installation.

Installation et droits d'administration

L'installation du SDK nécessite les droits d'administration de l'ordinateur.

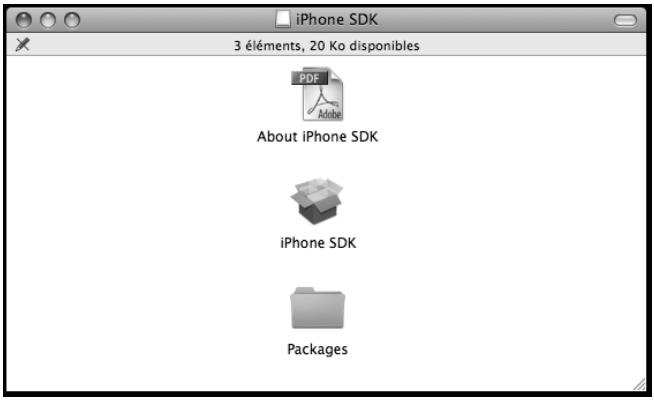


Figure 2 : Contenu de l'image disque du SDK iPhone

Un appareil de test

Vous pourrez tester plusieurs applications de ce livre en utilisant le simulateur fourni avec le SDK mais votre satisfaction sera décuplée lorsque vous exécuterez votre application sur votre appareil.

Il n'est généralement pas nécessaire de disposer d'un appareil spécifique pour effectuer vos tests ; l'iPhone, l'iPod Touch ou l'iPad que vous utilisez quotidiennement fera l'affaire.

Si vous souhaitez améliorer les tests avant de diffuser votre application, vous devrez disposer de plusieurs appareils, par exemple un iPhone 3G, un iPhone 3GS, un iPod Touch et un iPad. Certaines personnes de votre entourage disposent sûrement de ces appareils et seront sans doute fiers de vous aider à tester vos applications.

Afin de viser une plus large diffusion, il faudra tester vos applications non seulement sur plusieurs appareils mais aussi sous plusieurs versions d'OS. Il vaudra mieux à ce moment-là que vous disposiez d'appareils dont l'usage sera réservé aux tests ; vous risquez d'avoir moins d'amis si les changements d'OS ont provoqué des pertes de données sur les appareils qu'ils vous ont confiés.

L'inscription au programme des développeurs iPhone

Cette inscription payante, moins de 100 € par an, est indispensable pour tester son application sur un appareil réel. Elle vous permettra de :

- tester ou diffuser en mode Privé vos applications sur des appareils réels (jusqu'à 100) ;

- diffuser vos applications sur l'AppStore (et gagner de l'argent) ;
- disposer des versions Bêta de l'iPhone OS ;
- accéder à certaines informations spécifiques.

L'inscription se fait sur le site web des développeurs (**http://developer.apple.com**). Le processus d'inscription et de préparation aux tests réels est détaillé en annexe.

Une petite présentation

Les éléments de base

Pour développer une application logicielle, nous avons besoin des éléments suivants :

- Un langage de programmation ; pour développer sur l'iPhone OS, ce sera *Objective-C*.
- Des outils de développements (éditeur, compilateur, composition de l'interface graphique, débogueur, etc.) : nous utiliserons principalement *XCode* et *Interface Builder*.
- Une bibliothèque d'API (*Application Programming Interface*) permettant au code écrit dans le langage de programmation d'accéder aux fonctions fournies par le système d'exploitation ; sur les systèmes Mac OS X et iPhone OS, ces bibliothèques se nomment des *frameworks*. Si vous êtes familier de la terminologie PC/Windows, les DLL sont l'équivalent des frameworks.

Nous commencerons notre étude par la découverte des outils XCode et Interface Builder et du langage Objective-C puis nous aborderons progressivement les différents frameworks d'iPhoneOS.

Vous avez vraisemblablement déjà entendu parler de Cocoa Touch. Il s'agit de la partie de l'iPhone OS que nous utiliserons le plus souvent tout au long de notre parcours. Cocoa Touch est composé de 2 frameworks :

- *UIKit* prend en charge la gestion de l'interface utilisateur (`UI` pour User Interface) :
  - différents éléments de l'interface utilisateur (boutons, champs de texte, etc.) ;
  - gestion des événements (tapes et gestes) ;
  - fonctionnement général des applications.

- *Foundation* contient des classes utilitaires et des interfaces de haut niveau vers les fonctions du système :
  - classes de collection (tableaux, ensembles et dictionnaires) ;
  - classes utilitaires (dates, chaînes de caractères, etc.) ;
  - accès vers le gestionnaire de fichiers, les fonctions graphiques, l'accès au réseau, etc.

Spécificités du développement sur iPhone/ iPod touch/iPad

Le développement d'application pour iPhone OS présente peu de difficultés ; il existe de nombreux points communs avec le développement pour un ordinateur. Il faudra cependant tenir compte des limitations technologiques de ces objets épatants : écran de petite taille, nécessité d'économiser l'énergie, limitation de la mémoire. Quelques conséquences de ces limitations sont détaillées ci-après :

- Une seule application s'exécute à un instant donné (si l'on excepte le système d'exploitation) ; la première conséquence est qu'il est impossible de créer une application qui tourne en tâche de fond.
- Une seule fenêtre est affichée à l'écran.
- Chaque application possède un *bac à sable* (*sandbox*), un système de fichiers privé qui contiendra toutes ses données (fichiers de paramètres et fichiers de données) ; il est impossible à deux applications d'accéder au même fichier et donc d'échanger leur virus.
- La taille d'écran est limitée à 480 x 320 pixels sur iPhone et iPod Touch, et à 1024 x 768 pixels sur iPad.
- La taille de RAM d'un iPhone est de 128 Mo, approximativement la moitié de cette mémoire est utilisée par l'OS ; l'application en cours d'exécution doit se contenter d'environ 64 Mo, il faudra économiser la mémoire.

Langage Objective C

Objective-C est le langage de programmation "naturel" sur iPhone OS et aussi sur Mac OS X. Il est vrai que ce langage est rarement utilisé sur d'autres plateformes mais présente de nombreuses simi-

larités avec le langage Java plus largement employé. Objective-C est une extension "objet" du langage C, au même titre que C++, mais beaucoup plus simple que celui-ci et plus facile à apprendre.

Pour aller plus loin

Ce livre vous expliquera toutes les techniques fondamentales mises en œuvre dans les frameworks de l'iPhone OS. Il vous donnera les clés qui vous permettront de continuer votre exploration et de développer des applications dont la seule limite sera votre imagination.

Do you speak English ?

Ou plutôt, *Do you read english* ?

Pour aller plus loin, il vous faudra exploiter la riche documentation d'Apple celle intégrée à l'environnement de développement et celle disponible sur le site des développeurs. Cette documentation est en anglais. Ce sera donc un atout si vous êtes à l'aise avec la langue de Shakespeare.

Tout n'est pas perdu

Si vous êtes allergique à l'anglais, tout n'est pas perdu. Pour aller plus loin, il sera alors indispensable de vous inscrire sur un forum de développeurs français où vous trouverez toujours une bonne âme pour vous aider et vous transmettre son savoir, par exemple **http://forum.macbidouille.com** ou **www.pommedev.com**.

Les applications que nous détaillerons

Au fur et à mesure de notre parcours dans les frameworks de l'iPhone, nous développerons quelques applications. Vous pourrez les utiliser telles quelles sur votre iPhone et même les améliorer. En voici la liste :

- *HelloWorld* graphique ;
- Convertisseur de monnaie ;
- Prêts aux amis ;
- Détecteur de verticale.

Bonne exploration !

PREMIERS PAS

Dans ce chapitre, nous créerons notre première application. Comme le veut la tradition, il s'agira d'un "Hello World". Rien de très spectaculaire donc mais ce sera l'occasion de prendre en main les trois outils fondamentaux du SDK : **XCode**, **Interface Builder** et **iPhone Simulator**, le simulateur d'iPhone et d'iPad.

1.1. Créer un projet avec XCode

Lancer XCode

XCode est l'application qui va nous permettre de :

- gérer nos projets ;
- gérer et éditer les fichiers de code source ;
- construire et tester nos applications.

Un **projet** est l'ensemble des données nécessaires pour construire une application :

- code source ;
- ressources (images, sons, etc.) ;
- liste des frameworks utilisés ;
- informations complémentaires (fichier d'information, paramétrage de la construction, etc.).

Pour créer un projet, il faut d'abord lancer l'application **XCode**. L'installation standard du SDK place cette application dans le dossier */Developer/Applications*. Double-cliquez sur l'icône **XCode**.

Figure 1.1 : Localisation de l'application XCode

Si c'est la première fois que vous lancez **XCode**, la fenêtre d'accueil apparaît à l'écran. Vous pouvez fermer cette fenêtre.

Figure 1.2 : Fenêtre d'accueil de XCode

Accéder plus facilement à XCode

Vous aurez à vous servir intensément de XCode. Il sera plus pratique d'avoir l'application à disposition rapidement en la gardant dans le Dock.

Figure 1.3 : Garder XCode dans le Dock

Créer un projet

Procédez ainsi :

1 Sous XCode, activez la commande **New Project...** du menu **File**. Vous pouvez également cliquer sur **Create a new Xcode project** dans le panneau d'accueil.

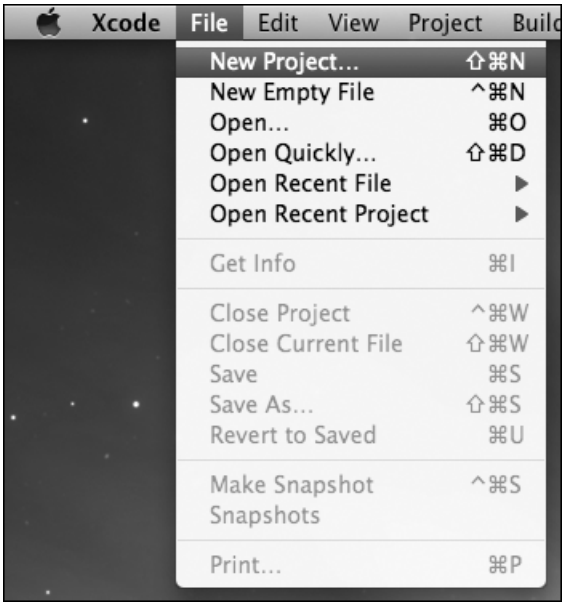


Figure 1.4 : Création d'un projet

L'Assistant New Project s'affiche, vous permettant de choisir le type de projet que vous souhaitez créer.

Figure 1.5 : Assistant Nouveau Projet

XCode prend en charge plusieurs types de projet. Lorsqu'on crée un projet, il faut choisir le bon modèle. Afin de vous faciliter le choix, la partie gauche de l'Assistant présente les groupes de modèles. Dans la suite de l'ouvrage, nous choisirons toujours le groupe *Application* pour **iPhone OS**.

2 Sélectionnez *View-based Application*, vérifiez que le menu déroulant *Product* est bien sélectionné sur **iPhone** et cliquez sur le bouton **Choose...**.

Les autres modèles d'application

Nous verrons les autres modèles d'applications dans la suite de cet ouvrage.

3 Un panneau s'affiche qui vous permet de nommer le projet en cours de création. Saisissez `HelloWorld` dans la zone de texte *Save As* puis cliquez sur le bouton *Save*.

Figure 1.6 : Panneau de sauvegarde

Le dossier de projet

XCode crée un dossier, du même nom que le projet, et y insère les fichiers composant le projet. Ce dossier recevra vos propres fichiers (images, icones, …). Il est créé par défaut dans votre dossier *Documents*. Vous pouvez choisir un autre emplacement avant de cliquer sur le bouton *Save*.

À ce stade, nous avons créé le dossier de projet et la fenêtre de projet de XCode s'affiche.

Gérer le projet

La fenêtre de projet permet de gérer tous les éléments d'un projet.

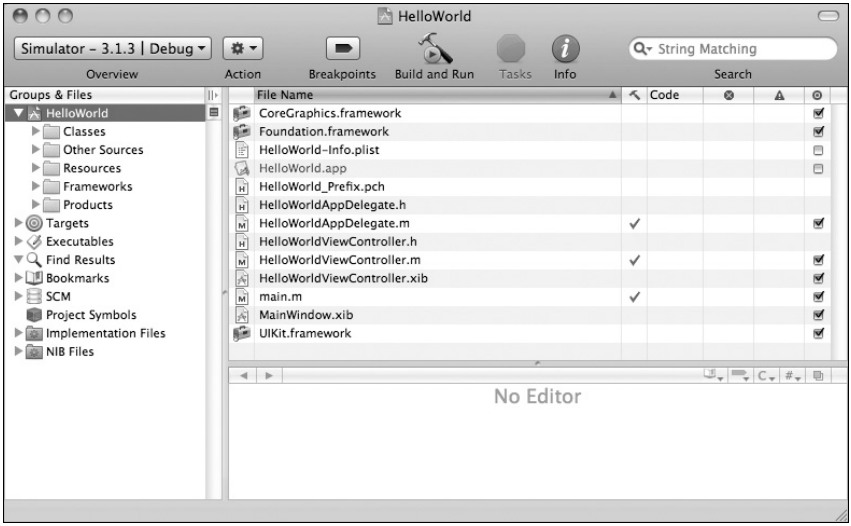


Figure 1.7 : La fenêtre de projet

Cette fenêtre est composée des éléments suivants :

- une barre d'outils en haut ;
- la zone des **Groupes et Fichiers** (Groups and Files) sur la partie gauche de la fenêtre, qui structure les différentes informations et les fichiers composant le projet ;
- la **Vue Détaillée** au centre de la partie droite de la fenêtre, dans laquelle on peut visualiser les attributs principaux de chacun des composants du projet ;

- une **Zone d'Édition** en bas de la partie droite de la fenêtre qui nous permettra d'éditer les fichiers du projet (c'est dans cette zone que nous saisirons le code source de l'application) ;
- une **Barre d'État** en bas de la fenêtre ; nous y lirons les messages émis par XCode lors de la construction de l'application.

Structure mais pas sous-dossiers

Les fichiers sont répartis sous XCode en *Classes*, *Autres Sources*, *Ressources*, etc. Cette structuration est indépendante de la façon dont les fichiers sont structurés en sous-dossiers dans le dossier du projet ; par défaut, seul un sous-dossier *Classes* est créé ; tous les autres fichiers sont "à plat" dans le dossier du projet.

Nous verrons l'utilisation de ces différentes parties dans la suite du livre. Pour l'heure, et si ce n'est pas déjà le cas, sélectionnez *HelloWorld* dans la zone des **Groupes et Fichiers** pour afficher la liste de tous les fichiers du projet dans la **Zone Détaillée**.

XCode a créé pour nous les fichiers suivants :

- Les fichiers de type *.framework* sont des liens vers les **frameworks** d'iPhone OS nécessaires pour notre application HelloWorld :
  - *CoreGraphics.framework* pour les fonctions de base de l'affichage graphique ;
  - *Foundation.framework* pour les accès aux fonctions de base du système (fichiers, réseau, etc.) ;
  - *UIKit.framework* qui contient toutes les classes d'objet de base pour créer une application sous **Cocoa Touch** (boutons, champs de texte, etc.).
- *HelloWorld-Info.plist* est le fichier des propriétés de l'application ; nous utiliserons bientôt ce fichier.
- *HelloWorld.app* est notre application. Son nom apparaît en rouge pour signaler que l'application n'est pas encore créée.
- Les fichiers de type *.pch* sont des fichiers intermédiaires dans le processus de construction de l'application ; nous ne nous en occuperons pas.
- Les fichiers *.m* (fichier des **définitions**) et *.h* (fichiers des **déclarations**) contiennent les codes source Objective-C de l'application :

— *main.m* à l'instar de *main.c* en langage C contient le code de la fonction *main* exécutée au lancement de l'application ; nous n'aurons généralement pas à modifier ce fichier.

— *HelloWorldAppDelegate.h* et *HelloWorldAppDelegate.m* contiennent le code source du **Délégué** de l'application.

— *HelloWorldViewController.h* et *HelloWorldViewController.m* contiennent le code source du **Contrôleur** de la Vue principale de l'application. Ces fichiers sont créés par XCode car nous avons utilisé le modèle *View-based Application* à la création du projet.

- Les fichiers de type *.xib* sont des fichiers **NIB**.

Les **Délégués** et les **Contrôleurs** sont des motifs de conception (*Design Pattern*) abondamment utilisés dans la programmation **Cocoa Touch** et que nous expliquerons très bientôt.

Les fichiers **NIB** contiennent des objets prêts à l'emploi, endormis en quelque sorte, qui sont réveillés lorsque le fichier est chargé dans l'application. L'interface utilisateur en particulier est définie dans les fichiers **NIB**, c'est pourquoi ces derniers sont édités avec l'outil **Interface Builder**. Découvrons sans plus tarder cet outil.

1.2. Composer l'interface utilisateur

Sous XCode, vérifiez que dans le menu *overview* de la barre d'outils, la valeur *Active SDK* est bien positionnée sur **iPhone Simulator 3.1.3**.

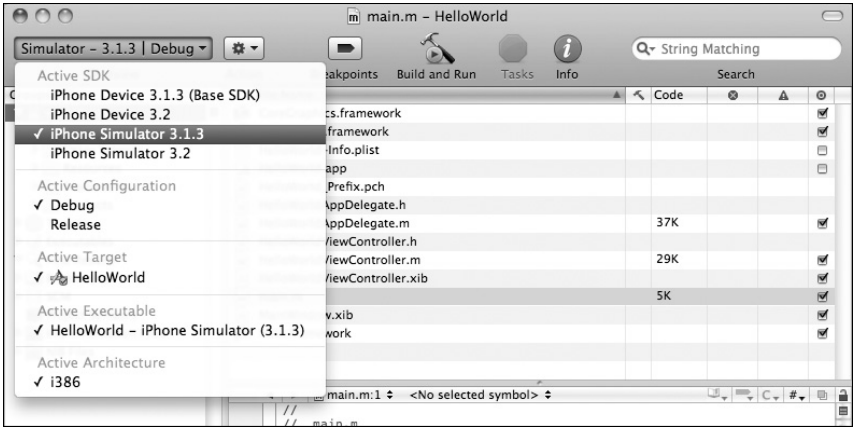


Figure 1.8 : Activation du SDK pour le simulateur d'iPhone

Vous modifierez la valeur du SDK actif pour tester votre application dans différents environnements :

- **Simulator 3.1.3** pour tester sur le simulateur d'iPhone ;
- **Simulator 3.2** pour tester sur le simulateur d'iPad ;
- **Device 3.1.3** pour tester sur un iPhone ou iPod Touch réels ;
- **Device 3.2** pour tester sur un iPad réel.

1 Sous XCode, double-cliquez sur le fichier *HelloWorldViewController.xib* dans la **Zone Détaillée** de la fenêtre de projet ; **Interface Builder** se lance.

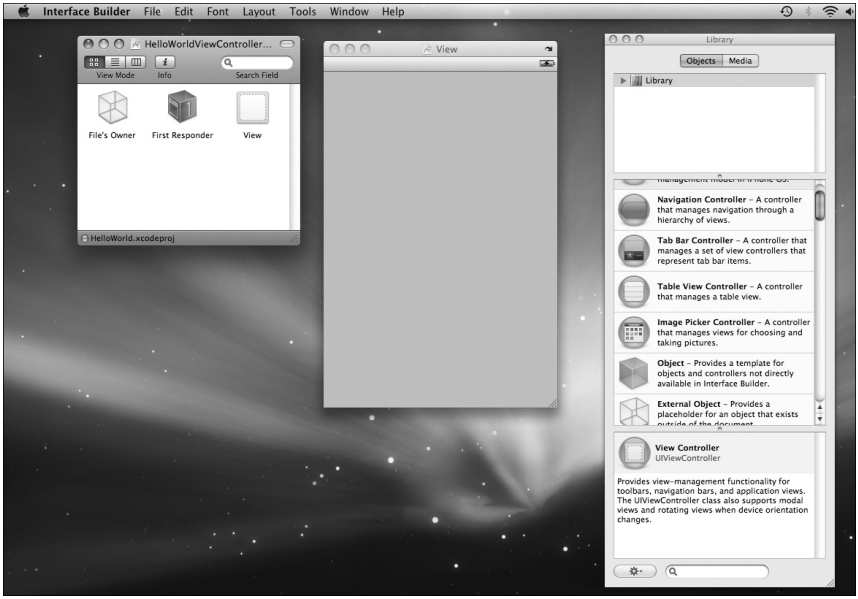


Figure 1.9 : Interface Builder

2 Trois fenêtres apparaissent. De gauche à droite :

- une fenêtre dont le titre est le nom du fichier NIB que l'on vient d'ouvrir, c'est le contenu du fichier NIB ;
- une fenêtre dans laquelle nous composerons la **Vue** (*View*) de notre interface utilisateur ;
- une fenêtre **Library** qui contient les objets que nous utiliserons pour composer l'interface utilisateur.

3 Cherchez l'objet *Label* dans la fenêtre **Library** et faites-le glisser sur la **Vue** de notre projet (voir Figure 1.10).

4 Sélectionnez le *Label* nouvellement déposé sur la **Vue** afin de le positionner où vous souhaitez (voir Figure 1.11).

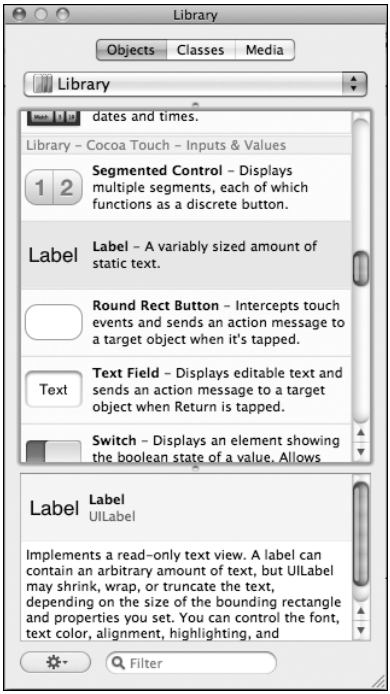


Figure 1.10 : Objet Label dans la fenêtre Library

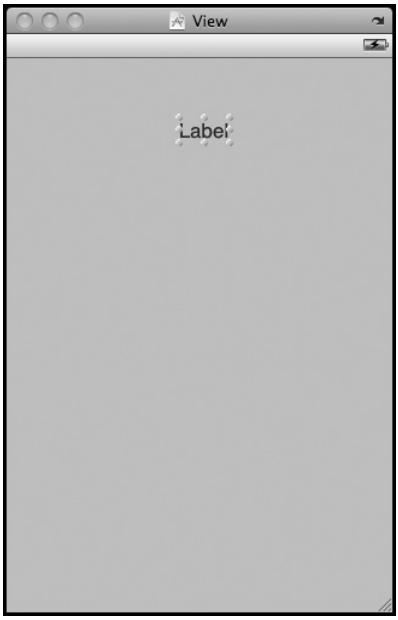


Figure 1.11 : Label positionné sur la Vue principale de l'application

5 

Double-cliquez sur le *Label* pour sélectionner le texte et saisissez `HelloWorld`. Enregistrez le fichier (commande **Save** du menu **File**), puis revenez dans **XCode** et cliquez sur le bouton **Build and Run** de la barre d'outils pour construire et lancer l'application.

6 Il est possible qu'une boîte de dialogue apparaisse pour signaler que nous n'avons pas enregistré tous les fichiers que nous avons modifiés. Cliquez alors sur le bouton **Save All**.

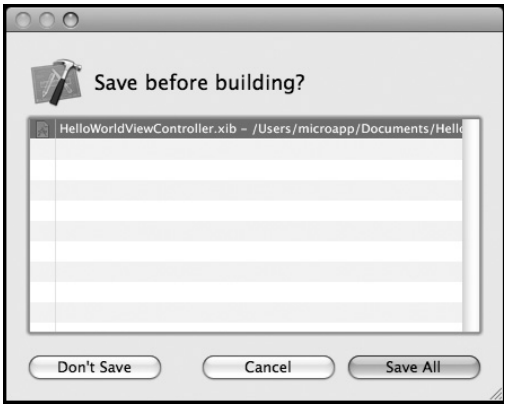


Figure 1.12 : Sauvegarder les fichiers modifiés

L'application *HelloWorld* est construite par XCode et lancée dans le simulateur d'iPhone

1.3. Tester l'application

iPhone Simulator est l'outil qui nous permet de tester les applications sur Mac. Il s'agit de la première étape de test. Bien sûr, il faudra tester l'application sur des appareils réels avant de la diffuser au public. Le test sur simulateur est intéressant ; il permet de déboguer plus facilement l'application (voir Figure 1.13).

Explorez les menus du simulateur. On peut basculer et même secouer virtuellement l'appareil. Une caractéristique intéressante est la possibilité de tester rapidement notre application sur des versions différentes d'iPhone OS : menu **Matériel**, sous-menu **Version**.

Explorez également le simulateur, comme si vous utilisiez votre iPhone (cliquez sur le gros bouton en bas de l'iPhone afin de revenir à l'écran d'accueil). Les applications **Photos**, **Contacts** et **Safari** fonctionnent normalement ; on pourra ainsi tester les applications utilisant les photos et les contacts de l'iPhone.

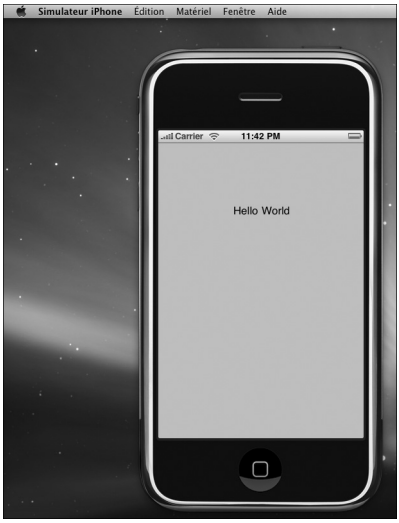


Figure 1.13 : Simulateur d'iPhone

Gestes avec 2 doigts

Les gestes avec 2 doigts sont simulés en maniant la souris tout en pressant la touche Alt du clavier. Essayez avec l'application Safari.

Vous pouvez aussi lancer votre application sur le simulateur d'iPad en sélectionnant le SDK **iPhone Simulator 3.2** sous XCode.

Figure 1.14 : Simulateur d'iPad

En explorant le simulateur, vous verrez que le logo permettant de lancer notre application HelloWorld est un carré blanc. Nous améliorerons cela immédiatement.

Figure 1.15 : Application sans logo

1.4. Finaliser l'application

Nous allons maintenant ajouter un logo à notre application *HelloWorld*.

Revenez dans l'application XCode et choisissez sur votre ordinateur l'image que vous souhaitez utiliser comme logo. Il est recommandé de choisir une image carrée au format *PNG*.

Changer d'application
Sur Mac OS X, on peut changer rapidement d'application par la combinaison de touches ⌘+⇥.

L'ajout du logo se fait en deux temps :

1. Ajoutez l'image au projet.
2. Déclarez cette image comme étant le logo de l'application.

Ajouter un fichier au projet

1. Dans XCode, sélectionnez *Resources* dans *HelloWorld* dans la zone *Groupes et Fichiers* de la fenêtre du projet et activez la commande **Add to project** ... du menu **Project**. Le panneau standard de Mac OS X permettant de choisir un fichier s'affiche.
2. Recherchez l'image désirée à l'aide de ce panneau et cliquez sur le bouton **Add**.

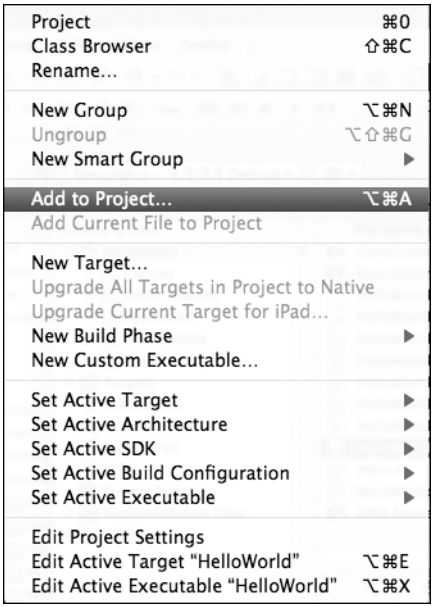


Figure 1.16 : menu Project de XCode

Un panneau s'affiche pour que vous puissiez préciser la façon dont vous souhaitez ajouter le fichier au projet.

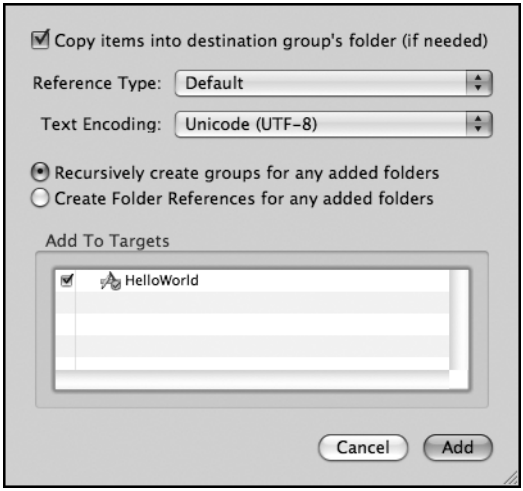


Figure 1.17 : Ajout d'un fichier au projet

Lorsque la case *Copy items into destination group's folder* de ce panneau est cochée, le fichier sélectionné est copié dans le dossier du projet et rangé dans le groupe sélectionné (en l'occurrence *Resources*).

3 Cochez cette case.

Par défaut, la case *HelloWorld* est cochée dans la liste *Add To Targets*. Cela signifie que le fichier que nous ajoutons sera copié dans le dossier des ressources de l'application *HelloWorld* lors de la construction.

4 Laissez cette case cochée et cliquez sur le bouton **Add**.

5 L'image apparaît dans la liste des fichiers du projet. Sélectionnez ce fichier dans la zone *groupes et fichiers* pour vérifier son contenu dans la zone d'édition.

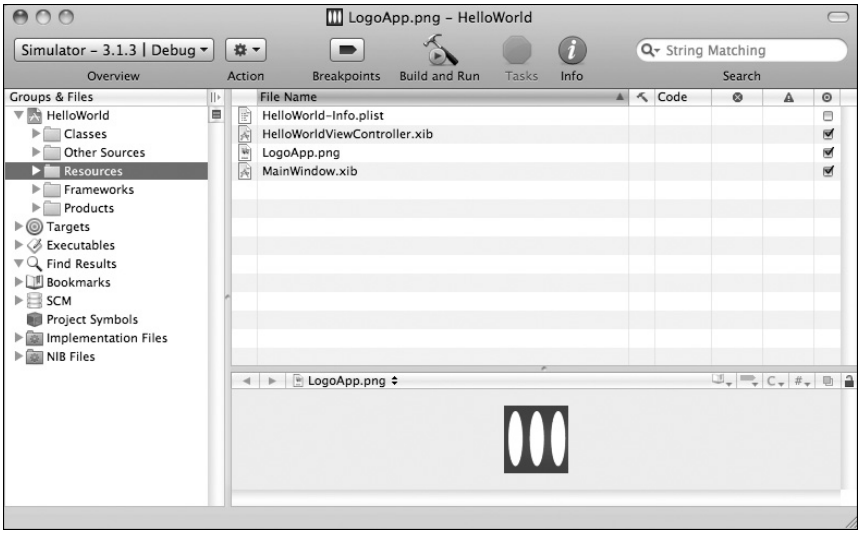


Figure 1.18 : L'image est ajoutée au projet

Vous savez maintenant ajouter un fichier à un projet. Vous procèderez exactement ainsi chaque fois que vous aurez besoin d'ajouter une image, une vidéo, un son ou tout autre fichier à un projet.

Déclarer le logo de l'application

Il faut maintenant indiquer que l'image que nous venons d'ajouter doit être utilisée comme logo de l'application *HelloWorld*.

1 Toujours sous XCode, sélectionnez le fichier *HelloWorld-Info.plist* dans le groupe *Resources*. Ce fichier contient les propriétés de l'application ; le logo est une de ces propriétés. Lorsque le fichier est sélectionné, son contenu apparaît dans la zone d'édition.

2 Sélectionnez la zone de texte à côté de la propriété *Icon File* et saisissez-y le nom du fichier contenant le logo.

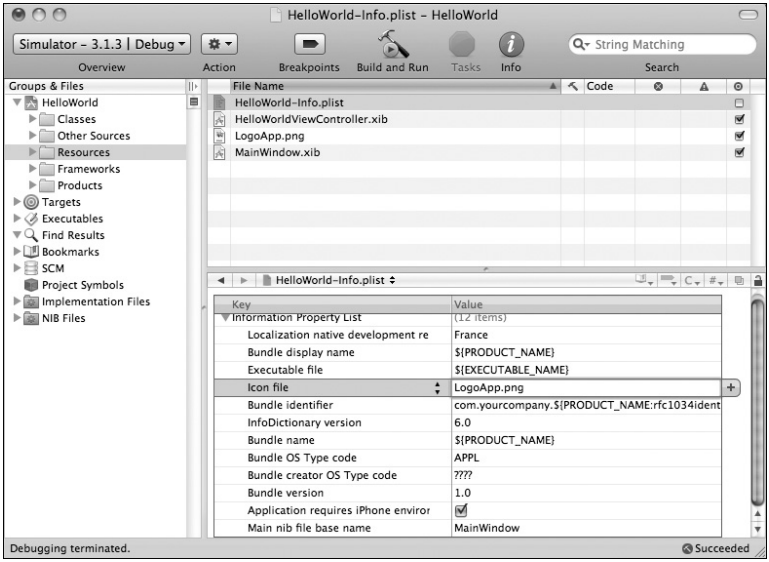


Figure 1.19 : Nom du fichier contenant le logo

3 Cliquez sur le bouton **Build and Go** de la barre d'outils de XCode. Si votre application est toujours en train de s'exécuter dans le simulateur, une boîte de dialogue s'affiche pour vous prévenir et vous demander si vous souhaitez arrêter l'exécution en cours. Cliquez sur OK.

Figure 1.20 : Arrêter l'exécution en cours

4 Le cas échéant, acceptez d'enregistrer les fichiers modifiés (vous venez de modifier *HelloWorld-Info.plist*). L'application s'exécute sur le simulateur d'iPhone. Cliquez sur le gros bouton du simulateur pour vérifier votre logo.

Figure 1.21 : HelloWorld avec logo

1.5. Agrémenter l'application

Nous n'allons pas nous arrêter en si bon chemin. Nous agrémenterons notre application par une illustration car pour l'instant, elle est un peu triste.

Choisissez une autre image et ajoutez-la au projet *HelloWorld* sous XCode.

Reportez-vous à la section précédente si vous ne vous souvenez plus comment on ajoute un fichier au projet.

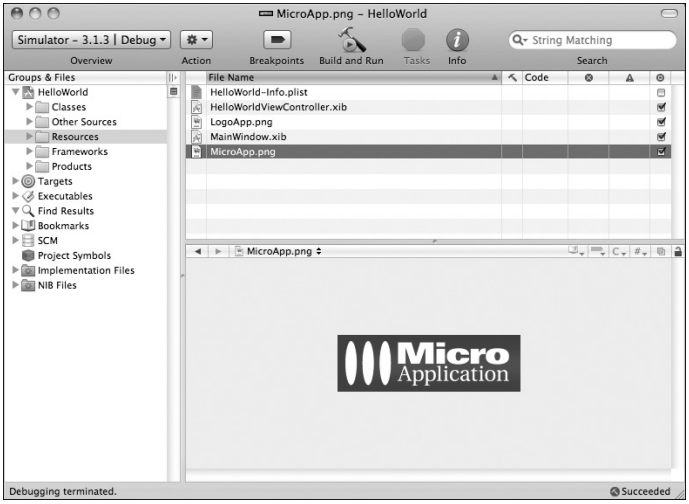


Figure 1.22 : Une nouvelle image est ajoutée au projet

Mettre notre image dans la vue

Pour rendre visible cette image lors de l'exécution de l'application, il faut retourner sous Interface Builder afin de modifier l'interface utilisateur.

1 Dans la fenêtre **Library** d'Interface Builder, cliquez sur le bouton **Media** pour obtenir la liste de tous les fichiers de type media (Image, Vidéo ou Son) de votre projet.

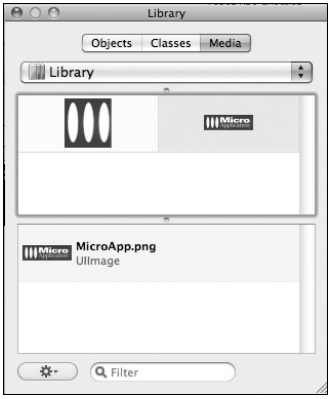


Figure 1.23 : Media du projet disponible sous Interface Builder

2 Faites glisser l'image souhaitée sous l'objet *Label* à l'aide de la souris. Si vous faites attention pendant ce positionnement, vous verrez apparaître des lignes pointillées de couleur bleue. Ces lignes de positionnement vous aide à aligner l'objet que vous êtes en train de déplacer.

Figure 1.24 : Application avec un Label et une Image

Image au lancement de l'application

Si votre projet contient une image dont le nom est *Default.png*, cette image sera automatiquement pendant le lancement de l'application. Nous nous en souviendrons lorsque nous bâtirons des applications un peu longues au démarrage.

1.6. Challenge

Avant de passer au chapitre suivant, nous vous invitons à modifier les attributs graphiques des objets *Label* et *Image* de notre interface utilisateur afin de découvrir les différentes possibilités.

Pour modifier les attributs d'un objet :

1 Activez l'**inspecteur** à l'aide du menu **Tools** sous Interface Builder et sélectionner le premier onglet de la fenêtre **Inspecteur**.

2 Sélectionnez l'objet sur lequel vous souhaitez travailler pour en visualiser les attributs dans l'inspecteur et les éditer.

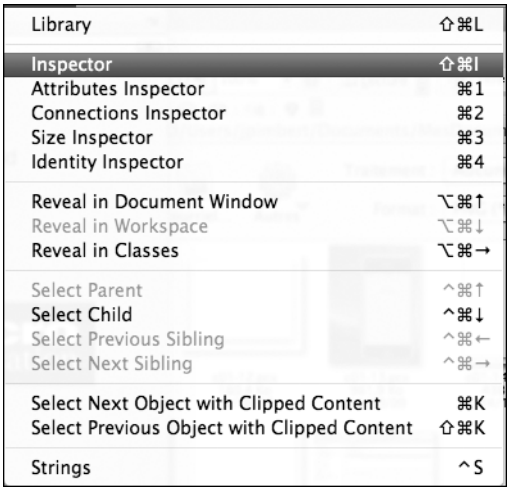


Figure 1.25 : Menu Tools d'Interface Buider

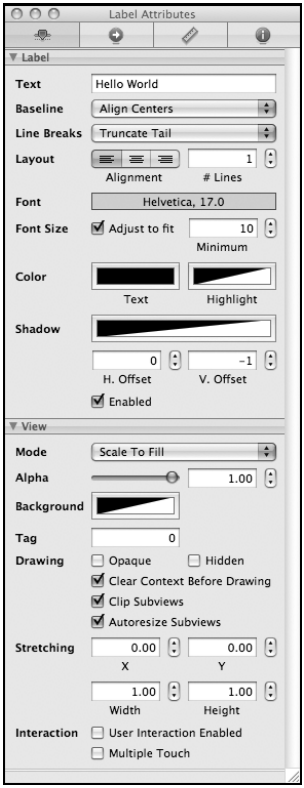


Figure 1.26 : Inspection des attributs de l'objet Label

1.7. Check-list

Nous venons de terminer notre première application pour iPhone. Pour cela, nous avons utilisé :

- **XCode** afin de :
  - créer un projet ;
  - ajouter des fichiers au projet ;
  - modifier les propriétés de l'application ;
  - construire l'application ;
  - lancer son exécution dans le simulateur.
- **Interface Builder** afin de :
  - composer l'interface utilisateur ;
  - ajouter un media à l'interface utilisateur.
- Et le **simulateur d'iPhone** pour tester notre application.

Au passage, nous remarquons que cette réalisation a été obtenue sans saisir une seule ligne de code Objective-C. C'est le résultat d'une caractéristique intéressante du framework **Cocoa Touch** et des modèles d'applications de **XCode** ; chaque modèle permet de construire une application qui fonctionne sans modification, le développeur se concentre sur l'écriture du code pour le comportement qu'il veut ajouter.

Notre application *HelloWorld* est imparfaite car nous ne pouvons pas interagir avec elle. Notre prochaine production sera plus satisfaisante et nous permettra d'écrire nos premières lignes de code.

INTERACTIONS SIMPLES

Notre objectif dans ce chapitre sera de réaliser une application pour notre prochain voyage aux États-Unis. Nous voulons connaître l'équivalent en euros des prix exprimés en dollars.

La première version de notre application se présentera ainsi. Nous la perfectionnerons ensuite.

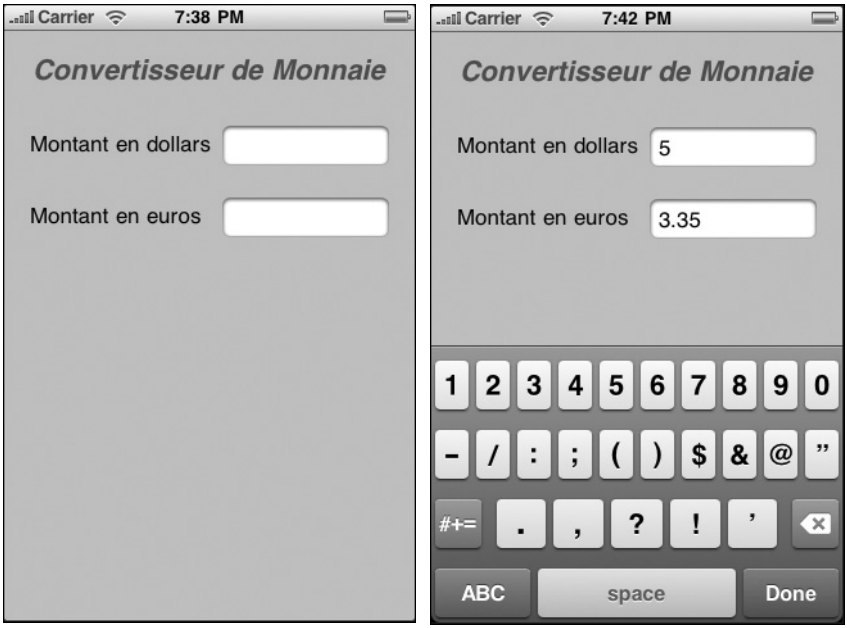


Figure 2.1 : Application Convertisseur1

Vous devrez d'abord patienter car il faut que vous compreniez ce qu'est la programmation orientée objet.

2.1. Programmation orientée objet

La programmation orientée objet est un style de programmation qui permet d'améliorer la testabilité, de faciliter la maintenance et donc de produire des logiciels de meilleure qualité aux fonctionnalités sont plus complexes. Nous n'exposerons pas ici la théorie de la POO (*Programmation orientée objet*). Notre ambition se limitera à comprendre comment nous utiliserons cette théorie dans le langage Objective-C, langage à objets, et avec les frameworks de Cocoa Touch, qui contiennent les objets prédéfinis dont nous aurons besoin. La terminologie POO employée sera celle du développement pour iPhone OS.

Nous traiterons ici les notions élémentaires ; d'autres notions plus avancées seront évoquées plus loin.

Objets

Un objet Objective-C permet de représenter un objet du monde "réel" ou manipulable par l'utilisateur. Par exemple, un label ou une image sur une interface sont des objets, l'application que nous développons est un objet, et nous aurons besoin plus loin de créer nos propres objets : un livre prêté, une monnaie à convertir, etc.

Les *objets* d'un langage de programmation comprennent :

- un *état*, c'est-à-dire la situation de l'objet à un moment donné de sa vie. Par exemple l'état d'un objet bouton pourrait contenir ;
  - sa position sur la fenêtre ;
  - l'image actuellement affichée ;
  - l'image à afficher si l'on clique sur le bouton, etc.
- un *comportement*, c'est-à-dire toutes les actions dont l'objet est capable :
  - cliquer sur un bouton (ou toucher le bouton pour se conformer à la terminologie iPhone OS).

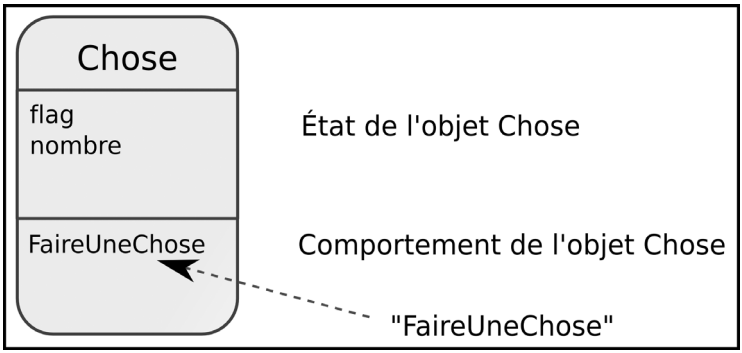


Figure 2.2 : Représentation d'un objet

Objets du langage Objective-C

Un objet comprend des *variables d'instance* et des *méthodes*. Les variables d'instance permettent de représenter l'état de l'objet, et les méthodes permettent de représenter son comportement.

Ne vous inquiétez pas si vous ne comprenez pas tout de suite les subtilités induites par cette définition ; cela viendra avec la pratique.

Les variables d'instance sont des variables du langage Objective-C. On peut également utiliser des variables du langage C puisque Objective-C en est une extension. Pourquoi parle-t-on d'instance ? Simplement pour préciser la portée de la variable ; elle est accessible uniquement depuis l'une des méthodes de l'objet (l'instance) et inaccessible depuis "l'extérieur" de l'objet.

Si vous avez besoin de vous rafraîchir la mémoire ou d'apprendre les bases du langage C, reportez-vous à l'*annexe B*.

Classes

Les informaticiens emploient rarement le terme *Objet* car il est ambigu (et les informaticiens n'aiment pas l'ambiguïté), ils emploient les termes *Classe* ou *Instance*.

Une classe est un modèle qui permet de reproduire des instances. La classe est un type d'objet, l'instance est un objet particulier de ce type.

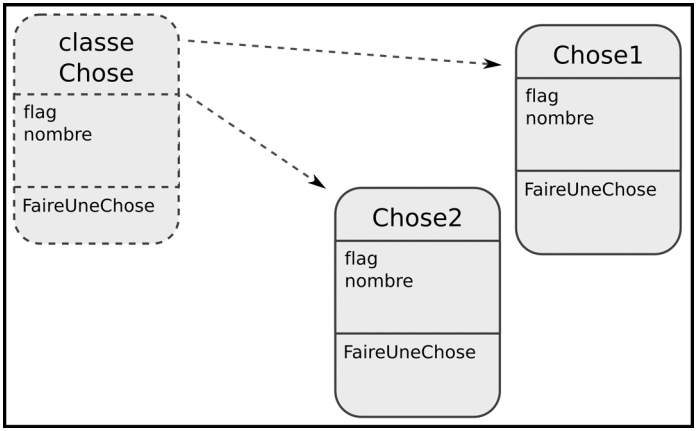


Figure 2.3 : Une classe et 2 instances du même type

Classe

Une classe est un modèle (*type*) d'objet qui permet de définir les variables d'instances et les méthodes qui devront exister pour toutes les instances de cette classe.

`Chose1` et `Chose2` sont deux instances de la classe `Chose`. Chaque instance possède ses propres variables d'instance, même si elles ont le même nom défini par la classe.

Messages

Lorsqu'une méthode est exécutée, elle doit pouvoir accéder aux variables d'instance de l'objet. Il faut donc que le programmeur précise, lorsqu'il écrit son programme, sur quelle instance il veut exécuter cette méthode. On dit que le programmeur envoie un *message* à l'objet.

Message
Un message est la demande transmise à une instance pour exécuter une méthode particulière dans le contexte de l'instance (en utilisant les variables d'instances lui appartenant).

Récepteur
Le récepteur d'un message est l'objet qui reçoit le message.

Après cet intermède d'explications théoriques, voyons comment on utilise ces concepts pour programmer sous Cocoa Touch, en commençant par le mécanisme cible-action.

2.2. Mécanisme Cible-Action

Nous allons mettre en pratique immédiatement le mécanisme cible-action en développant notre application *Convertisseur1*. La copie d'écran en début de chapitre montre le résultat auquel nous voulons arriver :

- un champ de texte dans lequel l'utilisateur saisit le montant en dollars ;
- un champ de texte dans lequel l'utilisateur peut lire le résultat de la conversion du montant en euros.

Nous souhaitons que la conversion soit réalisée pendant que nous inscrivons le montant en dollars ; la valeur en euros doit à tout instant être le résultat de la conversion de la valeur en dollars.

Nous avons donc besoin de trois objets :

- 2 instances de la classe `UITextField` pour les champs de texte ;
- 1 objet chargé de faire la conversion.

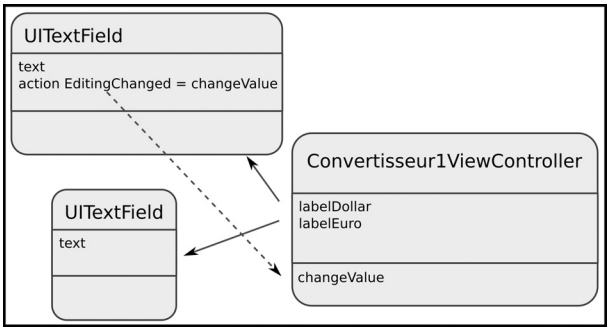


Figure 2.4 : Structure des objets pour l'application Convertisseur1

L'objet chargé d'effectuer les conversions sera une instance de la classe `Convertisseur1ViewController` ; nous verrons pourquoi dans un instant.

Créer les outlets

Notre `Convertisseur1ViewController` a besoin de connaître les deux champs de texte puisqu'il faudra qu'il "lise" la valeur en dollars et qu'il "écrive" la valeur en euros. Il a donc besoin de 2 variables d'instance de type "instance de classe `UITextField`".

1. Ouvrez XCode et créez un projet de type **View-based Application-** comme au chapitre précédent.
2. Intitulez ce projet `Convertisseur1` ; ce sera notre première version du Convertisseur de monnaies.
3. Sélectionnez le fichier *Convertisseur1ViewController.h* dans la fenêtre de projet et modifiez son contenu, dans la zone d'édition, pour obtenir le texte suivant :

```
#import <UIKit/UIKit.h>

@interface Convertisseur1ViewController :
UIViewController {
    IBOutlet UITextField *labelDollar;
    IBOutlet UITextField *labelEuro;
}
@property (retain,nonatomic) UITextField *labelDollar;
@property (retain,nonatomic) UITextField *labelEuro;

@end
```

Nous venons de définir deux variables d'instance, `labelDollar` et `labelEuro` de type `UITextField` pour les instances de la classe `Convertisseur1ViewController`. La ligne de code `IBOutlet UITextField *labelDollar;` signifie que `labelDollar` est une variable de type `UITextField *` (les adeptes du C liront "*est un pointeur sur une structure de type* `UITextField`") et que de plus, cette variable est un outlet (déclaré par `IBOutlet`).

Qu'est-ce qu'un outlet ? C'est simplement le moyen de dire à *Interface Builder* que l'on souhaite connecter cette variable d'instance ; nous allons expliquer cela. Au passage, notons que le "`IB`" de *IBOutlet* signifie *Interface Builder*.

Outlet

Un *outlet* est une sorte de variable d'instance, c'est un pointeur vers un autre objet. Un outlet est configurable à l'aide d'Interface Builder.

Après avoir saisi les quelques lignes de code précédentes dans le fichier *Convertisseur1ViewController.h*, l'icône de ce dernier est grisée dans la fenêtre de projet de XCode. Les icônes grisées signalent les fichiers qui ont été modifiés.

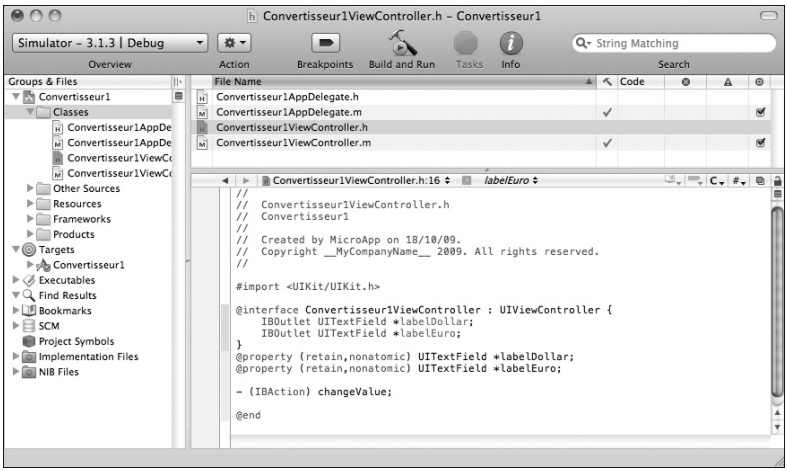


Figure 2.5 : Fichier Convertisseur1ViewController.h modifié

4 Sauvegardez le fichier que l'on vient de modifier soit par la combinaison de touches ⌘+S, soit en choisissant la commande **Save** du menu **File** sous XCode.

Cliquez dans la zone d'édition

Pour sauvegarder un fichier sous XCode, il faut qu'il soit sélectionné et il faut cliquer dans la zone d'édition de la fenêtre de projet.

Préparer l'interface utilisateur

1. Double-cliquez sur le fichier *Convertisseur1ViewController.xib* dans la fenêtre de projet sous XCode ; Interface Builder se lance.
2. Préparez l'interface utilisateur de l'application *Convertisseur1* à l'aide de 3 labels et de 2 champs de texte (*Text Field*).

Figure 2.6 : Interface utilisateur de Convertisseur1

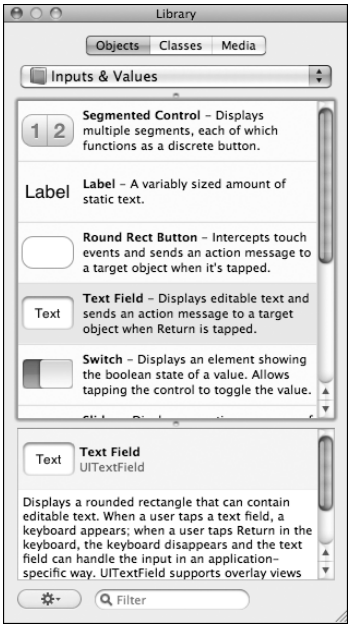


Figure 2.7 : Champ de texte

3 Si vous souhaitez que votre interface ait exactement le même aspect, utilisez l'inspecteur pour modifier les attributs graphiques des objets conformément au tableau.

Tableau 2.1 : Configuration des objets de l'interface

Champ	Type	Police	Couleur
Convertisseur de Monnaies	`Label`	*Helvetica Bold Oblique 24*	`Grape`
Montant en dollars	`Label`	*Helvetica 17*	Couleur par défaut
Montant en euros	`Label`	*Helvetica 17*	Couleur par défaut
-	`Text Field`	*Helvetica 17*	Couleur par défaut

Rappelez-vous ; pour modifier les attributs d'un objet, on utilise l'inspecteur après avoir sélectionné l'objet que l'on veut modifier.

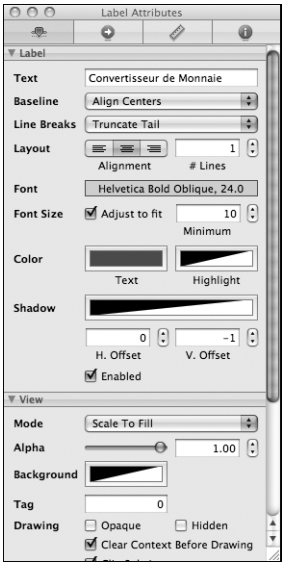


Figure 2.8 : Attributs du label de titre

4 Cliquez sur la description de l'attribut Font pour afficher la fenêtre flottante **Fonts.** Il faut cliquer sur la case de couleur *Color-Text* pour afficher la fenêtre flottante **Colors** (voir Figure 2.9).

Sélection multiple

Vous pouvez sélectionner simultanément plusieurs objets graphiques du même type (`label` ou `text field`) afin d'en modifier les attributs graphiques en une fois avec l'inspecteur. Cliquez sur le premier objet puis cliquez sur les suivants en maintenant la touche Maj enfoncée pour étendre la sélection.

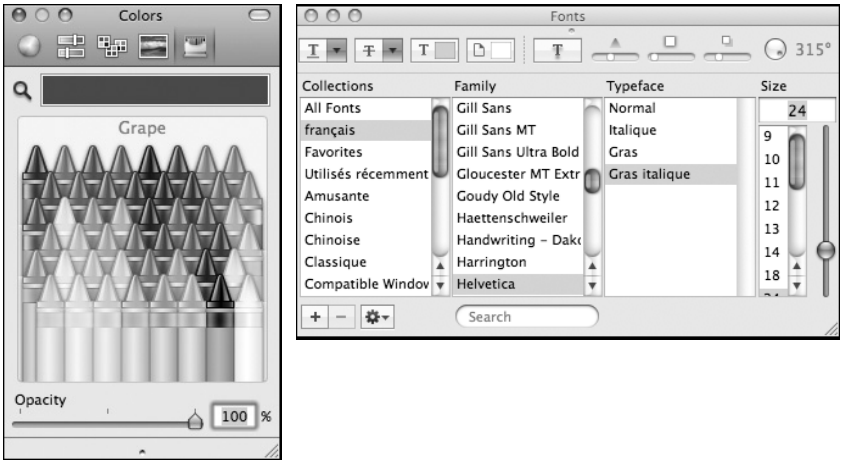


Figure 2.9 : Fenêtres flottantes d'attributs graphiques

Pour finaliser notre interface, nous allons indiquer que nous souhaitons que le clavier numérique s'affiche lorsque l'utilisateur touchera le champ de texte pour saisir le montant en dollars à convertir.

5 sélectionnez ce champ de texte sur la vue et choisissez *Numbers & Punctuation* dans la liste déroulante pour l'option **Keyboard** dans l'inspecteur .

Maintenant que notre interface utilisateur est terminée, nous pouvons connecter les outlets de notre application aux objets que nous venons d'agencer.

Figure 2.10 : Choix du clavier pour un champ de texte

Connecter les outlets

1 Toujours sous Interface Builder, sélectionnez *File's Owner* dans la fenêtre du contenu du fichier NIB puis sélectionnez l'onglet **Connections** de l'inspecteur (le deuxième onglet en partant de la gauche).

Figure 2.11 : Sélection de File's Owner

Nous découvrons dans l'inspecteur les deux outlets que nous avons ajoutés dans le fichier *Convertisseur1ViewController.h*.

2 Pour connecter un outlet à un objet de l'interface utilisateur, effectuez un cliquer-glisser-relâcher allant du petit cercle à droite de l'outlet dans l'inspecteur jusqu'à l'objet que vous souhaitez rattacher.

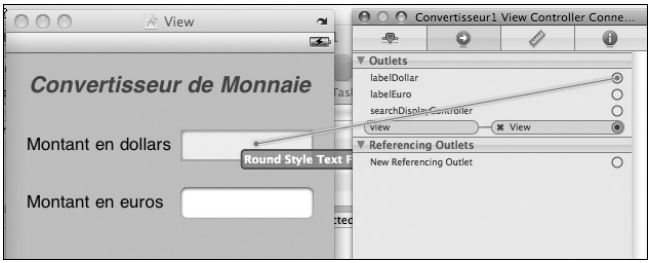


Figure 2.12 : Connexion d'un outlet à un objet

3 Connectez l'outlet *labelDollar* au champ de texte à côté du label *Montant en dollars* puis connectez l'outlet *labelEuro* au champ de texte à côté du label *Montant en euros*.

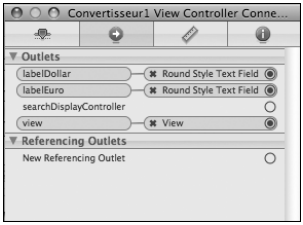


Figure 2.13 : Les outlets connectés

Tester l'interface

La combinaison de touches ⌘+R sous Interface Builder provoque le lancement du simulateur d'iPhone pour visualiser l'interface que nous venons de construire. L'application ne fonctionne pas mais nous pouvons activer tous les objets de l'interface pour vérifier leur comportement graphique.

Notre objet comprend 4 outlets alors que nous n'en avons défini que 2. Les outlets supplémentaires sont obtenus par *héritage*. Nous étudierons ce concept important dans quelques pages.

Pour le moment, nous terminons l'application *Convertisseur1*.

Déclarer les actions

Nous avons construit l'interface utilisateur de notre application *Convertisseur1*, nous lui avons indiqué comment communiquer avec le cœur de l'application en définissant des outlets et en établissant les connexions. Il nous faut maintenant définir précisément le comportement de l'application.

Revenez dans XCode et complétez le fichier *Convertisseur1 ViewController.h* de la façon suivante :

```
#import <UIKit/UIKit.h>

@interface Convertisseur1ViewController : UIViewController {
    IBOutlet UITextField *labelDollar;
    IBOutlet UITextField *labelEuro;
}
@property (retain,nonatomic) UITextField *labelDollar;
@property (retain,nonatomic) UITextField *labelEuro;

- (IBAction) changeValue;

@end
```

Nous venons de déclarer une *action* dont le nom est `changeValue`.

Action

Une action est une méthode d'instance dont la vocation est d'être exécutée lorsqu'un événement survient. Elle est configurable à l'aide d'Interface Builder.

Nous souhaitons que cette action soit appelée chaque fois que l'utilisateur change le montant en dollar, et qu'elle calcule le montant en euros puis l'affiche dans le champ adéquat. La connexion entre l'action et l'événement sera réalisée à l'aide d'Interface Builder. Le comportement de l'action est décrit en Objective-C dans la définition de l'action.

Définir les actions

Sauvegardez le fichier *Convertisseur1ViewController.h* que nous venons de modifier et, toujours sous XCode, ouvrez le fichier *Convertisseur1ViewController.m*.

Déclaration et Définition

À l'instar des langages C et C++, les déclarations et les définitions sont séparées en Objective-C. Par convention, une classe d'objets `NomDeLaClasse` est décrite dans 2 fichiers sources *NomDeLaClasse.h* et *NomDeLaClasse.m*.

Règle de nommage des classes

Afin de faciliter la lecture du code source, il est d'usage d'adopter des règles de nommage. En particulier, on essaiera de trouver un nom explicite ; il faut éviter les noms abrégés (par exemple `NdlC` au lieu de `NomDeLaClasse`) ; ils nuiront à la lisibilité et donc à notre capacité à modifier le code dans l'avenir.

En Objective-C, le nom d'une classe est une série de mots accolés et chaque mot commence par une majuscule y compris le premier mot, par exemple : `NomDeLaClasse`.

Le fichier avec l'extension *.h* contient la *déclaration* de la classe, *interface* en anglais, c'est-à-dire tout ce qui est nécessaire pour utiliser une instance de cette classe ; le type et le nom de chaque variable d'instance et de chaque méthode.

Le fichier avec l'extension *.m* contient la *définition* de la classe, *implementation* en anglais, c'est-à-dire le détail du comportement de l'objet.

Encapsulation

L'encapsulation est le principe selon lequel on doit pouvoir utiliser un objet sans connaître le détail de la façon dont cet objet travaille.

Pour illustrer le principe d'encapsulation, prenons un exemple : lorsque je veux démarrer ma voiture je tourne la clé de contact et le moteur se met en marche ; j'utilise l'interface, c'est simple. Imaginons tout ce que je devrais faire si je n'avais pas cette interface ; mettre en marche la pompe à essence, régler la richesse du mélange en fonction de la température du moteur, mettre le pignon du démarreur en contact avec l'arbre du moteur, faire tourner le démarreur pour lancer le moteur, activer les soupapes de façon synchronisée avec la position des pistons et dans le même temps injecter le mélange dans le moteur, déclencher les explosions dans les cylindres au bon moment (toujours en fonction de la température du moteur et enfin expulser les gaz brûlés. Tous les objets, y compris les objets Objective-C, devraient être aussi simples à utiliser que ma voiture.

L'encapsulation est importante pour faciliter la maintenance des applications. Lorsque nous sommes amenés à modifier un objet, nous n'avons pas nécessairement besoin de modifier tous les objets qui l'utilisent ; nous pouvons changer de voiture, nous n'avons pas besoin de changer notre façon de la démarrer.

Inclusion des déclarations

Vous n'avez peut-être pas fait attention mais nous avons déjà utilisé le principe d'encapsulation. Dans notre fichier *Convertisseur1ViewController.h*, nous employons les noms prédéfinis `IBOutlet`, `IBAction`, `UIViewController` et `UITextField`. Où sont définis tous ces noms ? Dans le framework *UIKit* d'iPhone OS. Comment indique-t-on qu'il faut utiliser ce framework ? La première instruction du fichier indique qu'il faut employer les déclarations du framework UIKit : `#import <UIKit/UIKit.h>`.

#import

`#import` demande au compilateur d'inclure un fichier dans le fichier courant. Cette instruction est utilisée principalement afin de récupérer les déclarations nécessaires pour employer un framework ou des objets définis ailleurs.

Syntaxe : `#import FichierAInclure`

`Fichier global` Le chemin d'accès au fichier à inclure doit être mis entre crochets lorsqu'il se situe dans la bibliothèque des frameworks d'iPhone OS. Par exemple, `#import <UIKit/UIKit.h>`.

`Fichier local` Le nom du fichier à inclure doit être mis entre guillemets lorsqu'il se situe dans le même dossier que le fichier dans lequel il est inclus, par exemple :

```
#import. "Convertisseur1ViewController.h"
```

Lorsque nous avons créé notre projet *Convertisseur1* de type **View-based Application**, XCode a créé pour nous une classe *Convertisseur1ViewController* et ses 2 fichiers *.h* et *.m*. C'est d'abord dans cette classe que nous devons introduire le comportement de notre application et c'est pourquoi nous modifions les fichiers *Convertisseur1ViewController.h* et *Convertisseur1ViewController.m*. Le fichier *.h* sera inclus partout où nécessaire. Vous devinez où il faut l'inclure en tout premier lieu ? Dans le fichier *.m*, bien sûr ; cela permettra au compilateur de vérifier que la déclaration et la définition de notre classe sont cohérentes.

#import et #include

Les développeurs C utilisent la clause `#include`. `#import` joue exactement le même rôle mais en évitant d'inclure plusieurs fois le même fichier. *UIKit.h*, par exemple, est inclus dans presque tous les *.h*. Dès que l'on inclut deux fichiers *.h* dans le même fichier *.m*, on y inclut plusieurs fois le fichier *UIKit.h* (les inclus de mes inclus sont mes inclus) ; le compilateur signalerait alors des erreurs car il n'aime pas que les mêmes noms soient déclarés plusieurs fois. La clause `#import` évite ce genre d'inconvénient.

Regardez le contenu du fichier *Convertisseur1ViewController.m*, XCode l'a préparé pour nous et il commence par l'instruction `#import "Convertisseur1ViewController.h"`.

Basculer entre déclaration et définition

Sous XCode, la combinaison de touches ⌘+⌥+⇧ permet de basculer du fichier *.h* vers le fichier *.m* et réciproquement. Cette astuce est très utile ; vous aurez souvent à passer de l'un à l'autre pendant la saisie du code source.

Il nous reste à terminer notre classe *Convertisseur1ViewController*.

Définition de l'action changeValue

Si ce n'est pas déjà fait, sélectionnez le fichier *Convertisseur1View Controller.m* pour en visualiser le contenu dans la zone d'édition de XCode. Modifiez-le afin d'obtenir le code suivant :

```
#import "Convertisseur1ViewController.h"

@implementation Convertisseur1ViewController

@synthesize labelDollar;
@synthesize labelEuro;

- (IBAction) changeValue {
    NSString *textDollar = labelDollar.text;
    float dollar = [textDollar floatValue];
    float euro = dollar / 1.4908;
    NSString *textEuro = [[NSString alloc]
                          initWithFormat: @"%.2f",euro];
    labelEuro.text = textEuro;
    [textEuro release];
}
```

Lorsque vous saisissez ce code source (veillez à le saisir très précisément tel que nous vous l'indiquons) vous constatez que XCode vous propose de compléter les mots au fur et à mesure de la frappe. Cette fonctionnalité de XCode est la **Terminaison de Code** (*Code Completion*) et permet de gagner beaucoup de temps lors de la saisie de code source.

Saisie des crochets droits et des accolades

Les accolades { et } sont obtenues par les combinaisons de touches ⌥+(et ⌥+). Les crochets droits [et] sont obtenus par les combinaisons de touches Maj+⌥+(et Maj+⌥+).

Utilisation de la Terminaison de Code

Lors de la saisie, si le mot proposé par XCode vous convient, pressez la touche ⇥. S'il ne vous convient pas, continuez la saisie ou pressez la touche Échap afin d'obtenir une liste de suggestions. Vous pouvez alors sélectionner le mot approprié.

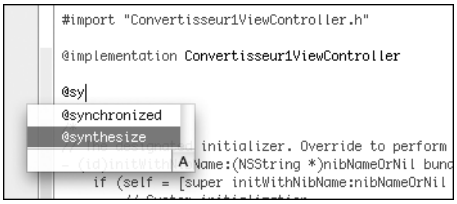


Figure 2.14 : Suggestions de la terminaison de code

Nous comprendrons bientôt de façon détaillée ces quelques lignes de code et leur syntaxe particulière, et vous serez même capable d'écrire la fonction *changeValue* en seulement une ou deux lignes.

Nous avons préféré décomposer les instructions pour vous aider à en suivre la logique :

- Les instructions `@synthesize` servent à générer les accesseurs pour les propriétés *labelDollar* et *labelEuro*.
- L'action *changeValue* est ensuite définie par les instructions entre les accolades `{` et `}`, soit dans l'ordre :
  - Un objet de type *NSString* est déclaré et initialisé avec la valeur écrite par l'utilisateur dans le champ de texte *labelDollar*.
  - Un nombre de type *float* est déclaré et initialisé avec le montant en dollars. (Nous avons besoin d'un nombre de type *float* pour faire notre calcul de conversion en euros car on ne peut pas effectuer de calculs sur un objet de type *NSString*.)
  - Un nombre de type *float* est déclaré et initialisé avec le montant converti en euros, avec un taux de conversion de 1,4908 $ pour 1 € (c'est le taux officiel à la date de rédaction de ces lignes).
  - Un objet de type *NSString* est créé et initialisé avec une chaîne de caractères représentant le montant en euros, avec une précision de 2 chiffres après la virgule.
  - Le montant en euros est affiché dans le champ de texte *labelEuro*, l'utilisateur peut le lire.
  - L'objet *NSString textEuro* est détruit car à chaque appel de *changeValue*, un nouvel objet est créé ; il faut éviter les fuites de mémoire.

Nous voyons ici l'intérêt d'avoir défini des outlets ; notre action `changeValue` peut "lire" et "écrire" les contenus des champs de texte connectés aux outlets.

Comme c'est notre premier bout de code, il n'est pas certain que nous l'ayons saisi correctement. Sous XCode, enregistrez le fichier modifié et construisez l'application sans lancer l'exécution.

Vérifier le code source

On peut vérifier le code source de l'application en la construisant sans l'exécuter. Sous XCode, sélectionnez la commande **Build** du menu **Build** ou tapez la combinaison de touches ⌘+B.

Par exemple, si nous avons saisi `LabelEuro` à la place de `labelEuro`, l'erreur nous est signalée dans la barre d'état de la fenêtre de projet et dans le code source.

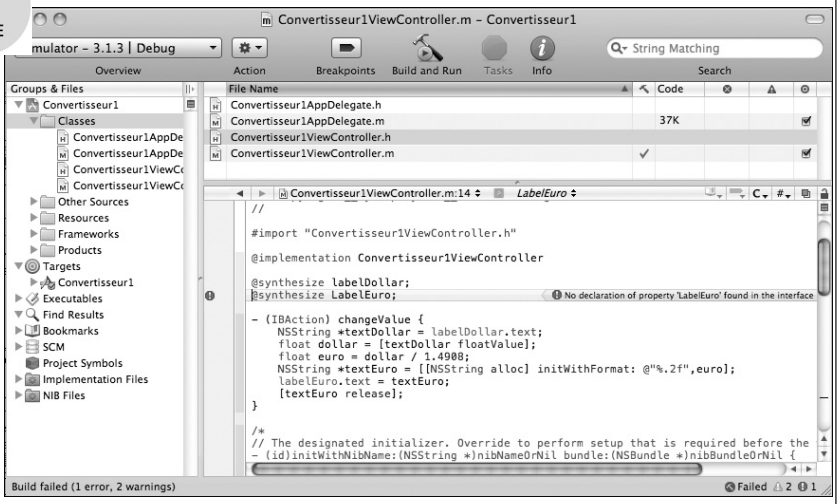


Figure 2.15 : Erreur signalée à la construction

Vérifiez que votre code source ne contient pas d'erreurs et corrigez ces dernières éventuellement avant de passer à l'étape suivante.

Connecter les cibles

1 Retournez vers l'application Interface Builder pour modifier le fichier *Convertisseur1ViewController.xib*. Sélectionnez le champ de texte dans lequel nous aurons le montant en dollars et visualisez l'onglet **Connections** de l'inspecteur.

Nous allons indiquer à Interface Builder que nous souhaitons que l'action `changeValue` du propriétaire du fichier *Convertisseur1View Controller.xib* (c'est-à-dire une instance de la classe *Convertisseur1View Controller*) soit déclenchée à chaque modification, par l'utilisateur, du contenu du champ de texte contenant le montant en dollar. Nous allons pour cela utiliser le mécanisme **Cible-Action** (*Target-Action*) de Cocoa-Touch. Pour mettre en œuvre ce mécanisme nous avons besoin :

- d'un objet dont les événements sont observés ; le champ de texte contenant le montant en dollars ;
- d'un événement qui va déclencher l'action ; ici ce sera l'événement *Editing Changed* ;

- d'un objet qui va recevoir l'action – la **cible** ; notre instance de la classe *Convertisseur1ViewController* ;
- d'une méthode à activer sur la cible – l'**action** ; la méthode *changeValue*.

2 Effectuez un cliquer-glisser-relâcher allant du petit cercle à droite de l'événement *Editing Changed* dans l'inspecteur jusqu'à l'objet *File's Owner* dans la fenêtre du contenu du fichier NIB.

Figure 2.16 : Connexion d'une cible

Une liste sur fond gris s'affiche en surimpression sur la cible que nous avons sélectionnée.

3 Cliquez sur *changeValue*, l'unique ligne de cette liste. Cette liste nous permet de choisir une action parmi celles définies pour la cible. Comme nous avons déclaré une seule action sur notre cible, la liste comprend une seule ligne.

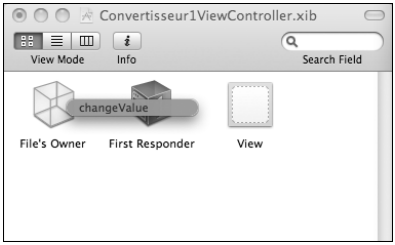


Figure 2.17 : Liste des actions définies sur la cible

4 Sélectionnez à nouveau *File's Owner* pour visualiser ses connexions dans l'inspecteur.

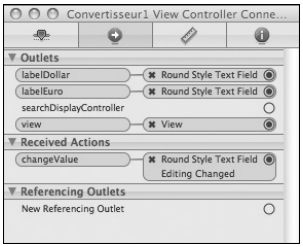


Figure 2.18 : Connexions de File's Owner

Construire et tester l'application Convertisseur1

1. Sauvegardez le fichier *NIB* sous Interface Builder et revenez sous XCode pour construire et lancer l'application (⌘+R).
2. Si la construction ne fonctionne pas correctement, vérifiez que le SDK actif est le simulateur et vérifiez votre code source.

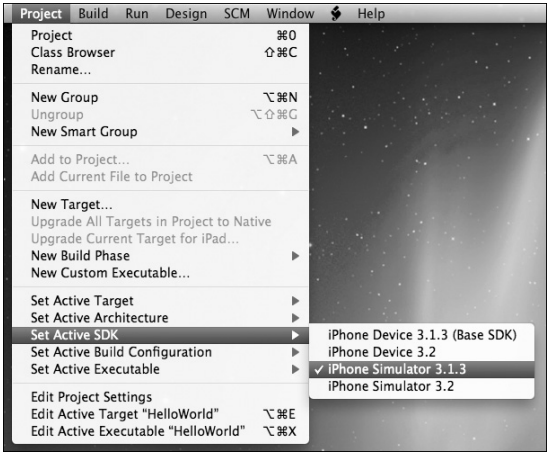


Figure 2.19 : Vérifiez que le SDK actif est le simulateur

3. Testez votre application sur le simulateur d'iPhone. N'hésitez pas à tout essayer : saisir des chiffres et des lettres, dans le champ des montants en dollars et dans celui des montants en euros, etc.

Figure 2.20 : Test de l'application Convertisseur1

Nous nous rendons compte de tous les petits défauts de notre application. Il faudra l'améliorer avant de réellement l'utiliser :

- empêcher que l'utilisateur tape des lettres ;

- pouvoir faire disparaître le clavier ;
- pouvoir effectuer les conversions dans les deux sens, des dollars en euros et vice-versa ;
- lorsqu'on revient dans l'application, retrouver les montants tels qu'ils étaient lorsqu'on l'a quittée ;
- pouvoir modifier facilement le taux de conversion.

Nous aborderons ces améliorations après avoir approfondi ce que nous venons de voir.

2.3. Hiérarchie des classes de Convertisseur1

Vous êtes sans doute un peu frustré car nous venons de créer une application mais vous n'avez sans doute pas compris tous les détails du code que nous avons écrit. Nous allons approfondir tout cela et vous comprendrez mieux le code de *Convertisseur1* à la fin de ce chapitre.

Héritage

Notre classe *Convertisseur1ViewController* offre deux outlets supplémentaires que nous n'avons pas déclarés. Ils appartiennent à notre classe par héritage. Nous avons représenté l'arbre d'héritage des objets que nous avons manipulés pour notre application *Convertisseur1* (voir Figure 2.21).

Convertisseur1ViewController hérite de ou "est une sorte de" *UIViewController*. En plus des variables d'instances et des méthodes que nous avons définies pour *Convertiseur1ViewController*, ce dernier possède également les attributs et le comportement de *UIViewController* ; en particulier les deux outlets supplémentaires que nous avons vus précédemment.

Notre classe *Convertisseur1ViewController* contient, par héritage, tout le code des classes *UIViewController*, *UIResponder* et *NSObject*.

Examinons plus attentivement la classe *UITextField* :

- `NSObject` est la classe de base ; tous les objets doivent dériver (doivent hériter) de NSObject.
- `UIResponder` ; un **Répondeur** est un objet possédant la capacité de recevoir des événements et de les traiter ou de les transmettre soit

par le mécanisme *Cible-Action*, soit au répondeur suivant dans la chaîne des répondeurs. Nous expliquerons plus loin la chaîne de répondeurs.

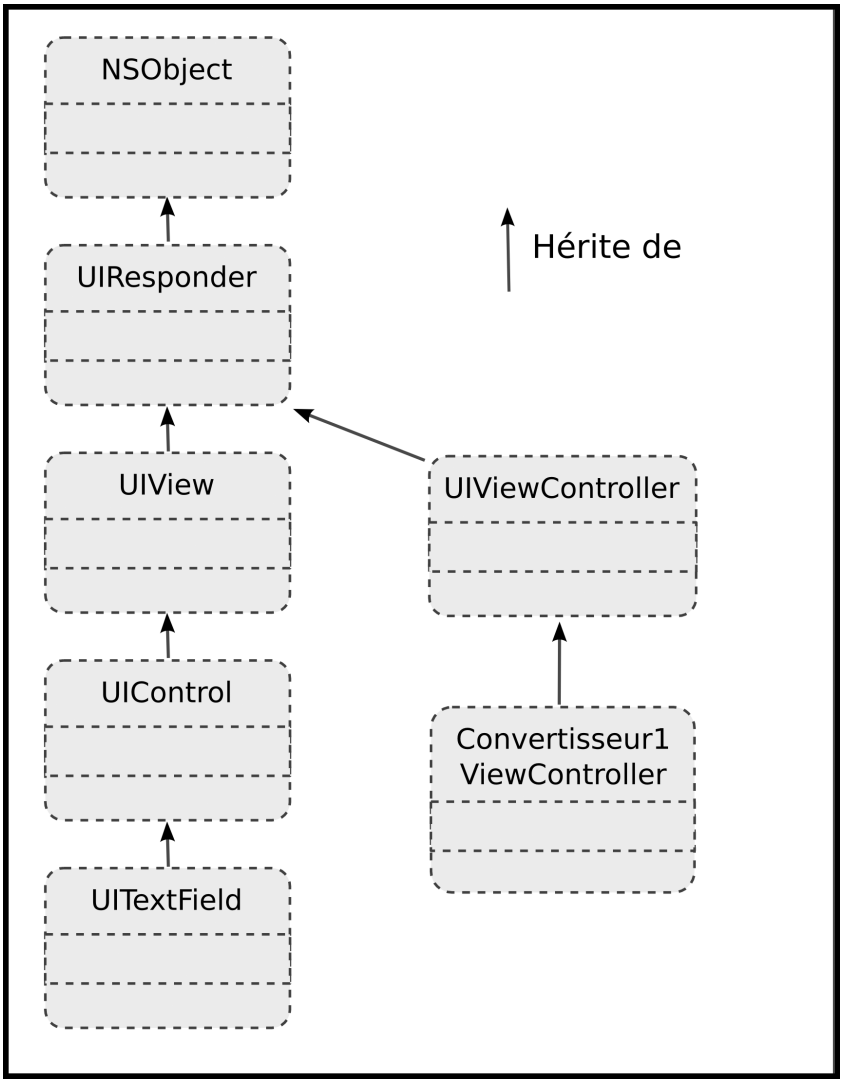


Figure 2.21 : Graphe d'héritage de Convertisseur1

- `UIView` ; une **Vue** est un objet qui apparaît à l'écran et appartient à la hiérarchie des vues (voir ci-après). Une vue est également un répondeur puisqu'elle reçoit et réparti les événements créés par le système lorsque l'utilisateur touche cette vue sur l'écran.

- `UIControl` ; un **Contrôle** est une vue particulière qui présente une liste d'événements spécifiques et peut activer une action sur une cible pour chacun de ces événements.
- `UITextField` ; un champ de texte est un contrôle particulier (et donc aussi une Vue et un Répondeur) qui permet à l'utilisateur de visualiser et modifier une ligne de texte.

Et `UILabel` ? Vous devinez de quoi il hérite ? Réfléchissons… C'est un objet, cela s'affiche sur l'écran, c'est donc vraisemblablement une vue. Est-ce un contrôle ? Peut-on configurer le mécanisme Cible-Action à partir d'un `UILabel` ? Vous pouvez consulter l'onglet **Connections** de l'inspecteur sous Interface Builder, après avoir sélectionné l'un des deux labels de l'application *Convertisseur1*, vous constaterez qu'aucun événement n'est défini pour un `UILabel` ; on ne peut pas définir des Cibles-Actions pour un label. Donc `UILabel` hérite de `UIView` mais pas de `UIControl`.

Hiérarchie des vues

Chaque application d'un iPhone affiche une fenêtre unique qui occupe tout l'écran. Une fenêtre est un objet de type `UIWindow` qui hérite de `UIView` (une fenêtre est un type particulier de vue). Cette fenêtre contient généralement une vue qui elle-même contient une ou plusieurs vues et qui, à leur tour, peuvent contenir des vues, etc. La fenêtre et toutes les vues incluses constituent la **hiérarchie des vues**. La hiérarchie des vues peut évoluer pendant l'exécution de l'application. Par exemple dans l'application **Contacts**, lorsque l'utilisateur passe de la liste des contacts à la visualisation d'une fiche des contacts, l'apparence visuelle de l'interface évolue ; la hiérarchie des vues a changé.

Vous pouvez visualiser la hiérarchie de vues de l'application *Convertisseur1* sous Interface Builder, dans la fenêtre du contenu du fichier *NIB*. Cliquez sur le bouton du milieu de la rubrique **View Mode**. Cliquez sur le triangle à côté de l'objet `View` pour visualiser son contenu (voir Figure 2.22).

Il n'y a pas de fenêtre de type `UIWindow` dans le fichier *Convertisseur1ViewController.h*. La racine de la hiérarchie est une vue dont le nom est *View*, elle contient les 5 labels et les 2 champs de texte que nous y avons ajoutés. La fenêtre se trouve en fait dans le fichier *MainWindow.xib*, vous pouvez ouvrir ce fichier pour vous en convaincre.

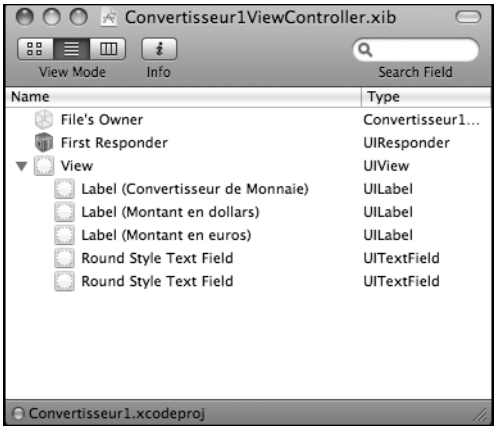


Figure 2.22 : Hiérarchie des vues de Convertisseur1

Toutes les applications pour iPhone sont structurées de la même façon :

- une fenêtre unique (`UIWindow`) qui restera affichée pendant toute l'exécution de l'application (elle est décrite dans le fichier *MainWindow.xib*) ;
- une ou plusieurs vues principales qui occupent chacune toute la fenêtre (chacune de ces vues est décrite dans un fichier NIB spécifique).

Nous avons créé un projet de type **View-based Application** sous XCode pour construire notre application *Convertisseur1*. Ce type d'application ne comprend qu'une vue principale, nous aurons l'occasion de créer d'autres types d'application avec plusieurs vues principales.

2.4. Manipulation des objets en Objective-C

Après ces quelques éléments théoriques, voyons comment on manipule les objets dans le langage Objective-C.

Déclaration

La **déclaration** d'une classe s'effectue dans un fichier source qui porte le nom de la classe et dont l'extension est *.h*. Par exemple *MaClasse.h* pour la classe *MaClasse*. Une déclaration suit toujours le même schéma :

- Une instruction `@interface` précisant le nom de classe déclarée et le nom de la classe dont elle hérite ; cette dernière est appelée la superclasse de la classe en cours de création. Toutes les classes doivent dériver d'une superclasse.
- Un bloc, délimité par des accolades, contenant les déclarations de chaque variable d'instance de la classe.
- Les déclarations de chaque méthode de la classe.
- L'instruction `@end` pour indiquer la fin du bloc `@interface` :

```
@interface MaClasse : SuperClasse
{
    // déclaration des variables d'instance
}
//    déclaration des méthodes
@end
```

Commentaires

Sur chaque ligne de code source, le texte à partir de la double barre oblique "`//`", jusqu'à la fin de la ligne, est considéré comme un commentaire par le compilateur.

Déclaration des variables d'instance

Chaque variable d'instance a un **nom** et un **type**. Vous choisissez le nom de chaque variable ; c'est ce nom que vous utiliserez pour employer cette variable. Bien sûr, deux variables d'instances ne peuvent porter le même nom.

```
type nom1, nom2, …, nomN ;
```

On peut déclarer une ou plusieurs variables dans la même instruction ; on utilise la virgule pour séparer les variables. L'instruction de déclaration se termine par un point-virgule.

Ne pas oublier le point-virgule

L'oubli du point-virgule en fin d'instruction est une erreur courante que même les programmeurs confirmés peuvent faire. Le point-virgule est le marqueur de fin d'instruction du langage C et du langage Objective-C.

Vous pouvez utiliser :

- N'importe quel type défini dans le langage C (`int`, `long`, `float`, `double`, pointeur, etc.) ou un type élaboré à l'aide des règles du langage C (`typedef`).

- Un pointeur sur une classe d'objets, par exemple `labelDollar` de type `UITextField *` ; un pointeur sur une classe d'objet permet de manipuler les instances de cette classe ;
- Le type `id`, qui est un type prédéfini dans Objective-C, pointeur vers une classe non précisée ; on utilise le type id pour manipuler des instances dont on ne connaît pas la classe.

Règle de nommage des variables

Comme les classes, les variables doivent porter un nom explicite et il faut éviter les noms abrégés (par exemple `lD` au lieu de `labelDollar`).

En Objective-C, le nom d'une variable est une série de mots accolés et chaque mot commence par une majuscule sauf la première lettre qui reste minuscule, par exemple : `labelDollar`.

Déclaration des méthodes

Nous arrivons aux caractéristiques d'Objective-C les plus déroutantes pour les programmeurs C ou C++ : la déclaration et l'appel des méthodes.

Commençons par le plus facile, la déclaration d'une méthode qui n'a pas de paramètres :

```
// déclaration d'une méthode d'instance
- (type-de-la-valeur-de-retour) nomDeLaMethode ;
// déclaration d'une méthode de classe
+ (type-de-la-valeur-de-retour) nomDeLaMethode ;
```

Comme en C, le type est `void` si la méthode ne retourne pas de valeur. `IBAction` est équivalent à `void` ; une action ne retourne pas de valeur.

Méthode d'instance / de classe

Une méthode d'instance s'exécute dans le contexte d'une instance de classe, elle accède aux variables propres à cette instance.

Lors de son exécution, une méthode de classe n'est pas attachée à une instance particulière ; elle ne peut accéder aux variables d'instance.

Variables de classe ?

Contrairement à d'autres langages objet, il n'y a pas de variables de classe en Objective-C, seulement des variables d'instance. Nous verrons comment nous en passer dans la suite de l'ouvrage.

Pour comprendre la déclaration des méthodes ayant des paramètres, nous allons détailler un exemple :

```
- (void)getCharacters:(unichar *)buffer
                range:(NSRange)aRange ;
```

Cet exemple déclare une méthode d'instance :

- dont le nom est `getCharacters:range:` ;
- qui ne retourne pas de valeur ; type de retour (`void`) ;
- dont le premier paramètre est de type pointeur sur un unichar – `unichar *` (un `unichar` est un caractère Unicode) ;
- dont le second paramètre est de type `NSRange` (intervalle).

La méthode `–getCharacters:range:` est une méthode d'instance de `NSString` qui permet d'obtenir au format *Unicode* les caractères situés dans un intervalle donné dans la chaîne de caractère.

Une méthode prend des paramètres lorsque son nom comprend des caractères deux-points, autant de paramètres que de caractères deux-points. Ceci est un peu déroutant pour les programmeurs C ou C++ qui ont l'habitude de bien séparer le nom de la fonction de la liste des paramètres qui est mise entre parenthèses. En Objective-C, on mélange. Vous verrez à l'usage que ce procédé améliore la lisibilité du code ; le rôle de chaque paramètre est identifié.

Règle de nommage des méthodes

En Objective-C, la règle de nommage est identique pour les méthodes et les variables : une série de mots accolés et chaque mot commence par une majuscule sauf la première lettre qui reste minuscule, par exemple : `changeValue`.

Accesseurs et Manipulateurs

En respect du principe d'encapsulation, les variables d'instance appartiennent en propre à chaque instance, elles sont accessibles uniquement par une méthode d'instance de la même classe. Comment faire si l'on a besoin de modifier l'état d'un objet ? (voir Figure 2.23)

Le seul moyen pour accéder depuis un objet A à une variable d'instance appartenant à un objet B, est de définir des méthodes d'instance dans la classe de l'objet B pour cet usage. La méthode qui permet d'obtenir la valeur d'une variable d'instance est appelées **accesseur** (*getter*). La méthode qui permet de définir la valeur d'une variable d'instance est appelée **manipulateur** (*setter*).

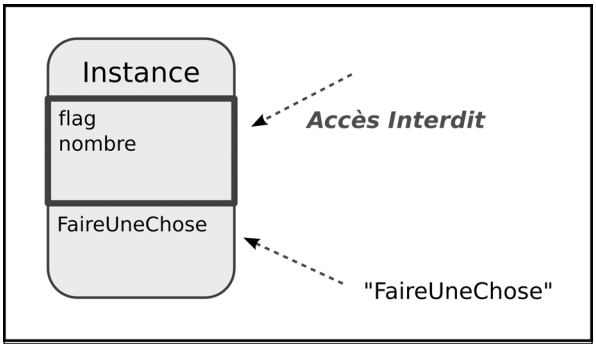


Figure 2.23 : Les variables d'instances sont privées

Accesseur

La méthode permettant d'obtenir la valeur d'une variable d'instance est appelée accesseur. L'accesseur porte le même nom que la variable d'instance, son type de retour est le type de la variable d'instance et il ne prend pas de paramètre.

```
@interface MaClasse : SuperClasse
{
  // déclaration d'une variable d'instance
  typeVar varInstance
}
// déclaration de l'accesseur de varInstance
- (typeVar) varInstance ;

@end
```

Manipulateur

La méthode permettant de définir la valeur d'une variable d'instance est appelée manipulateur. Le nom du manipulateur est construit en mettant une majuscule au nom de la variable puis en le préfixant par `set`. Le manipulateur ne retourne pas de valeur (`void`) et prend un unique paramètre du même type que la variable d'instance.

```
@interface MaClasse : SuperClasse
{
  // déclaration d'une variable d'instance
  typeVar varInstance ;
}
// déclaration du manipulateur de varInstance
- (void) setVarInstance: (typeVar) nouvelleValeur ;

@end
```

Nous verrons que les propriétés permettent de déclarer et définir plus facilement les accesseurs et les manipulateurs.

Définition

La **définition** d'une classe s'effectue dans un fichier source qui porte le nom de la classe et dont l'extension est *.m*, par exemple *MaClasse.m* pour la classe *MaClasse* :

- l'importation du fichier *.h* de la classe (clause `#import`) ;
- une instruction `@implementation` précisant le nom de classe définie ;
- les définitions de chaque méthode de la classe ;
- l'instruction `@end` pour indiquer la fin du bloc `@implementation` :

```
#import "MaClasse.h"
@implementation MaClasse
//    définition des méthodes
@end
```

La définition d'une méthode :

- commence par une ligne de code qui reprend la déclaration de la méthode sans le point-virgule à la fin ;
- se poursuit par un bloc d'instructions entre accolades.

Par exemple :

```
#import "Convertisseur1ViewController.h"
@implementation Convertisseur1ViewController
- (IBAction) changeValue {
   NSString *textDollar = labelDollar.text;
   float dollar = [textDollar floatValue];
   float euro = dollar / 1.4908;
   NSString *textEuro = [[NSString alloc]
                              initWithFormat: @"%.2f",euro];
   labelEuro.text = textEuro;
   [textEuro release];
}
@end
```

Messages

Nous savons maintenant déclarer et définir une classe d'objets ; il s'agit d'un modèle pour créer des instances d'objet. La programmation orientée objet consiste à créer des objets, leur envoyer des messages puis à les libérer lorsque nous n'en avons plus besoin (sur un iPhone, il faut économiser la mémoire). Lorsqu'un objet reçoit un message, il peut à son tour créer des objets, leur envoyer des messages ou les libérer, etc.

La syntaxe pour envoyer un message à un objet est la suivante : `[objet appel-de-méthode]`. `objet` est une variable désignant l'instance (pointeur sur l'instance) si la méthode est une méthode d'instance ou le nom de la classe si c'est une méthode de classe. `appel-de-methode` est le nom de la méthode avec les paramètres à utiliser.

Par exemple, dans l'instruction `dollar = [textDollar floatValue];`, la variable `dollar` prend pour valeur le retour de l'appel de la méthode d'instance `-floatValue` (qui ne prend pas de paramètre) sur l'instance désignée par `textDollar`.

Ou encore, dans l'instruction `[window addSubview:myView];`, la fenêtre désignée par `window` reçoit le message `-addSubview:` avec le paramètre `myView`. Ce message ne retourne pas de valeur.

Propriétés

Une propriété (*property*) est un attribut de l'objet auquel on peut accéder par un accesseur et par un manipulateur. Généralement, cet attribut est concrétisé par une variable d'instance mais ce n'est pas une obligation.

Le langage Objective-C (plus précisément à partir de sa version 2.0) propose des mécanismes pour faciliter l'emploi des propriétés.

@property

La directive `@property` permet de déclarer l'accesseur et le manipulateur d'une propriété. Elle s'utilise à l'endroit où l'on aurait déclaré ces méthodes. Il faut préciser le type et le nom de la propriété.

```
@property UITextField *labelDollar;
```

Dans cet exemple, nous déclarons une propriété `labelDollar` de type `UITextField *` (pointeur sur une instance de la classe *UITextField*). Cette déclaration est totalement équivalente aux deux déclarations :

```
- (UITextField *) labelDollar ;
- (void) setLabelDollar: (UITextField *) newLabelDollar ;
```

@synthesize

La directive `@synthesize` s'utilise dans la définition d'une classe (fichier *.m*) pour générer le code de l'accesseur et du manipulateur d'une propriété.

```
@synthesize labelDollar;
```

Cette instruction génère le code permettant d'accéder à la propriété `labelDollar`.

Notation pointée

La notation pointée permet d'alléger la lecture du code source.

`[objet propriete]` et `[objet setPropriete:newValue]` sont remplacés par `objet.propriété`.

```
NSString *textDollar = labelDollar.text;
labelEuro.text = textEuro;
```

équivaut à

```
NSString *textDollar = [labelDollar text];
[labelEuro setText:textEuro];
```

Création

La création d'une instance s'effectue en deux étapes :

- allocation de la mémoire ;
- initialisation des variables d'instance.

L'allocation de mémoire est effectuée par l'envoi du message `+alloc` (définie dans la classe *NSObject*, mère de toutes les classes) à la classe de l'instance à créer. Ce message renvoie un pointeur sur l'instance qui vient d'être allouée. Par exemple, pour allouer une nouvelle chaîne de caractères :

```
// déclaration d'un pointeur sur UITextField
UITextField *monTextField;
monTextField = [UITextField alloc];
```

Échec lors de l'allocation

Si la création de l'objet échoue (par exemple s'il n'y a plus de mémoire disponible), le message retourne la constante `nil`.

`nil`

`nil` est la constante de type id qui signifie *pointeur nul*. `nil` est équivalent à `NULL` à la seule différence qu'il est de type *pointeur sur un objet* alors que `NULL` est un *pointeur générique*.

On doit toujours initialiser une instance avant de l'utiliser, c'est-à-dire définir les valeurs de ses variables d'instance. Les méthodes d'instance dont le nom commence par `init` servent à initialiser les variables d'instance.

Initialiseur

Un initialiseur est une méthode d'instance destinée à initialiser les variables de l'instance sur laquelle elle est appelée. Une classe peut comporter plusieurs initialiseurs, leur nom commence toujours par `init`. L'initialiseur renvoie un pointeur sur l'instance initialisée.

Chaque classe doit avoir un **initialiseur désigné** unique, généralement celui qui offre le plus de paramètres. Tous les autres initialiseurs appellent l'initialiseur désigné qui est le seul à effectuer réellement le travail.

Tout initialiseur désigné doit commencer par appeler l'initialiseur désigné de sa superclasse.

Retour de `–init`

Si l'initialisation ne se déroule pas correctement, par exemple dans le cas où les paramètres passés ne sont pas corrects, l'initialiseur désigné doit libérer la mémoire allouée et retourner `nil`.

Le message `+alloc` renvoie un pointeur sur une instance, et le message `–init` est susceptible de modifier ce pointeur, c'est pourquoi il est d'usage de grouper l'allocation et l'initialisation dans une seule instruction.

```
UITextField *monTextField;
monTextField = [[UITextField alloc] initWithFrame:rect];
```

On peut même combiner la déclaration, l'allocation et l'initialisation en une seule instruction.

```
UITextField *monTextField = [[UITextField alloc]
                                        initWithFrame:rect];
```

message à `nil`

Que se passe-t-il si l'allocation renvoie `nil` puis que l'on envoie un message d'initialisation au pointeur nul ? Rien. Le langage Objective-C autorise l'envoi de messages à `nil`.

Un initialiseur doit appeler l'initialiseur désigné de sa superclasse. Le schéma habituel pour définir un initialiseur est le suivant :

```
- (id)init {
    if (self = [super init]) {
       // initialisation des variables d'instances
    }
return self;
```

```
}
```

C'est l'occasion de faire la connaissance de deux mots-clés importants en Objective-C : `super` et `self`.

`super`

`super` est un mot-clé Objective-C qui désigne la superclasse de l'instance courante. On l'utilise dans une méthode d'instance pour appeler une méthode de sa superclasse.

Par exemple, le message `[super init]` dans un initialiseur appelle l'initialiseur de la superclasse ; avant d'initialiser les variables d'instance définies dans une classe, on initialise les variables définies dans la superclasse.

`self`

`self` est un mot-clé Objective-C qui désigne l'instance courante. On l'utilise par exemple dans une méthode d'instance pour envoyer un autre message à cette même instance.

Libération

Lorsqu'un objet doit être détruit, il reçoit un message `dealloc`. La méthode d'instance `–dealloc` ne retourne pas de valeur.

Avant d'être détruit, une instance doit penser à détruire, ou du moins libérer (nous verrons bientôt la subtile différence) les instances qu'elle possède. En d'autres termes, les variables d'instances qui occupent de la mémoire doivent être libérées dans la méthode d'instance `–dealloc` ; il s'agit des variables de type pointeur, en particulier les pointeurs sur des objets. Pour libérer un objet, il suffit de lui envoyer le message `release`.

```
[textDollar release];
```

Le langage Objective-C 2.0 sous MacOSX dispose d'un ramasse-miettes (*garbage collector*), comme le langage Java, qui rend inutiles les instructions de libération (*release*). Cette caractéristique n'est actuellement pas disponible sous iPhone OS en raison du manque de mémoire (il faut la libérer dès que possible) et du manque de performance du processeur (le ramasse-miettes est gourmand en processeur).

Les initialiseurs commencent par initialiser les variables d'instances de la superclasse ; à l'inverse, la méthode `–dealloc` doit se terminer

par un appel à la même méthode sur la superclasse. La structure classique d'une méthode –dealloc est la suivante :

```
- (void)dealloc {
    // libération des variables d'instances
[super dealloc];
}
```

Release et pas dealloc

Pour libérer une instance, on lui transmet le message `release`. On ne doit jamais transmettre directement un message `dealloc` à un objet.

Le message `release` informe l'instance que l'un des objets qui l'utilisent n'en a plus besoin. Le message `dealloc` sera transmis automatiquement à cette instance lorsqu'elle ne sera plus employée par aucun objet.

2.5. Check-list

Ce chapitre a commencé par une introduction à la *Programmation Orientée Objet*, ce qui nous a permis de découvrir :

- les objets, classes et instances ;
- l'état des objets, leur comportement et la transmission de messages ;
- le principe d'encapsulation ;
- l'héritage.

Nous avons réalisé notre première application interactive *Convertisseur1* pour notre prochain voyage aux États-Unis. Nous avons fait connaissance avec :

- les outlets, les actions et le mécanisme cible-action ;
- le champ de texte `UITextField` ;
- l'arbre d'héritage de `UITextField` ;
- la hiérarchie des vues ;
- la saisie de code source sous *XCode* et la terminaison de code.

Nous avons terminé par un approfondissement de la syntaxe du langage *Objective-C* :

- clause `#import` ;
- déclaration d'une classe :
  - `@interface` ;
  - déclaration des variables d'instance ;
  - `@property` pour déclarer les propriétés ;

  - déclaration des méthodes de classe et d'instance.

- définition d'une classe :
  - `@implementation` ;
  - `@synthesize` pour générer *l'accesseur* et le *manipulateur* d'une propriété.
- envoi de message ;
- mots-clés `super` et `self` et `nil` ;
- méthodes impliquées dans le cycle de vie des objets :
  - `+alloc` ;
  - initialiseurs `–init` et *initialiseur désigné* ;
  - `–release` ;
  - `–dealloc`.

L'application *Convertisseur1* est bourrée de défauts. Nous allons corriger cela dès le prochain chapitre. Ce sera l'occasion de découvrir de nouveaux mécanismes de *Cocoa Touch*.

GESTION DE LA MÉMOIRE

Alors qu'un ordinateur dispose fréquemment de 1 Go de RAM ou plus, la mémoire est limitée à 128 Mo sur un iPhone. Cet espace est partagé entre le système iPhone OS, l'affichage graphique et l'application en cours d'exécution ; cette dernière ne dispose que d'environ 64 Mo. La mémoire est donc une ressource précieuse qu'il faudra économiser.

Nous allons comprendre dans ce chapitre comment est gérée la mémoire sous Cocoa Touch et Objective-C. Nous mettrons en œuvre les **Instruments** du SDK pour chasser les erreurs courantes relatives à la gestion de la mémoire et nous améliorerons le comportement de notre application *Convertisseur1* vis-à-vis de la mémoire.

3.1. Diagnostiquer les fuites mémoire avec Leaks

Zombi

Vous connaissez certainement les morts-vivants de cinéma et autres zombis de même nature. Un **zombi** Objective-C est aussi un objet mort-vivant.

À la création d'une instance, il faut conserver son adresse dans une variable de type pointeur sur un objet. Cette variable est appelée **référence** sur l'instance. Pour pouvoir émettre un message vers un objet, vous devez disposer d'une référence sur cet objet. Dans l'exemple, les messages sont transmis à l'instance référencée par la variable `textEuro`.

```
// création d'une instance
NSString *textEuro = [NSString alloc];
// émission d'un message
textEuro = [textEuro initWithFormat: @"%.2f",euro];
// émission d'un autre message
valeur = [texteuro floatValue];
```

Si nous perdons la référence à un objet, nous ne pouvons plus lui envoyer de message. Dans le deuxième exemple, nous créons une instance (appelons-la *Objet1*) référencée par le pointeur `textEuro` puis une seconde instance (*Objet2*) référencée par le même pointeur ; la valeur précédente de `textEuro` est alors perdue.

```
// création d'une instance Objet1
NSString *textEuro = [NSString alloc];
// émission d'un message vers Objet1 référencé par textEuro
textEuro = [textEuro initWithFormat: @"%.2f",euro];
// création d'une instance Objet2
```

```
textEuro = [NSString alloc];
// émission d'un message vers Objet2,
//       Objet1 n'est plus accessible
textEuro = [textEuro initWithFormat: @"%.2f",euro];
```

Nous n'avons plus de référence vers *Objet1*. Cette instance n'est pas détruite (elle est toujours vivante) mais n'est plus accessible (comme si elle était morte) ; *Objet1* est un zombi.

Zombi

Un zombi est un objet qui n'est pas accessible car il n'est référencé par aucun pointeur dans l'application. Cet objet est inutilisable ; il occupe donc inutilement de la mémoire.

Si vous avez déjà vu des zombis au cinéma, vous savez qu'ils ne sont pas très sympathiques. Et les zombis Objective-C, gentils ou méchants ? Vous l'avez déjà deviné ; ils ne sont pas très sympathiques non plus, pas vraiment dangereux, mais nuisibles car ils occupent un espace mémoire qui est irrémédiablement perdu ; cet espace sera récupéré par le système uniquement lorsque l'utilisateur quittera l'application.

Détecter les fuites mémoire

Définition d'une fuite mémoire

Un zombi apparaît à cause d'une erreur de programmation (perte de la dernière référence d'un objet non libéré). Lorsque ces erreurs sont trop nombreuses ou lorsqu'elles sont rencontrées plusieurs fois au cours de l'exécution, le nombre de zombis s'accroît au fur et à mesure ; c'est ce que l'on nomme une **fuite mémoire**. Cette erreur très courante peut dégrader les performances de l'application, voire provoquer un plantage.

Fuite mémoire

Erreur courante en programmation orientée objet. Une fuite mémoire est l'accroissement progressif, tout au long de l'exécution d'une application, de la mémoire allouée inutilement. Ce phénomène peut aboutir à une diminution des performances ou un plantage de l'application.

Nous allons provoquer une fuite mémoire dans notre application *Convertisseur1*. Modifiez le code source de *Convertisseur1ViewController.m* ; mettez en commentaire l'instruction qui libère `texteEuro`.

```
- (IBAction) changeValue {
    NSString *textDollar = labelDollar.text;
    float dollar = [textDollar floatValue];
    float euro = dollar / 1.4908;
    NSString *textEuro = [[NSString alloc]
                              initWithFormat: @"%.2f",euro];
    labelEuro.text = textEuro;
//    [textEuro release];
}
```

À chaque exécution de la méthode `–changeValue`, c'est-à-dire à chaque modification du champ de texte contenant la valeur en dollars, une nouvelle instance de `NSString` est référencée par `textEuro` ; l'instance précédente devient un **zombi**.

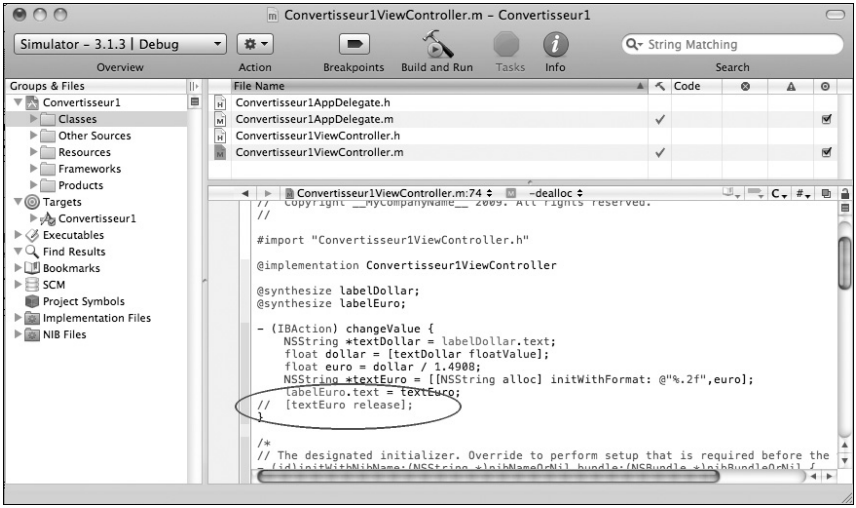


Figure 3.1 : Création d'une fuite mémoire

Instruments Leaks

Il est indispensable **d'instrumenter** l'application en cours d'exécution pour détecter les fuites mémoire (*leaks*). Nous allons mettre en œuvre les instruments fournis avec le SDK d'Apple sur notre application *Convertisseur1* modifiée.

1. Reconstruisez l'application mais ne lancez pas son exécution immédiatement ; commande **Build** du menu **Build** sous XCode ou ⌘+B.
2. Lancez l'exécution sous instrumentation des fuites mémoire. Pour cela, sélectionnez la commande **Leaks** du sous-menu **Start with Performance Tools** du menu **Run** sous XCode.

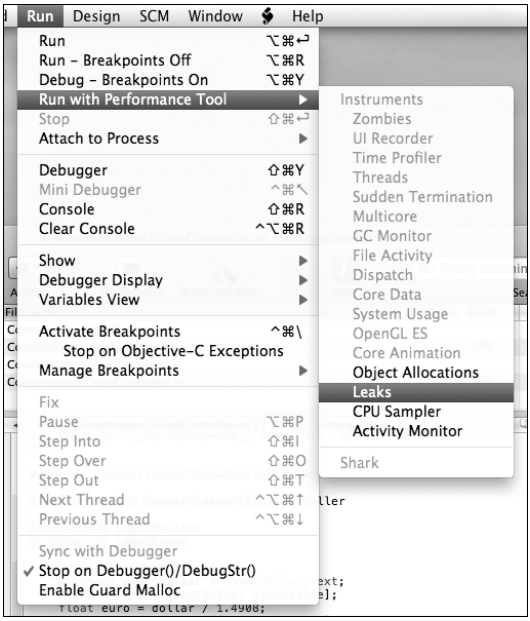


Figure 3.2 : Lancement du détecteur de fuites mémoire

L'application *Convertisseur1* se lance dans le simulateur iPhone en même temps que l'application *Instruments* ; l'enregistrement des événements mémoire débute automatiquement.

3 Manipulez l'application avec le simulateur ; saisissez des chiffres dans le champ de texte contenant le montant en dollars, effacez des caractères puis recommencez pendant une trentaine de secondes.

4 Quittez l'application *Convertisseur1* en pressant le bouton **Home** du simulateur ou à l'aide de la sélection de touches ⌥+⌘+H. L'enregistrement des données sous **Instruments** s'arrête automatiquement.

Dans la fenêtre principale de l'application Instruments, réalisez les opérations suivantes :

1 Sélectionnez l'outil **Leaks** dans la liste des instruments sur la partie gauche, en haut de la fenêtre.

2 Cliquez sur l'icône **Extended Detailed View** (*Vue détaillée étendue*) sur la barre d'état en bas de la fenêtre.

3 Dans la partie centrale de la fenêtre, chaque ligne est un zombi. Sélectionnez une de ces lignes ; **l'état de la pile** (*stack trace*) au

moment de la dernière opération mémoire sur cet objet apparaît dans la partie droite de la fenêtre (*vue détaillée étendue*).

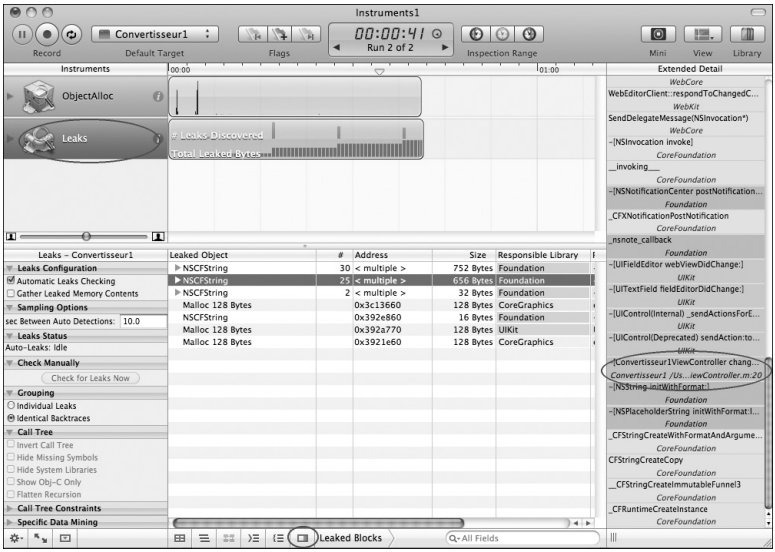


Figure 3.3 : Instrument détecteur de zombis (leaks)

Il n'est pas inhabituel que la pile contienne une cinquantaine d'éléments. Chaque élément est décrit sur deux lignes :

- le nom de la fonction appelée ;
- le nom du framework ou du projet XCode et du fichier source auquel appartient cette fonction.

DEFINITION

Pile

La Pile (*Stack*) est l'espace mémoire utilisé par le processeur pour conserver le contexte (paramètres et variables locales) d'une fonction ou d'une méthode.

L'état de la pile permet de connaître l'enchaînement des appels de fonction à un instant donné ainsi que les valeurs des paramètres et des variables locales de chaque fonction lors de l'appel à la suivante.

La pile informatique fonctionne comme une pile d'assiettes. Lors de l'appel à une fonction, une assiette est ajoutée en haut de la pile pour contenir le contexte de la fonction appelée. Lorsqu'une fonction se termine (instruction `return`), l'assiette en haut de la pile est enlevée ; le contexte de la fonction qui se termine est détruit et l'on revient au contexte de la fonction appelante.

S'il y a des erreurs de programmation, elles sont probablement davantage dans le code que nous avons écrit que dans les frameworks du SDK.

Recherchons dans la vue détaillée étendue (partie droite de la fenêtre d'Instruments) les fonctions appartenant au projet *Convertisseur1*. Nous en trouvons trois :

- Tout en haut de la liste ; les fonctions `start` et `main`. Ce n'est pas étonnant, toutes les applications commencent par l'exécution de ces deux fonctions.
- Vers le bas de la liste, nous trouvons le message `-[Convertisseur1 ViewController changeValue]`.

L'application *Instruments* détecte les fuites mémoire que nous avons créées.

Diagnostiquer les fuites mémoire

Une fois identifiées la fonction ou la méthode en cause, il faut déterminer quel est précisément l'objet qui se transforme en zombi.

1 Double-cliquez sur le message `-[Convertisseur1ViewController changeValue]` dans la vue détaillée étendue sous *Instruments*. Ce double-clic vous renvoie vers le code source de la méthode en cause, sous XCode ; une ligne de code est surlignée.

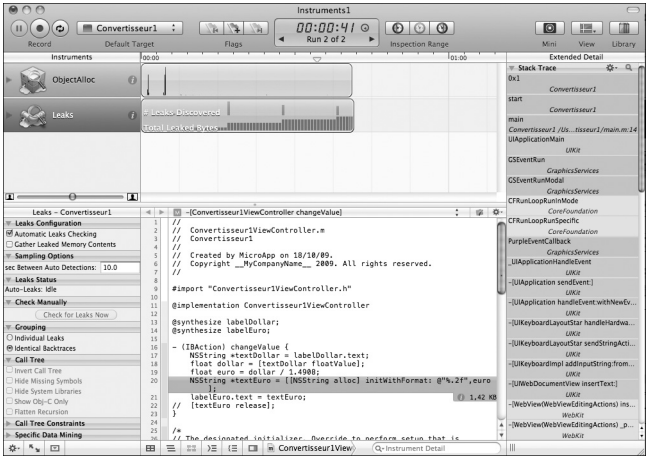


Figure 3.4 : Identification du zombi

L'instruction de notre application qui provoque une fuite mémoire est celle qui crée une instance de `NSString` référencée par la variable `textEuro`. C'est cette instance qu'il faut libérer pour éviter les zombis.

2 Enlevez la mise en commentaire de l'instruction `[textEuro release];`, reconstruisez l'application et testez-la à nouveau en utilisant l'instrument **Leaks**. Il y a beaucoup moins de fuites mémoire.

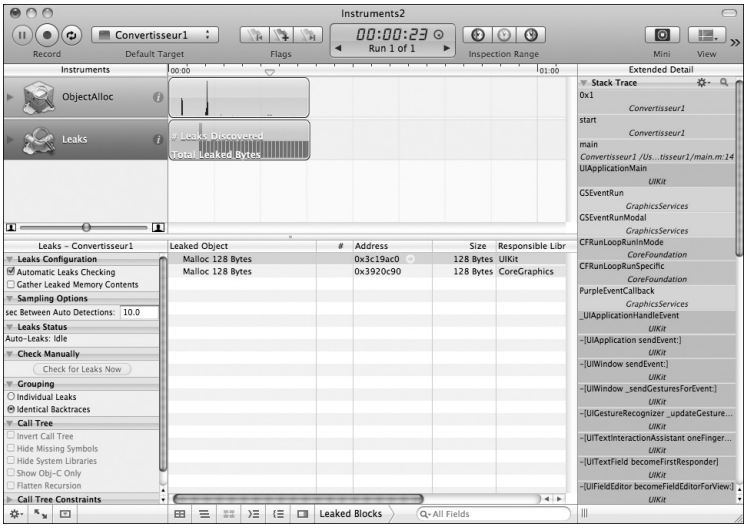


Figure 3.5 : Vérification de la diminution des fuites mémoires

Selon la version du SDK que vous utilisez, il est même possible qu'il n'y ait plus du tout de fuites mémoire. Il est aussi possible qu'il reste quelques lignes, vérifiez que le code que vous avez écrit n'est pas impliqué dans ces lignes résiduelles.

Nous venons d'illustrer la recherche des fuites mémoire en utilisant une application dans laquelle nous avons introduit une erreur de programmation. Bien sûr, la recherche de ce type d'erreur est un peu moins facile dans la réalité mais le principe est toujours le même : mettre en œuvre l'instrument **Leaks**.

La première chose à faire est tout de même d'éviter d'introduire ce genre d'erreur dans notre code source. C'est pourquoi nous allons étudier les règles qui régissent la gestion de la mémoire.

Chassez les fuites mémoire

Malgré tout le soin que nous mettrons à respecter les règles de gestion de la mémoire, il est indispensable de vérifier que notre programmation est correcte en testant notre application à l'aide de l'instrument **Leaks**.

3.2. Éviter les fuites mémoire

Compteur de références

Chaque instance dispose d'un compteur de références qui lui est propre, quelle que soit sa classe d'appartenance. Lorsqu'une instance est créée, son **compteur de références** (*retain count*) prend la valeur 1.

Lorsqu'une instance reçoit le message `retain`, son compteur de références est incrémenté, tandis que lorsqu'elle reçoit le message `release`, il est décrémenté. Le message `retainCount` permet de connaître la valeur du compteur de références.

Lorsque son compteur de références atteint la valeur nulle, le message `dealloc` est transmis à l'instance :

```
// création le compteur vaut 1
MaClasse *monInstance = [[MaClasse alloc] init];
// le compteur est incrémenté, il vaut 2
[monInstance retain];
// compteur vaut 2
int compteur = [monInstance retainCount];
// le compteur est décrémenté, il vaut 1
[monInstance release];
// le compteur est décrémenté à 0, l'instance est détruite
[monInstance release];
```

Compteur de références

Chaque objet dispose d'un compteur de références (`retain count`) qui vaut `1` à la création de l'instance. Ce compteur est incrémenté par le message `retain`, il est décrémenté par le message `release`. L'objet est détruit lorsque son compteur de référence est égal à `0`.

Rappelez-vous, pour chacune des classes que vous écrivez, il est important de définir une méthode `–dealloc` dans laquelle vous transmettez un message `release` à chaque instance retenue par la l'objet de la classe (en particulier les propriétés). Il faut invoquer `dealloc` sur la superclasse.

Gestion des propriétés

Logiquement, s'il faut libérer les propriétés d'une instance lorsqu'elle est détruite, c'est qu'elles ont été retenues auparavant.

Attributs des propriétés

Souvenez-vous du fichier *Convertisseur1ViewController.h* :

```
#import <UIKit/UIKit.h>
@interface Convertisseur1ViewController : UIViewController {
   IBOutlet UITextField *labelDollar;
   IBOutlet UITextField *labelEuro;
}
@property (retain,nonatomic) UITextField *labelDollar;
@property (retain,nonatomic) UITextField *labelEuro;
- (IBAction) changeValue;
@end
```

Nous avons ajouté deux attributs à la clause `@property` : `retain` et `nonatomic`. L'attribut `retain` indique que le manipulateur de la propriété doit la retenir et qu'il doit libérer la valeur précédente. Le principe du code du manipulateur généré par la clause `@synthesize` est alors le suivant :

```
- (void) setLabelDollar: (UITextField *) textField {
   [textField retain];
   [labelDollar release];
   labelDollar = textField;
}
```

Le tableau résume les attributs utilisables avec la clause `@property`.

Tableau 3.1 : Principaux attributs de la clause @property

Thème	Attribut	Rôle
Nom des méthodes générées	`getter=` nom-de-l-accesseur	Permet de définir un nom pour l'accesseur autre que celui par défaut.
	`setter=` nom-du-manipulateur	Permet de définir un nom pour le manipulateur autre que celui par défaut.
Propriété modifiable	`readwrite` (attribut par défaut)	`@synthesize` générera l'accesseur et le manipulateur.
	`readonly`	`@synthesize` générera seulement l'accesseur.
Gestion de la mémoire Le compilateur émet un avertissement si l'un de ces attributs n'est pas utilisé	`assign`	Dans le manipulateur généré par `@synthesize`, la nouvelle valeur remplace simplement l'ancienne valeur.
	`retain`	Dans le manipulateur généré par `@synthesize`, la nouvelle valeur est retenue et l'ancienne valeur est libérée.
	`copy`	Dans le manipulateur généré par `@synthesize`, la nouvelle valeur est dupliquée et l'ancienne valeur est libérée.

Tableau 3.1 : Principaux attributs de la clause @property

Thème	Attribut	Rôle
Atomicité	`nonatomic`	Permet d'améliorer les performances dans une application n'utilisant pas le parallélisme d'exécution (*multi-threading*).

Recommandations d'emploi

nonatomic

Sur iPhone OS, il est recommandé d'employer l'attribut `nonatomic`, sauf dans les cas rares où les instances de la classe sont susceptibles d'être utilisées dans un contexte d'exécution concourante. Nous n'étudierons pas ces situations dans le cadre de cet ouvrage.

retain

L'attribut `retain` est recommandé pour les propriétés qui sont des instances de classe Objective-C. Il n'est pas utilisable pour les propriétés de type scalaire.

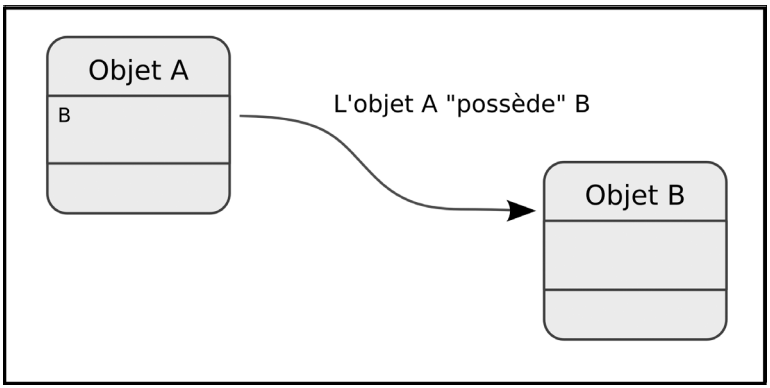


Figure 3.6 : Relation d'appartenance

Dans cet exemple, l'objet `A` possède l'objet `B`. La propriété `B` doit être retenue (déclarée avec l'attribut `retain`). Les propriétés retenues doivent être libérées dans la méthode `–dealloc`.

Un objet `B` peut appartenir simultanément à plusieurs autres objets. Si chacun des propriétaires retient cet objet `B`, son compteur de références sera égal au nombre de propriétaires. Cela garantit que l'objet `B` sera détruit seulement lorsqu'il n'aura plus de propriétaires.

assign

L'attribut `assign` est employé avec des propriétés de type scalaire (par opposition à Objet), c'est-à-dire les types fondamentaux du langage C (`int`, `float`, `double`, `char`, `struct`, etc.) ou leur équivalent Cocoa Touch (`NSInteger`, `CGRect`, etc.).

On l'utilise également avec des propriétés de type *Objet* lorsque l'on souhaite qu'un objet connaisse un autre objet sans pour autant le posséder. Dans l'exemple, l'objet `A` possède l'objet `B`, et donc la propriété `B` de l'objet `A` est déclarée avec l'attribut `retain`. Il est alors incorrect que l'objet `B` possède l'objet `A`. Par analogie, vous possédez votre iPhone mais lui ne vous possède pas. Il vous connaît néanmoins car vous avez saisi votre nom lorsque vous l'avez initialisé ; la propriété `A` de l'objet `B` est déclarée avec l'attribut `assign`.

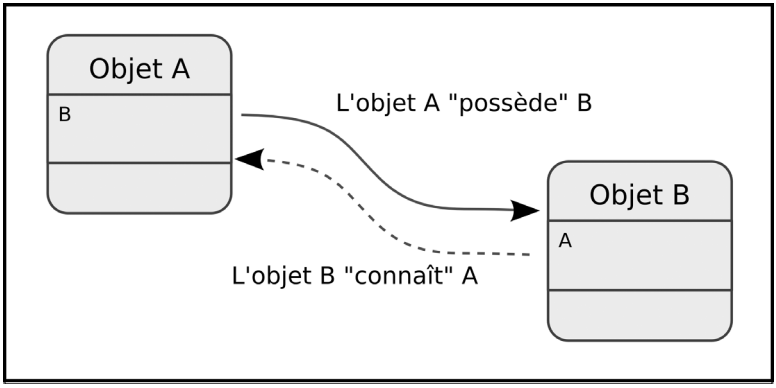


Figure 3.7 : Connaissance (Assign) n'est pas possession (retain)

copy

L'attribut `copy` indique que le paramètre passé au manipulateur doit être dupliqué avant d'être affecté à la propriété.

```
-(void)setString:(NSString *)newString {
   if (string != newString) {
      [string release];
      string = [newString copy];
   }
}
```

La méthode `–copy` est déclarée dans `NSObject`. Elle doit être définie dans toutes les classes où l'on souhaite en disposer. Elle duplique l'instance (et toutes ses propriétés) et renvoie un pointeur sur le duplicata.

Responsabilités des objets

Règle de gestion de la mémoire

La règle qui permet d'éviter les zombis est simple.

Règle de gestion de la mémoire
Une classe d'objet qui obtient une instance par `+alloc`, par `–copy` (ou ses dérivés) ou qui retient une instance (`retain`) est responsable de sa libération. Les instances obtenues par un autre moyen ne doivent pas être libérées.

Dire qu'une classe est responsable de la libération des instances obtenues, cela signifie que le programmeur doit veiller à envoyer le message `release` aux variables référençant un objet avant de les modifier ; c'est ce que nous avons fait dans la méthode `–changeValue` de la classe `Convertisseur1ViewController`.

Quels sont les autres moyens d'obtenir une instance ? Il s'agit de façon générale de la valeur de retour d'une méthode autre que `+alloc` ou `–copy`. Vous voyez un exemple ? Un accesseur bien sûr, renvoie une référence sur un objet sans l'intention d'en transférer la propriété ; il appartient à celui qui le reçoit de le retenir s'il le souhaite.

Cycle de vie des objets

La durée de vie des objets pendant l'exécution d'une application est très variable. Certains objets sont créés au lancement de l'application et vont perdurer jusqu'à sa terminaison, par exemple les champs de texte de notre Application *Convertisseur1*. D'autres ont une vie extrêmement brève : l'instance `textEuro` de la méthode `–changeValue`.

Dans le contexte de l'iPhone OS où la mémoire est une denrée précieuse, il est recommandé de détruire les objets dès que possible. Mais que se passe-t-il dans le cas où nous voulons écrire une méthode qui crée un objet pour le retourner. Par exemple, nous voulons ajouter à la classe Convertisseur1ViewController une méthode qui retourne une chaîne de caractères contenant la valeur convertie en euros.

```
- (NSString *) euroAsString {
   NSString *textDollar = labelDollar.text;
   float dollar = [textDollar floatValue];
   float euro = dollar / 1.4908;
   NSString *textEuro = [[NSString alloc]
                              initWithFormat: @"%.2f",euro];
```

```
    return textEuro;
}
```

Le problème ici est que nous violons la règle de gestion de la mémoire : la méthode obtient une instance par `+alloc` mais ne se préoccupe pas de sa libération ; si la méthode libérait cette instance, elle ne pourrait pas la retourner à l'appelant.

Heureusement, il existe une autre façon de libérer les objets qui va nous permettre de continuer à respecter la règle.

Pool d'autolibération

Il faut ajouter une toute petite instruction importante, avant de retourner l'instance nouvellement créée :

```
- (NSString *) euroAsString {
    NSString *textDollar = labelDollar.text;
    float dollar = [textDollar floatValue];
    float euro = dollar / 1.4908;
    NSString *textEuro = [[NSString alloc]
                              initWithFormat: @"%.2f",euro];
    [textEuro autorelease];
    return textEuro;
}
```

Le message `autorelease` programme la libération pour plus tard. Ainsi la méthode `–euroAsString` s'occupe de libérer l'instance créée ; elle respecte donc la règle mais le fait de façon à se laisser le temps de renvoyer la valeur attendue.

Et quand la libération sera-t-elle opérée ? En fait, les objets auxquels ont transmet le message `autorelease` sont ajouté au pool d'autolibération (*autorelease pool*) courant. Lorsque le pool est détruit, tous les objets qui y sont rattachés sont libérés.

Sous Cocoa Touch, un pool d'autolibération est créé au début de chaque boucle d'événement et détruit à la fin de la boucle.

Boucle d'événement

Une application sur iPhone passe son temps à attendre des événements et à y répondre.

Lorsque l'utilisateur saisit des caractères sur le clavier ou lorsqu'il touche un bouton, lorsque un appel est reçu, que l'appareil est secoué, etc. Tous ces événements sont représentés par des instances de la classe `UIEvent` par le système iPhone OS puis transmis à l'application.

Lorsque l'application reçoit un événement, elle débute une boucle d'événement :

- création d'un pool d'autolibération ;
- détermination du contrôle (instance de la classe `UIControl`) dans l'application qui peut traiter l'événement (par exemple un champ de texte dans le cas de saisie de caractères) ;
- traitement par le contrôle d'une partie de l'événement (affichage du caractère) et génération de l'action éventuellement attachée suivie d'une transmission à la cible ;
- traitement par la cible de l'action (`-changeValue` dans notre application) ;
- fin de traitement de l'événement par l'application (mise à jour de l'affichage des autres vues) ;
- destruction du pool d'autolibération ;
- attente de l'événement suivant.

3.3. Améliorer Convertisseur1

Nous allons vérifier que notre application respecte la règle de gestion de la mémoire.

La seule classe que nous avons modifiée est `Convertisseur1ViewController` ; nous allons donc y concentrer nos efforts.

Instances manipulées

Commençons par identifier la liste des instances obtenues. Nous allons dresser un tableau pour préciser comment nous obtenons chaque instance et comment nous la libérons.

Tableau 3.2 : Liste des instances manipulées dans Convertisseur1ViewControlleur

Portée	Instance	Obtention	Libération
Propriété	`labelDollar`	`-retain` dans le manipulateur	
	`labelEuro`	`-retain` dans le manipulateur	
`-changeValue`	`textDollar`	accesseur de `text` sur `labelDollar`	Inutile car obtenu autrement que par `+alloc`, `-retain` ou `-copy`
	`textEuro`	`+alloc`	`-release`

Nous nous apercevons que nous ne gérons pas correctement les propriétés. Il faut y remédier.

Mise en conformité avec la règle

Méthode -dealloc

Nous avons oublié de libérer les propriétés dans la méthode `–dealloc` de la classe.

Sélectionnez le fichier *Convertisseur1ViewController.m* sous XCode. La méthode `–dealloc` a déjà été préparée par XCode ; modifiez cette méthode pour libérer les propriétés.

```
- (void)dealloc {
    self.labelDollar = nil;
    self.labelEuro = nil;
    [super dealloc];
}
```

Libération des propriétés

Notez la façon dont nous libérons les propriétés, au lieu d'écrire `[labelDollar release];`, nous utilisons la notation pointée qui est équivalente à `[self setLabelDollar:nil];`. La propriété étant déclarée avec l'attribut `retain`, le manipulateur commence par libérer la propriété `Actuelle` ; c'est ce que nous souhaitons.

L'intérêt d'employer `self.labelDollar = nil` est double :

1. la propriété vaut `nil` au lieu de contenir une référence obsolète, ce qui va éviter beaucoup de plantage ;
2. cette instruction fonctionne aussi bien quel que soit l'attribut (`assign`, `retain`, `copy`) de la propriété, ce qui n'est pas le cas du message `release` qui est une erreur de programmation si l'attribut est `assign`.

Ainsi le code de notre méthode `–dealloc` est insensible aux modifications d'attribut que nous pourrions faire par la suite. Notre code est plus robuste ; il est plus facile d'en assurer la maintenance.

Méthode -viewDidUnload

XCode a préparé d'autres méthodes dans la classe `Convertisseur1ViewController`.

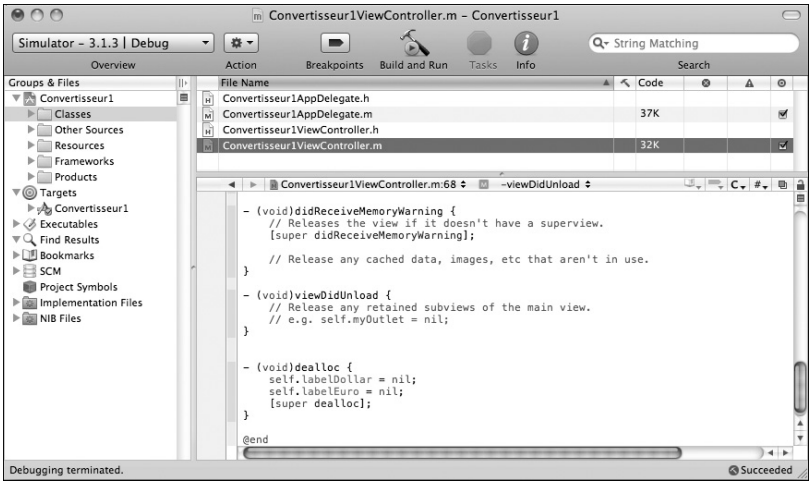


Figure 3.8 : XCode a préparé des méthodes

Nous pouvons voir juste au-dessus de la méthode `–dealloc` une méthode `–viewDidUnload` dans laquelle nous pouvons lire un commentaire nous incitant à y libérer les outlets ; c'est exactement ce que nous venons de faire dans la méthode `–dealloc`.

La méthode `–viewDidUnLoad` est appelée lorsque la Vue contrôlée par l'instance du contrôleur de Vue vient d'être libérée. Cela se produit quand la mémoire sature et que la Vue n'est pas affichée à l'écran ; elle occupe inutilement de l'espace mémoire et pourra être rechargée à partir du fichier NIB si l'utilisateur y revient. Cette caractéristique est utile uniquement dans les applications multivues, où l'utilisateur peut passer d'une vue à l'autre ; les vues qui sortent de l'affichage à un moment donné restent dans la mémoire afin de pouvoir être réaffichées plus rapidement.

Notre application *Convertisseur1* n'est pas multivue, pour l'instant, mais prenons tout de suite de bonnes habitudes. Modifiez le code de la méthode `-viewDidUnload` :

```
- (void)viewDidUnload {
   // Release any retained subviews of the main view.
   // e.g. self.myOutlet = nil;
   self.labelDollar = nil;
   self.labelEuro = nil;
}
```

Factorisation du code

Les méthodes `–dealloc` et `–viewDidUnload` se ressemblent beaucoup.

C'est une bonne pratique de la programmation que d'éviter de telles ressemblances ; elles rendent le code plus difficile à maintenir. L'élimination de ces ressemblances s'appelle la **factorisation du code**.

Factorisation du code
La factorisation de code est l'élimination des séquences de code qui se ressemblent. De telles ressemblances rendent le code difficile à maintenir et sont sources d'erreurs.

Dans le cas présent, le code est facile à factoriser. Le rôle de `-viewDidUnload` est de libérer les outlets, et nous voulons que les outlets soient libérés lors de l'appel de `-dealloc` : il suffit d'appeler `-viewDidUnload` depuis `-dealloc`. Modifiez le code de la classe `Convertisseur1ViewController`.

```
- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    self.labelDollar = nil;
    self.labelEuro = nil;
}

- (void)dealloc {
    [self viewDidUnload];  // libère les outlets
    [super dealloc];
}
```

Nous avons fait un excellent travail. Reconstruisez l'application et testez-la. Jetez un coup d'œil avec l'instrument **Leaks** pour vérifier que tout fonctionne, sans fuite mémoire provoquée par votre code.

Résumé

Résumons les recommandations importantes qui vous serviront pour tous les développements :

- Souvenez-vous de la règle de gestion de la mémoire : une classe propriétaire d'un objet est responsable de sa libération.
- Élaborez un tableau des instances manipulées pour chaque classe que vous développez, cela vous aidera à vérifier que la règle est respectée.
- Factorisez votre code, il y aura moins d'erreurs.
- Utilisez l'instrument Leaks pour vérifier qu'il n'y a pas de fuites mémoire.

Références obsolètes

Ce chapitre a traité jusqu'ici des zombis, des fuites mémoire, donc des risques liés au manque de libération. Nous ne pouvons le clore sans dire ce qui se passe si on libère trop. Que se passe-t-il si une instance reçoit trop de message `release` ? Autrement dit, que se passe-t-il si l'on envoie un message à un objet qui a été détruit (on parle de référence obsolète) ? Essayons.

Référence obsolète

Une **référence obsolète** est une variable qui pointe sur une instance qui n'existe plus.

Provoquer une référence obsolète

Modifiez la méthode `–changeValue` ; libérez l'instance `textEuro` avant de l'affecter au champ de texte contenant le montant en euros.

```
- (IBAction) changeValue {
   NSString *textDollar = labelDollar.text;
   float dollar = [textDollar floatValue];
   float euro = dollar / 1.4908;
   NSString *textEuro = [[NSString alloc]
                         initWithFormat: @"%.2f",euro];
   [textEuro release];
   labelEuro.text = textEuro;
//   [textEuro release];
}
```

Construisez l'application et testez-la. Inutile ici d'employer les instruments.

L'application plante et on revient à XCode dès que l'on essaie de saisir un caractère dans le champ de texte contenant le montant en dollars. Ce plantage peut se manifester de différentes façons : l'affichage peut simplement se figer ou un message d'erreur peut être émis ou l'application quitte brutalement sans message, ou encore il ne se passe plus rien (voir Figure 3.9).

Si un message est affiché dans une boîte de dialogue, vous pouvez cliquer sur le bouton **Rapport**... pour avoir plus de détails sur l'erreur (inutile d'envoyer le rapport à Apple) (voir Figure 3.10).

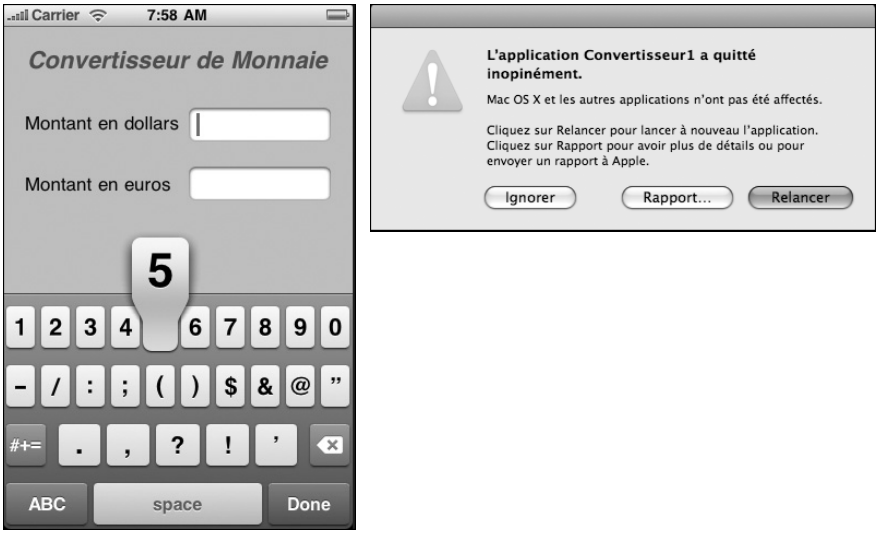


Figure 3.9 : Différentes façons de planter

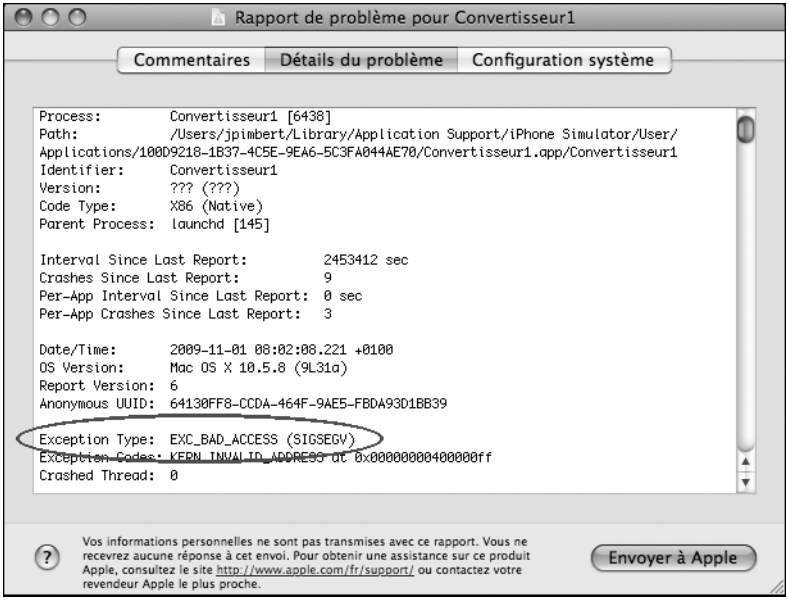


Figure 3.10 : Rapport détaillé d'une référence obsolète

Ce rapport indique qu'un accès mémoire illégal a été effectué : `EXC_BAD_ACCESS`.

Diagnostiquer l'erreur

Dans tous les cas, que le message s'affiche ou pas, nous pouvons diagnostiquer l'erreur en utilisant le **Debogueur** de XCode (*debugger*).

Sélectionnez la commande **Debug - Breakpoints On** du menu **Run** de XCode ou utilisez la combinaison de touches ⌥+⌘+Y.

Testez votre application jusqu'au plantage. Si la fenêtre du débogueur ne s'affiche pas sous XCode, sélectionnez la commande **Debugger** du menu **Run** (⌥+⌘+Y). Une fois affichée, elle nous permet de visualiser l'état détaillé du programme au moment du plantage. Sélectionnez la ligne numéro 2 dans la partie gauche de la fenêtre pour voir ce qui s'est passé dans notre méthode `-changeValue`.

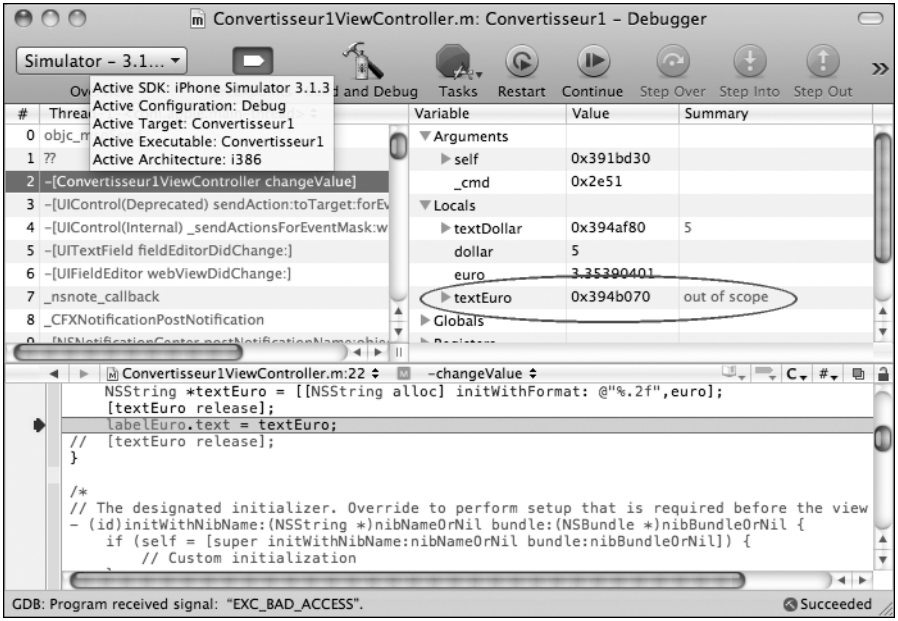


Figure 3.11 : État du programme lors du plantage

Le programme s'est arrêté alors qu'il essayait d'affecter l'instance `textEuro` au champ de texte devant afficher la valeur en euros (partie basse de la fenêtre). Nous voyons aussi et surtout, sur la partie droite de la fenêtre, que la variable `textEuro` présente une valeur **invalide** (*out of scope*). Cette variable pointe sur une instance qui n'existe pas puisqu'elle vient d'être détruite dans la ligne de code précédente.

 Si la barre d'état ne contient pas de message d'erreur, cliquez sur le bouton **Continue** de la barre d'outils. Vous pouvez alors lire le message "*GDB: Program received signal: EXC_BAD_ACCESS*" dans la ligne d'état en bas de la fenêtre.

Nous venons de montrer que l'utilisation d'une référence obsolète provoquait généralement le plantage de l'application associé à l'émission d'un signal `EXC_BAD_ACCESS`. Nous avons vu également comment utiliser le débogueur pour localiser l'instruction ayant provoqué ce plantage et visualiser l'état des variables à ce moment-là.

La fenêtre du débogueur

Nous allons terminer par quelques explications concernant la fenêtre du débogueur qui est composée de quatre parties :

- une barre d'outils en haut, que nous apprendrons bientôt à utiliser ;
- l'état de la pile dans la partie gauche : les fonctions de notre application y apparaissent en couleur foncée, signe que nous pouvons en visualiser le code source ;
- le code source dans la partie basse ; lorsque nous sélectionnons une fonction de notre application dans l'état de la pile, ou le code machine lorsque nous sélectionnons une fonction de l'un des frameworks de Cocoa Touch ;
- l'état des variables de la fonction sélectionnée dans la partie droite. Là aussi, cet état est facile à interpréter uniquement si nous sélectionnons une fonction écrite dans notre application ;
- une barre d'état en bas de la fenêtre.

Le code source et l'état des variables sont lisibles dans le débogueur uniquement pour les fonctions qui sont compilées en mode **Debug**. Ce mode de compilation indique que le code exécutable doit être enrichi pour permettre au débogueur d'afficher ces informations. Lorsque vous livrerez votre application sur l'AppStore, vous pourrez les supprimer et compiler pour cela en mode Release.

Pour changer le mode de compilation, utilisez le sous-menu **Set Active Build Configuration** du menu **Project** sous XCode (voir Figure 3.12).

Vous pouvez aussi utiliser le menu **Overview** de la barre d'outils de XCode.

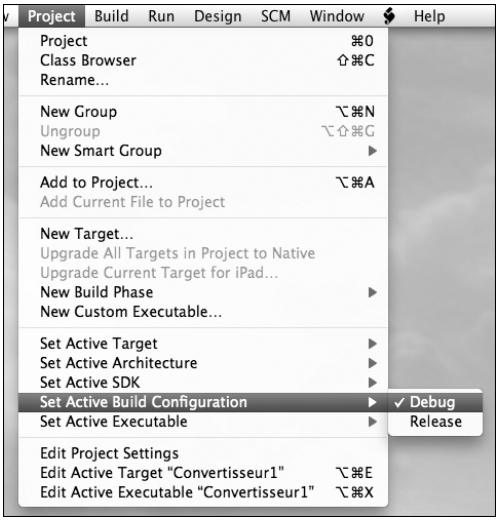


Figure 3.12 : Changement du mode de compilation sous XCode

3.4. Check-list

Nous avons vu dans ce chapitre comment gérer la mémoire, ressource précieuse dans le contexte de l'iPhone : la règle de gestion de la mémoire ainsi qu'une méthode pour s'assurer que la règle est appliquée dans chacune de nos classes.

Nous avons également examiné les 2 erreurs les plus courantes et la façon de les éliminer :

- les fuites mémoire et la chasse aux zombis avec l'instrument Leaks ;
- les références obsolètes et leur localisation avec le Débogueur.

Ce parcours a été l'occasion d'approfondir quelques éléments de la programmation sur iPhone :

- le compteur de références et les messages `-retain` et `-release` ;
- les attributs de la clause `@property` ;
- le pool d'autolibération et le message `-autorelease` ;
- le traitement des demandes de libération mémoire dans les contrôleurs de vues dans la méthode `-viewDidUnload` ;
- le fonctionnement de la boucle d'événement ;
- le principe de factorisation du code.

MOTIFS FONDAMENTAUX

Nous allons faire connaissance avec quelques motifs de conception (*design patterns*) fondamentaux de la programmation sur iPhone : la **délégation**, le motif **Modèle-Vue-Contrôleur** (**MVC**, *Model-View-Controller*), le **codage par valeur de clé** (**KVC** et *Key Value Coding*). Ces mécanismes permettront d'améliorer notre convertisseur de monnaies et d'aboutir à une application de niveau professionnel. Prenons en compte les éléments suivants :

- Nous voulons interdire la frappe de lettres.
- Nous souhaitons pouvoir faire disparaître le clavier.
- Nous allons vraisemblablement trouver d'autres améliorations.

4.1. Mécanisme de délégation

Délégué

Commençons par interdire à l'utilisateur de frapper des lettres.

Nous voulons modifier le comportement de l'objet `UITextField` vis-à-vis de l'utilisateur. La manière la plus directe d'y parvenir serait de modifier l'objet lui-même, ou de créer un autre objet nommé `UINumericField` par exemple – qui hérite de `UITextField`. Il faudrait ensuite réécrire quelques méthodes dérivées de celles de `UITextField` pour donner à `UINumericField` le comportement souhaité. Tout cela paraît compliqué. La dérivation de classe est en effet une opération délicate qui nécessite une très bonne compréhension du fonctionnement de la classe mère.

La **délégation** permet de modifier le comportement d'un objet sans avoir à le modifier ni le dériver. Les objets `UITextField` disposent d'un outlet *delegate* (objet délégué) configurable sous Interface Builder (voir Figure 4.1).

L'objet délégué reçoit un message avant ou après chaque opération effectuée par son propriétaire :

- pour demander l'autorisation au délégué d'effectuer cette opération ;
- pour l'informer et lui permettre de compléter l'opération.

De façon générale, une information est transmise par un message dont la méthode est fonction de l'opération, qui ne retourne pas de valeur et prend comme unique paramètre une référence à l'objet effectuant l'opération. Par exemple `-(void) textFieldDidBegin`

`Editing: (UITextField *)textField` pour notifier au délégué que le champ de texte vient d'entrer en mode Édition.

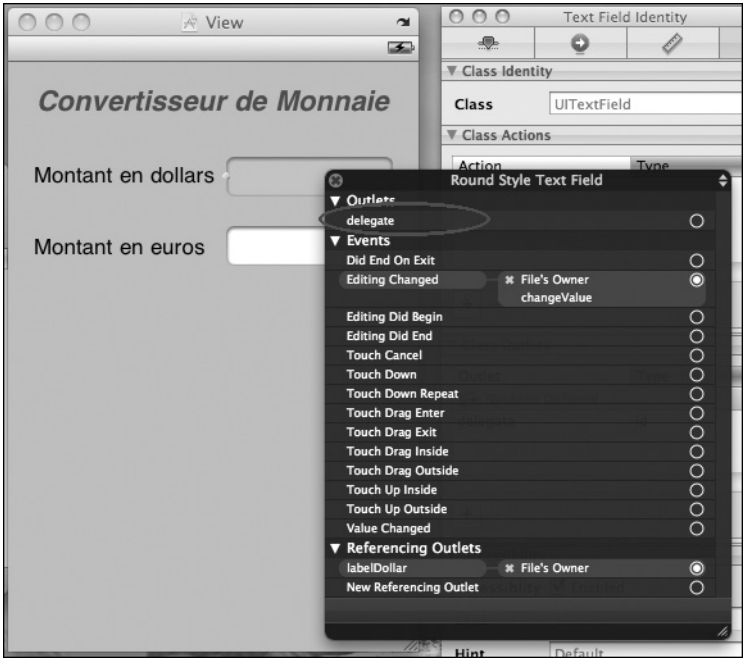


Figure 4.1 : Outlet delegate d'un objet UITextField

Une demande d'autorisation est très souvent un message qui renvoie une valeur de type `BOOL`. Si le délégué autorise l'opération, il renvoie `YES` ; autrement, il renvoie `NO`. Par exemple – `(BOOL) textFieldShouldBeginEditing: (UITextField *)textField` pour demander au délégué si le champ de texte peut entrer en mode Édition.

Type BOOL

`BOOL` est un type prédéfini dans Objective-C qui représente une valeur logique ou **booléenne**. Les deux seules valeurs que peut prendre une variable de type `BOOL` sont `YES` et `NO`.

Déléguer le champ dollar

Nous allons apporter une première amélioration dans *Convertisseur1* que nous perfectionnerons par la suite. Notre objectif sera de limiter la frappe aux caractères numériques dans le champ de texte contenant la valeur en dollars.

Création du délégué

1 Ouvrez le fichier *Convertisseur1ViewController.xib* sous Interface Builder et connectez l'outlet **delegate** du champ de texte à l'objet **File's Owner**.

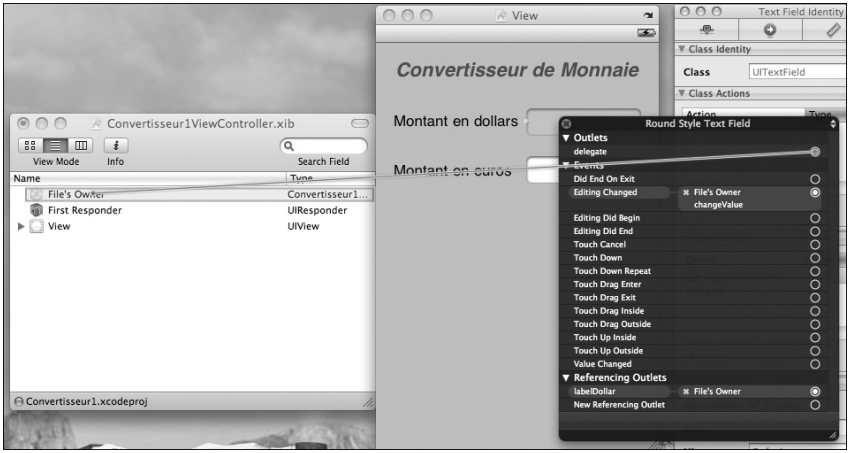


Figure 4.2 : Définir Convertisseur1ViewController comme délégué du champ dollar

Le contrôleur de vue comme délégué

Il est courant de définir le contrôleur de vue comme délégué des objets contenus dans cette vue principale, surtout dans les applications simples.

2 Sous XCode, modifiez le fichier *Convertisseur1ViewController.h* pour indiquer que cette classe est aussi un délégué de champ de texte :

```
#import <UIKit/UIKit.h>
@interface Convertisseur1ViewController :
          UIViewController <UITextFieldDelegate> {
   IBOutlet UITextField *labelDollar;
   IBOutlet UITextField *labelEuro;
}
@property (retain,nonatomic) UITextField *labelDollar;
@property (retain,nonatomic) UITextField *labelEuro;
- (IBAction) changeValue;
@end
```

3 Ajoutez la définition de la méthode suivante dans le fichier *Convertisseur1ViewController.m* :

```
- (BOOL)textField:(UITextField *)textField
```

```
        shouldChangeCharactersInRange:(NSRange)range
        replacementString:(NSString *)string {
    return [[NSCharacterSet decimalDigitCharacterSet]
        isSupersetOfSet:[NSCharacterSet
            characterSetWithCharactersInString:string]];
}
```

La méthode `-textField:shouldChangeCharactersInRange:replacementString` est invoquée sur le délégué chaque fois que l'utilisateur modifie le contenu du champ de texte. Le paramètre `range` (intervalle) indique à quel endroit dans le texte existant la modification sera effectuée et éventuellement quels sont les caractères qui seront supprimés. Le paramètre `string` contient la chaîne de caractères que l'utilisateur souhaite insérer ou ajouter au champ de texte.

Classe NSCharacterSet

Nous en profitons pour faire connaissance avec la classe permettant de manipuler des ensembles de caractères, `NSCharacterSet`, et trois de ses méthodes :

- `+decimalDigitCharacterSet` construit l'ensemble de caractères formé des chiffres de 0 à 9.
- `+characterSetWithCharactersInString:` construit l'ensemble des caractères inclus dans la chaîne de caractères passée en paramètre.
- `-isSuperSetOf:` retourne `YES` si l'ensemble passé en paramètre est inclus dans l'ensemble récepteur.

La démarche pour vérifier que l'utilisateur saisit uniquement des chiffres est la suivante :

1. Constituez un ensemble de caractères composé des caractères saisis.
2. Constituez l'ensemble des caractères numériques (chiffres).
3. Vérifiez que le premier ensemble est inclus dans le second.

Taille de la saisie de caractères

On pourrait simplifier l'écriture de cette méthode de délégué en supposant que l'utilisateur ne peut saisir qu'un caractère à la fois ; il suffirait de tester si ce caractère est compris entre 0 et 9. Malheureusement, cette simplification serait erronée car l'utilisateur peut coller un texte préalablement copié et donc "saisir" plusieurs caractères en une fois dans un champ de texte.

Figure 4.3 : Saisie de plusieurs caractères

4 Reconstruisez l'application et testez-la sur le simulateur. Vérifiez que vous ne pouvez plus saisir de lettres ni de signes de ponctuation. Notre délégué fonctionne.

Délégué pour un champ de texte

Dans la section précédente, nous avons défini l'objet `Convertisseur1ViewController` comme délégué du champ de texte contenant le montant en dollars, puis nous y avons ajouté une méthode. Nous allons détailler ici toutes les méthodes que l'on pourrait écrire au besoin.

Méthodes du délégué

Son délégué est interrogé par un champ de texte lors des situations suivantes pour savoir si :

- l'édition peut commencer ;
- l'édition peut se terminer, c'est l'occasion de vérifier la validité du texte saisi par l'utilisateur et d'alerter ce dernier si le champ n'est pas correctement saisi ;
- le texte peut être modifié lors de chaque opération de saisie, c'est cette situation que nous venons d'utiliser ;
- le texte peut être effacé en début de saisie ;

- l'action consécutive à la frappe de la touche ↵ est autorisée.

Le délégué est également informé dans les situations suivantes ; cela lui permet ainsi d'effectuer des actions complémentaires que ne saurait pas faire le champ de texte :

- L'édition vient de commencer.
- L'édition vient de se terminer.

Le tableau ci-après résume les messages émis vers le délégué par le champ de texte dans chacune de ces situations. Pour chaque message, la déclaration précise de la méthode est indiquée.

Tableau 4.1 : Méthodes du protocole UITextFieldDelegate

Signature de la méthode	Objet de la méthode
`- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField`	Demande au délégué si l'édition du champ de texte peut commencer.
`- (void)textFieldDidBeginEditing:(UITextField *)textField`	Informe le délégué que l'édition du champ de texte vient de commencer.
`- (BOOL)textFieldShouldEndEditing:(UITextField *)textField`	Demande au délégué si l'édition du champ de texte peut se terminer.
`- (void)textFieldDidEndEditing:(UITextField *)textField`	Informe le délégué que l'édition du champ de texte vient de se terminer.
`- (BOOL)textField:(UITextField *)textField shouldChangeCharactersInRange:(NSRange)range replacementString:(NSString *)string`	Demande au délégué si le texte peut être modifié.
`- (BOOL)textFieldShouldClear:(UITextField *)textField`	Demande au délégué si le contenu du champ de texte peut être effacé.
`- (BOOL)textFieldShouldReturn:(UITextField *)textField`	Demande au délégué s'il faut utiliser le comportement par défaut de la touche ↵ qui vient d'être frappée.

Toutes les méthodes prennent un paramètre `textField`. Le champ de texte communique systématiquement une référence sur lui-même à son délégué. Cela permet à un objet d'être le délégué de plusieurs autres objets ; il sait qui lui transmet un message.

Méthodes optionnelles

Dans notre classe `Convertisseur1ViewController`, nous avons écrit seulement l'une des 7 méthodes définies pour un délégué de champ de texte. Voici une caractéristique intéressante du langage Objective-C : dans de nombreux autres langages, nous aurions été obligés d'écrire les 7 méthodes dont 6 qui ne faisaient rien.

Comment fait le champ de texte pour savoir s'il peut ou non émettre un message vers son délégué ?

Appel de méthode inexistante

L'appel d'une méthode inexistante sur un objet provoque généralement le plantage de l'application et la levée de l'exception `EXC_BAD_INSTRUCTION`.

respondsToSelector

La classe `NSObject`, de laquelle dérive directement ou indirectement toutes les autres classes, définit une méthode `respondsToSelector:` disponible pour tous les objets.

```
- (BOOL) respondsToSelector:(SEL)aSelector
```

Le type `SEL` est un sélecteur, c'est-à-dire un pointeur sur une méthode. La primitive `@selector` d'Objective-C permet d'obtenir un sélecteur à partir du nom complet d'une méthode (sans les types ni les noms de paramètres). Ainsi le champ de texte peut interroger le délégué pour savoir s'il répond à une méthode avant de lui transmettre un message.

```
SEL aSelector = @selector(
textField:shouldChangeCharactersInRange:replacementString:);
if ( [delegate respondsToSelector:aSelector] ) …
```

Déclarer un protocole

Adopter un protocole

Souvenez-vous de la petite modification que nous avons effectuée dans le fichier *Convertisseur1ViewController.h* :

```
@interface Convertisseur1ViewController :
       UIViewController <UITextFieldDelegate> {
```

L'ajout de `<UITextFieldDelegate>` permet de préciser que la classe que nous déclarons adopte le protocole `UITextFieldDelegate`. Nous pouvons spécifier une liste de protocoles entre les crochets, en les séparant par une virgule.

Protocole

Un protocole est une liste de déclarations de méthodes, certaines requises d'autres optionnelles. Cette liste n'est pas attachée a priori à une classe particulière. Il appartient à la classe qui adopte un protocole de définir toutes les méthodes requises et les méthodes optionnelles nécessaires.

Les protocoles Objective-C sont équivalents aux interfaces Java.

Les frameworks de Cocoa Touch définissent un protocole pour chaque type de délégué. Par exemple, la classe `UITextField` a besoin d'un objet délégué qui adopte le protocole `UITextFieldDelegate`.

Lorsque vous définissez un objet qui doit être un délégué, n'oubliez pas d'indiquer le protocole qu'il adopte.

Définir un protocole

Vous pouvez définir vos propres protocoles à l'aide des mots-clés `@protocol`, `@optional` et `@end` :

```
@protocol nom-du-protocole
// Déclarations des méthodes requises
@optional
// Déclarations des méthodes optionnelles
@end
```

Par exemple, le protocole `UITextFieldDelegate` pourrait être déclaré comme suit :

```
@protocol UITextFieldDelegate
@optional
- (BOOL)textFieldShouldBeginEditing:(UITextField*)textField;
- (void)textFieldDidBeginEditing:(UITextField *)textField;
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField;
- (void)textFieldDidEndEditing:(UITextField *)textField;
- (BOOL)textField:(UITextField *)textField
          shouldChangeCharactersInRange:(NSRange)range
                      replacementString:(NSString *)string;
- (BOOL)textFieldShouldClear:(UITextField *)textField;
- (BOOL)textFieldShouldReturn:(UITextField *)textField;
@end
```

Lancement de l'application

Les autres délégués

Nous avons détaillé le protocole du délégué pour un champ de texte. Les frameworks de Cocoa Touch proposent une quarantaine de protocoles de délégué ; tous portent un nom qui se termine par `Delegate`.

Il est important de bien comprendre le mécanisme de délégation. En effet, ce sont des objets délégués que le programmeur Cocoa Touch créera pour modifier le comportement des objets standard fournis par Apple : vues, contrôleurs, etc.

Vous êtes-vous aperçu que nous avons utilisé un autre délégué ? Celui de la classe `UIApplication`. Nous allons examiner dans cette section ce qui se passe lorsqu'on lance l'application. Pour cela, nous ferons connaissance avec le protocole `UIApplicationDelegate`.

UIApplication

Regardez sous XCode le contenu du fichier *main.m* :

```
#import <UIKit/UIKit.h>
int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool =
                          [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Vous n'aurez généralement pas à modifier ce code mais il est intéressant de comprendre le rôle de cette fonction. Sous un système de la famille UNIX, tel que Linux, iPhone OS ou MacOSX, toute application doit contenir une fonction `main` appelée lors de son lancement. Cette fonction réalise les opérations suivantes :

- création d'un pool d'autolibération principal pour l'application l'application (rappelez-vous qu'il sert à libérer les instances dont la durée de vie est limitée à une boucle d'événements) ;
- appel de la fonction `UIApplicationMain` qui va effectuer tout le travail :
  - création d'un objet de la classe `UIApplication` ;
  - chargement du fichier *MainWindow.xib* dont l'objet `UIApplication` sera propriétaire ;
  - création de la boucle d'événement ;
  - lancement de la boucle d'événement tant que l'application n'a pas été quittée ;
  - fonction terminée lorsque l'application est quittée.
- libération du pool d'autolibération puisque l'application se termine ;
- Fin de l'application.

MainWindow.nib

Le fichier *MainWindow.xib* est donc chargé au lancement de l'application. Jetons un coup d'œil dans ce fichier à l'aide d'Interface Builder.

Nom du fichier NIB principal

Le nom du fichier NIB principal est la valeur de la propriété **Main nib file base name** du fichier *Info.plist* de l'application. XCode le définit par défaut à `MainWindow`, il est possible de le changer.

Le fichier NIB principal contient 5 objets :

- Le propriétaire (*File's Owner*) comme dans tout fichier NIB. Il est ici de type UIApplication puisque il doit être chargé par l'application.
- Le premier répondeur (*First Responder*) également comme dans tout fichier NIB. Cet objet représente la vue de l'interface en cours d'édition par l'utilisateur. On utilise cet objet comme cible pour des actions que l'on veut transmettre à cette vue.
- Un objet Window de type UIWindow. Cet objet est la fenêtre de l'application dans laquelle toutes les vues seront affichées.
- *Convertisseur1 App Delegate*, de type *Convertisseur1AppDelegate*, que nous allons examiner.
- *Convertisseur1 View Controller* de type *Convertisseur1View Controller* que nous connaissons déjà car nous avons modifié cette classe.

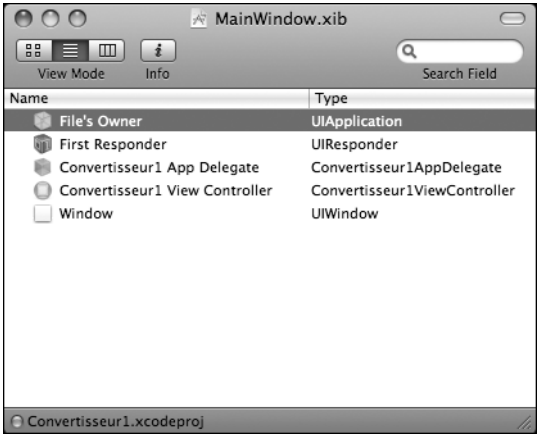


Figure 4.4 : Contenu de MainWindow.nib

Toujours sous Interface Builder, examinez les connexions entre ces différents objets (voir Figure 4.5).

Au chargement du fichier *MainWindow.xib*, les objets du fichier sont créés et reliés entre eux en utilisant les outlets (voir Figure 4.6).

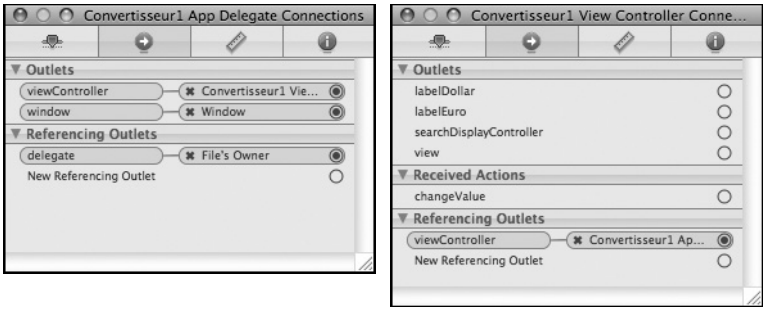


Figure 4.5 : Connexions dans MainWindow.xib

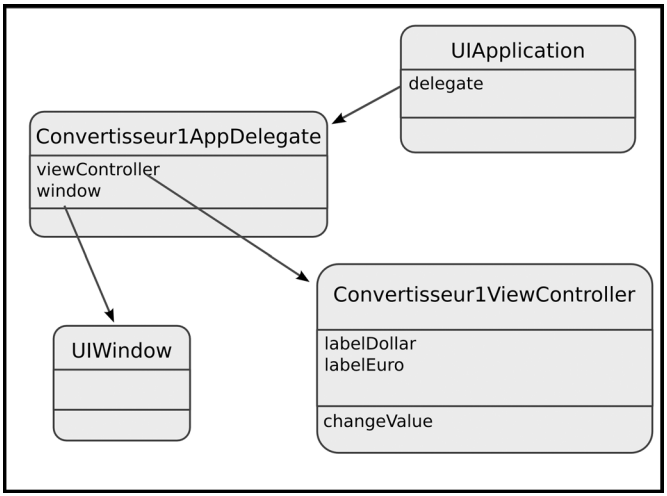


Figure 4.6 : Structure d'objets créée au chargement de MainWindow.xib

Le délégué de l'application est créé et lié à l'application à ce moment-là.

Délégué d'application

Nous connaissons le contrôleur de vue `Convertisseur1ViewController`. Faisons connaissance au délégué d'application que XCode a créé pour nous.

Regardez le contenu du fichier *Convertisseur1AppDelegate.h* :

```
#import <UIKit/UIKit.h>
@class Convertisseur1ViewController;
@interface Convertisseur1AppDelegate : NSObject
                                    <UIApplicationDelegate> {
   UIWindow *window;
   Convertisseur1ViewController *viewController;
```

```
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet
            Convertisseur1ViewController *viewController;
@end
```

Seule l'instruction `@class` est nouvelle. Nous savons déjà interpréter les autres lignes de code :

- Une classe `Convertisseur1AppDelegate` est déclarée, qui hérite de `NSObject` et adopte le protocole `UIApplicationDelegate` ; il s'agit d'un délégué d'application.
- Les outlets `window` et `viewController` sont déclarés ; ce sont ceux utilisés dans le fichier *MainWindow.xib* pour relier cette classe aux autres objets du fichier NIB.

@class

La clause `@class` permet d'identifier un nom comme étant celui d'une classe. Ce nom pourra ensuite être utilisé pour déclarer des objets de cette classe.

On utilise `@class nom-de-la-classe;` lorsqu'on veut simplement déclarer des objets de cette classe, sans les utiliser. On emploie `#import "nom-de-la-classe.h"` lorsqu'on veut utiliser des objets de cette classe, leur transmettre des messages.

Où en sommes-nous dans le processus de lancement de l'application ?

- Un pool d'autolibération a été créé.
- Un objet *UIApplication* a été créé et il est activé.
- Le fichier *MainWindow.nib* a été chargé et une structure d'objets a été créée qui comprend :
  - une fenêtre (*window*) pour l'application ;
  - un délégué (*Convertisseur1AppDelegate*) ;
  - un contrôleur de vue (*Convertisseur1ViewController*).

L'application est sur le point de lancer la boucle d'événement. Auparavant, elle va informer son délégué que le lancement vient de se terminer en émettant le message `applicationDidFinishLaunching`.

Examinons le code source du délégué d'application, fichier *Convertisseur1AppDelegate.m* :

```
#import "Convertisseur1AppDelegate.h"
#import "Convertisseur1ViewController.h"
@implementation Convertisseur1AppDelegate
@synthesize window;
@synthesize viewController;
- (void)applicationDidFinishLaunching:(UIApplication *)
                                                application {
    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}
- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}
@end
```

Le délégué affiche la vue contrôlée par notre instance de `Convertisseur1ViewController` puis la fenêtre est affichée et activée sur l'écran de l'appareil. Qu'est-ce qui est affiché à l'écran précisément ? Examinez le contrôleur de vue dans le fichier *MainWindow.xib* sous Interface Builder.

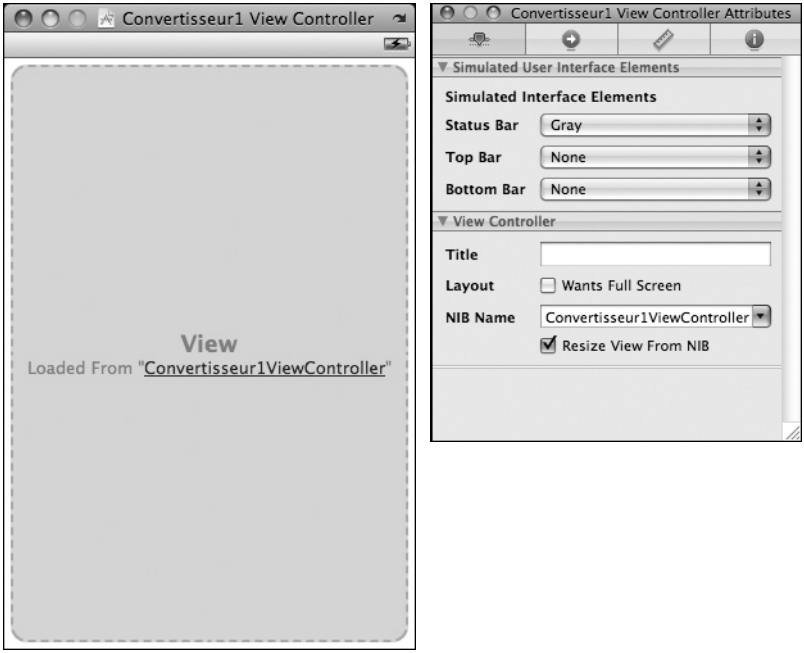


Figure 4.7 : Convertisseur1ViewController dans MainWindow.xib

Le chargement du fichier NIB *MainWindow.xib* crée un objet `Convertisseur1ViewController` associé au fichier NIB *Convertisseur1 ViewController.xib*. C'est l'utilisation de la propriété `view` du contrôleur de vue qui provoque le chargement du fichier NIB associé.

Structurer une application

Ce principe peut être étendu à la plupart des applications sur iPhone OS :

- L'application charge un fichier NIB qui contient :
  - une fenêtre et éventuellement des vues ;
  - généralement un délégué pour le propriétaire du fichier NIB ;
  - un ou plusieurs contrôleurs de vue.
- Chaque contrôleur de vue possède lui-même un fichier NIB qui contient :
  - une hiérarchie de vue à l'intérieur de sa vue principale ;
  - à son tour, un ou plusieurs contrôleurs de vue ;
  - éventuellement d'autres objets.
- Chaque contrôleur peut également posséder un fichier NIB, etc.

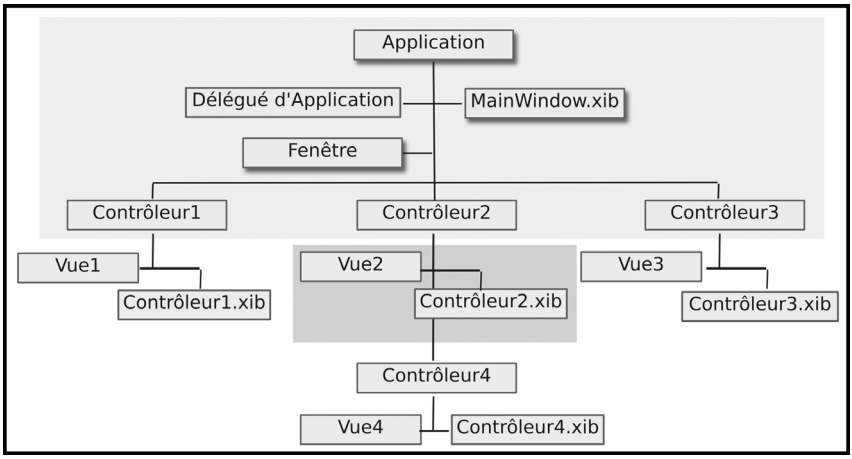


Figure 4.8 : Structuration classique d'une application

4.2. Améliorer Convertisseur1

Nous avons réussi à mettre en œuvre le délégué pour limiter la frappe aux seuls chiffres mais le résultat n'est pas très satisfaisant, en tout cas loin d'une application professionnelle :

- Il n'y a plus moyen de saisir des chiffres après la virgule.
- Ce n'est même pas une virgule, c'est un point qui s'affiche pour séparer les centimes.
- Toujours pas moyen de se débarrasser du clavier qui reste bêtement affiché tout le temps.
- Le nom de l'application ne s'affiche pas en entier sous le logo.
- La moindre des choses serait de pouvoir faire les conversions dans les deux sens.

Nous avons du pain sur la planche ; ne traînons pas.

Retrouver la virgule

Si nous voulons pouvoir saisir un point décimal ou une virgule, il faut autoriser la frappe de l'un de ces caractères dans le délégué du champ de texte. Ce n'est pas si simple ; on pourrait alors saisir plusieurs virgules, ce qui n'est pas autorisé pour représenter un nombre.

Il serait plus simple de vérifier au fur et à mesure de la saisie que la chaîne de caractères obtenue est la représentation d'un nombre. Et pour le vérifier, le plus simple est de convertir cette chaîne en nombre. Puisqu'il faut économiser les ressources précieuses de l'iPhone, évitez d'effectuer cette conversion une fois dans la méthode du délégué et juste après dans la méthode `changeValue`.

Nous allons créer un objet dont le rôle sera d'effectuer cette vérification, la conversion en nombre et la conversion en dollars.

Objet Convertisseur

1 Créez un nouveau fichier de type **classe Objective-C** sous XCode, en sélectionnant la commande **New File...** du menu **File** (⌘+N). Nommez-le *Convertisseur.m* et cochez la case *Also create "Convertisseur.h"* (voir Figure 4.9).

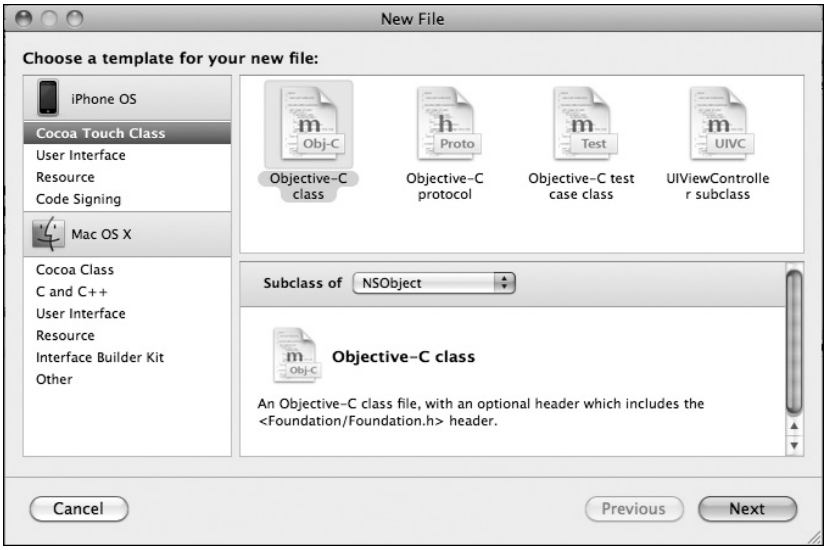


Figure 4.9 : Créer une classe Objective-C

2 Modifiez le fichier *Convertisseur.h* comme suit :

```
#import <Foundation/Foundation.h>
@interface Convertisseur : NSObject {
   float    euro;
   float    dollar;
   float    dollarsPourUnEuro;
}
@property (nonatomic,assign) float euro;
@property (nonatomic,assign) float dollar;
@property (nonatomic,assign) float dollarsPourUnEuro;
- (BOOL) setDollarWithString: (NSString *) string;
@end
```

Notre intention est que la méthode `-setDollarWithString:` retourne `YES` si la chaîne string représente un montant en dollars. Dans ce cas, il est aussi converti en euros. Si ce n'est pas le cas, la méthode doit retourner `NO` et les valeurs euro et dollar ne sont pas modifiées.

Il faudra aussi initialiser la propriété `dollarsPourUnEuro` quelque part ; dans la méthode `-init` de la classe, c'est le plus simple.

3 Modifiez le fichier *Convertisseur.m* :

```
#import "Convertisseur.h"
@implementation Convertisseur

@synthesize euro;
@synthesize dollar;
@synthesize dollarsPourUnEuro;
```

```
-(id) init {
   if (self = [super init]) {
      self.dollarsPourUnEuro = 1.4908;
   }
   return self;
}

-(BOOL) setDollarWithString:(NSString *)string {
   float valeur;
   BOOL result;
   NSScanner *scan = [NSScanner
                      localizedScannerWithString:string];
   [scan scanFloat:&valeur];
   result = [scan isAtEnd];
   if (result) self.dollar = valeur;
   return result;
}

-(void) setDollar:(float)newValue {
   dollar = newValue;
   euro = newValue / dollarsPourUnEuro;
}

@end
```

Classe NSScanner

Nous rencontrons la classe `NSScanner` pour la première fois. C'est une chaîne de caractères à laquelle on a ajouté un curseur (propriété `scanLocation`) initialisé à `0` à la création d'une instance. Chaque fois qu'une instance de `NSScanner` reçoit un message de conversion (par exemple `–scanFloat:`), le curseur progresse jusqu'au prochain caractère qui ne peut participer à la conversion. Une conversion est effectuée à partir de la position du curseur. Ainsi, la chaîne passée en paramètre à la méthode `–setDollarWithString:` est un nombre si le curseur est à la fin de la chaîne après la conversion.

Nous avons ici une petite particularité : les méthodes de conversion telle que `–scanFloat:` retourne 2 valeurs :

- La valeur de retour est un `BOOL` qui indique si la conversion est réussie (que nous n'utilisons pas ici).
- Une valeur convertie au format souhaité (`float` pour `–scanFloat:`).

Le paramètre passé à `–scanFloat:` est `&valeur` et pas simplement `valeur`. Comme nous souhaitons récupérer une valeur dans la variable `valeur`, ce n'est pas sa valeur qu'il faut passer mais l'adresse de la variable. Nous utilisons donc `&`, l'opérateur de référencement du langage C.

Opérateur de référencement
L'adresse d'une variable s'obtient en utilisant l'opérateur `&` juste avant le nom de la variable.

Déclarer une propriété Convertisseur

Nous avons créé une classe *Convertisseur*. Pour l'utiliser, il faut en déclarer une instance dans un des objets de notre application. Le plus logique est de la créer dans la classe `Convertisseur1ViewController`, c'est là que nous en aurons besoin. Nous allons donc ajouter une propriété dans notre contrôleur de vue.

1 Modifiez le fichier *Convertisseur1ViewController.h* :

```
#import <UIKit/UIKit.h>
@class Convertisseur;

@interface Convertisseur1ViewController : UIViewController
                                    <UITextFieldDelegate> {
   IBOutlet UITextField *labelDollar;
   IBOutlet UITextField *labelEuro;
   Convertisseur *convertisseur;
}
@property (retain,nonatomic) UITextField *labelDollar;
@property (retain,nonatomic) UITextField *labelEuro;
@property (retain,nonatomic) Convertisseur *convertisseur;

- (IBAction) changeValue;

@end
```

2 Dans le fichier *Convertisseur1ViewController.m*, modifiez les méthodes `-viewDidLoad` (enlevez les commentaires `/*` et `*/`) et `-viewDidUnload` :

```
// Implement viewDidLoad to do additional setup after
//      loading the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
    convertisseur = [[Convertisseur alloc] init];
}
- (void)viewDidUnload {
   // Release any retained subviews of the main view.
   // e.g. self.myOutlet = nil;
   self.labelDollar = nil;
   self.labelEuro = nil;
   self.convertisseur = nil;
}
```

3 N'oubliez pas d'ajouter `#import "Convertisseur.h"` en tête du fichier et `@synthesize convertisseur;` avec les autres synthétiseurs de propriété.

La méthode `–viewDidLoad` est invoquée à la fin du chargement du fichier NIB du contrôleur de vue. C'est le bon endroit pour effectuer les initialisations du contrôleur ; on est certain que toutes les vues sont créées et que les outlets sont utilisables si besoin.

Modifier le délégué

Modifiez la méthode de délégué dans le fichier *Convertisseur1 ViewController.m* :

```
- (BOOL)textField:(UITextField *)textField
        shouldChangeCharactersInRange:(NSRange)range
                    replacementString:(NSString *)string {
   NSString *resultingString = [textField.text
                stringByReplacingCharactersInRange:range
                                        withString:string];
   return [self.convertisseur
                    setDollarWithString:resultingString];
}
```

Nous constituons une chaîne de caractères `resultingString` dont la valeur est celle qu'aura le champ de texte après lui avoir appliqué la modification demandée par l'utilisateur. Ensuite, nous utilisons notre instance de *Convertisseur* pour vérifier que l'on peut effectuer la conversion.

Modifier l'action

1 Modifiez la méthode `changeValue` dans le fichier *Convertisseur1 ViewController.m*. Cette méthode devient beaucoup plus simple car le travail de conversion est maintenant effectué par la classe `Convertisseur` :

```
- (IBAction) changeValue {
   NSString *textEuro = [NSString localizedStringWithFormat:
                         @"%.2f",self.convertisseur.euro];
   labelEuro.text = textEuro;
}
```

textEuro n'est plus libéré

L'instance `textEuro` n'étant plus créée par la méthode `+alloc`, nous savons qu'elle est dans le pool d'autolibération. Il ne faut donc pas la libérer explicitement, sauf si nous la retenions, ce qui n'est pas le cas ici.

2 Reconstruisez et testez l'application.

Localiser les montants

Vous avez remarqué deux des nouvelles méthodes que nous avons utilisées :

- `+localizedScannerWithString:` de la classe `NSScanner` ;
- `+localizedStringWithFormat:` de la classe `NSString`.

Chaîne localisée

Une chaîne de caractères localisée est un texte dont le format précis dépend des réglages régionaux de l'appareil. Les textes localisés concernent les dates, les heures, les numéros de téléphone et les nombres à virgule. Le séparateur décimal est une virgule en France et un point dans les pays anglo-saxons.

Pour vérifier si la localisation des montants en euros et en dollars fonctionne, nous allons modifier les réglages du simulateur d'iPhone.

1 Lancez l'application *Settings* (ou *Réglages* si le simulateur est réglé en français).

Figure 4.10 : Application Settings sur le simulateur

2 Choisissez **Général** puis **International**. L'option **Format régional** (**Region Format**) vous permet de changer le paramètre régional du simulateur. Vous pouvez en profiter aussi pour changer la langue de l'iPhone OS.

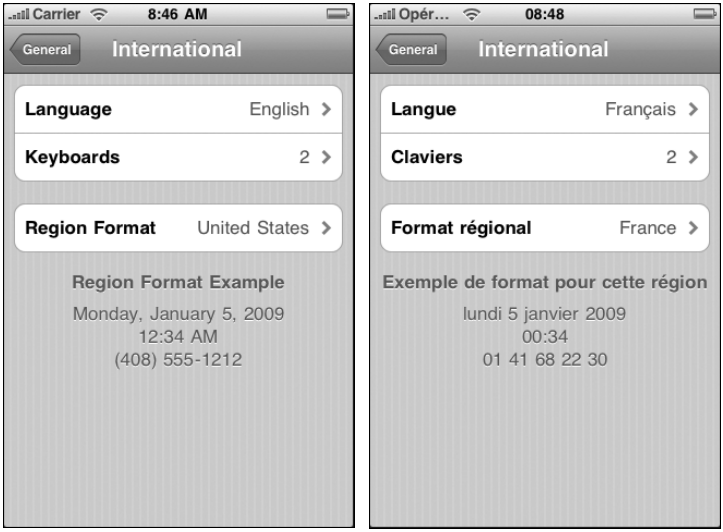


Figure 4.11 : Localisation du simulateur

3 Testez l'application dans les formats régionaux **France** et **États-Unis** pour vérifier que le séparateur des nombres décimaux est une virgule ou un point suivant le cas.

Figure 4.12 : Test pour les régions États-Unis et France

Localiser l'interface utilisateur

Nous venons de voir comment prendre en compte les paramètres régionaux pour mettre en forme les nombres ; pensez aux méthodes contenant le terme `localized` dans leur nom. Pour localiser vraiment l'application, il faut aussi que l'interface utilisateur soit présentée dans sa langue. Nous localiserons donc le fichier NIB de la vue principale.

1 Sous XCode, sélectionnez le fichier *Convertisseur1View Controller.xib* et cliquez du bouton droit pour afficher le menu contextuel. Sélectionnez la commande **Get Info**.

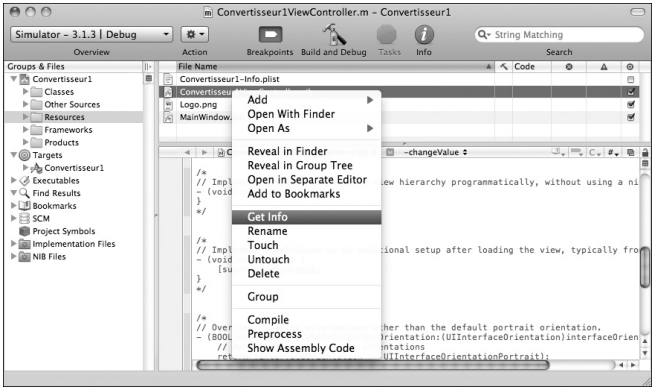


Figure 4.13 : Afficher les informations relatives au fichier NIB

2 Dans la fenêtre d'informations qui s'affiche, cliquez sur le bouton **Make File Localizable**.

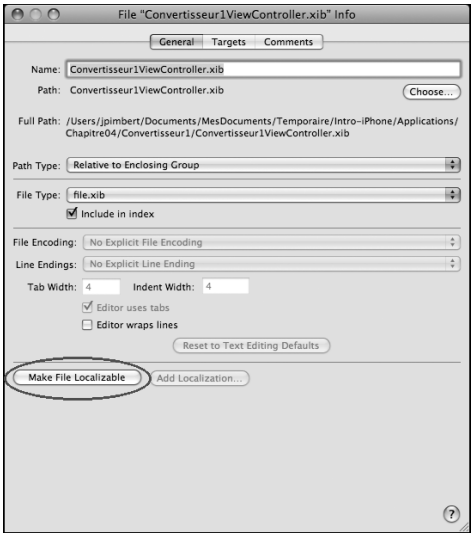


Figure 4.14 : Créer une version localisée du fichier NIB

La fenêtre change de titre car nous venons de créer un groupe localisé.

3 Dans l'onglet **General**, cliquez sur le bouton **Add Localization** pour créer une version française (*French*) en plus de la version anglaise (*English*) existante.

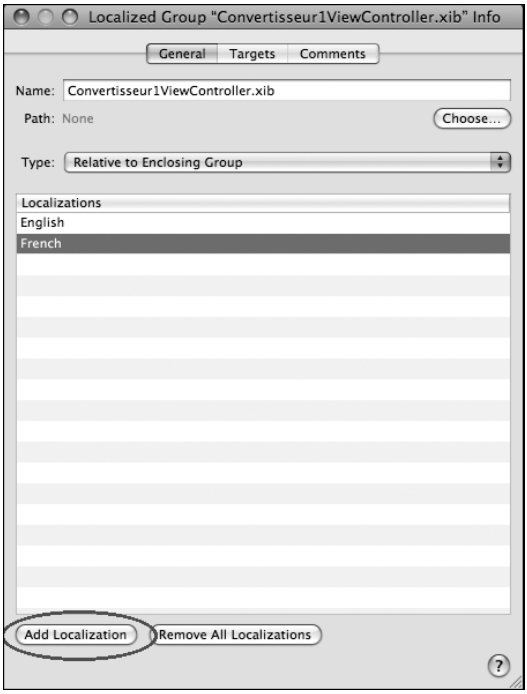


Figure 4.15 : Le groupe localisé contient une version française et une version anglaise

Comme notre version actuelle est en français, la version française du fichier NIB est correcte et il faut modifier la version anglaise.

4 Fermez la fenêtre d'information et sous XCode, double-cliquez sur la version anglaise du fichier *Convertisseur1ViewController.xib (English)* pour l'ouvrir sous Interface Builder.

Interface Builder est muni d'un outil très pratique pour réaliser les traductions.

5 Activez la commande **Strings** du menu **Tools**. Une fenêtre s'affiche qui présente toutes les chaînes de caractères contenues dans le fichier NIB.

6 Saisissez les traductions dans la colonne *Value* pour chacune des chaînes de caractères : `Currency Converter`, `US $ amount` et `Euros amount`.

Figure 4.16 : Menu Tools d'Interface Builder

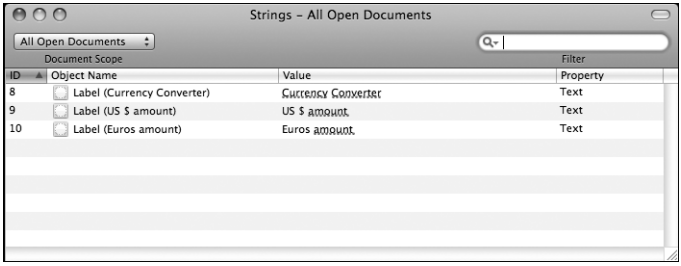


Figure 4.17 : Traduction des chaînes de caractères du fichier NIB

7 Enregistrez le fichier NIB et revenez sous XCode. Avant de reconstruire l'application, il faut nettoyer les constructions précédentes. Autrement, la version non localisée du fichier NIB continuera à être installée et utilisée.

8 Sélectionnez la commande **Clean All Targets** du menu **Build**.

9 Dans la boîte de dialogue qui s'affiche, cochez toutes les cases et cliquez sur **Clean**.

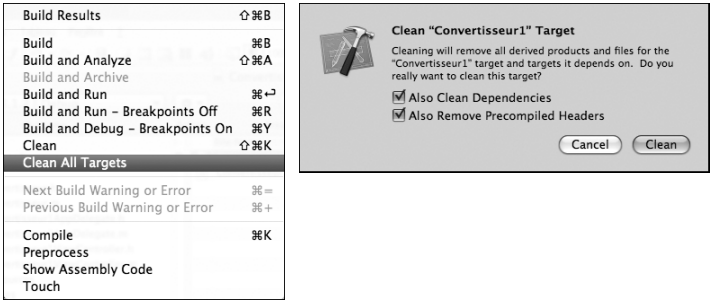


Figure 4.18 : Nettoyage des constructions précédentes

10 Reconstruisez l'application et testez-la avec le simulateur d'iPhone. Essayez-la avec les paramètres régionaux US, UK et France et dans les langues anglaise et française.

Figure 4.19 : Application localisée

Nettoyage (Clean)

Lorsque nous modifions la liste des ressources de l'application, il est prudent de faire un nettoyage des constructions afin que les ressources devenues inutiles ne perturbent pas le fonctionnement de l'application.

Utiliser le motif KVC

Il est regrettable que notre convertisseur ne fonctionne que dans un sens, des dollars vers les euros. Remédions à cela. Nous aurons ainsi l'opportunité d'aborder le puissant **codage par valeur de clé** (*KVC*, *Key Value Coding*).

Adapter la classe Convertisseur

1 Ajoutez une méthode `–setEuro:` dans le fichier *Convertisseur.m* pour effectuer la conversion en dollars. Cette méthode est analogue à `–setDollar:` :

```
-(void) setEuro:(float)newValue {
    euro = newValue;
    dollar = newValue * dollarsPourUnEuro;
}
```

setEuro est déjà déclaré par @property

Il n'est pas utile de déclarer `-setEuro:` dans l'interface de la classe car `euro` est déjà déclaré comme une propriété. Nous écrivons en fait un manipulateur qui remplace celui créé par défaut par la clause `@synthesize`.

Il faudrait également ajouter une méthode `-setEuroWithString:` qui serait analogue à `-setDollarWithString:`. Nous allons plutôt modifier la méthode existante pour qu'elle fonctionne dans les deux cas. Souvenez-vous ; il faut essayer de **factoriser** le code pour qu'il soit plus facile à maintenir.

2 Modifiez le fichier *Convertisseur.h*. Remplacez la déclaration de la méthode `-setDollarWithString:` par la méthode suivante.

```
- (BOOL) setValueForKey:(NSString *)key WithString:
(NSString *) string;
```

Notre intention est d'utiliser le paramètre `key` pour transmettre le nom de la propriété à modifier, `euro` ou `dollar`.

3 Modifiez la méthode `-setDollarWithString:` dans le fichier *Convertisseur.m*.

```
-(BOOL) setValueForKey:(NSString *)key
          WithString:(NSString *)string {
    float valeur;
    BOOL result;
    NSScanner *scan = [NSScanner
                       localizedScannerWithString:string];
    [scan scanFloat:&valeur];
    result = [scan isAtEnd];
    if (result) [self setValue:[NSNumber
                     numberWithFloat:valeur] forKey:key];
    return result;
}
```

Key Value Coding

Objective-C et NSObject nous proposent d'accéder aux propriétés d'un objet en donnant le nom de la propriété dans une chaîne de caractères.

Accès par méthodes	Accès par clé
`- (id) property`	`- (id) valueForKey: @"property"`
`- (void) setProperty:(id) value`	`- (void) setValue:(id)value forKey: @"property"`

Les méthodes `-valueForKey:` et `-setValue:forKey:` appellent respectivement l'accesseur et le manipulateur par défaut de la propriété dont le nom est passé dans le paramètre `key`. Il est donc indispensable d'utiliser la règle standard de dénomination pour que le **KVC** fonctionne.

Les méthodes du KVC manipulent des valeurs de type `id`, c'est-à-dire une référence à une instance d'objet. Lorsque les propriétés sont de type scalaire (*int*, *float*, etc.), il faut les encapsuler dans un objet `NSNumber`. Les méthodes du KVC réalisent les conversions appropriées entre les instances de `NSNumber` et les valeurs scalaires.

Modifier le délégué

Nous avons modifié la classe `Convertisseur`, il faut donc prendre en compte ce changement dans la méthode de délégué de champ de texte que nous avons écrite dans la classe `Convertisseur1ViewController`. De plus, nous voulons maintenant réagir aux actions de l'utilisateur dans le champ de texte contenant le montant en euros.

Modifiez la méthode de délégué dans le fichier *Convertisseur1ViewController.m* :

```
- (BOOL)textField:(UITextField *)textField
        shouldChangeCharactersInRange:(NSRange)range
                    replacementString:(NSString *)string {
   NSString *resultingString = [textField.text
               stringByReplacingCharactersInRange:range
                                       withString:string];
   if (textField==labelDollar)
      return [self.convertisseur setValueForKey:@"dollar"
                                     withString:resultingString];
   else
      return [self.convertisseur setValueForKey:@"euro"
                                     withString:resultingString];
}
```

Nous utilisons ici le paramètre `textField` de la méthode délégué pour identifier quel champ de texte a transmis le message.

Constante NSString

Nous utilisons ici la notation `@".."` qui permet de définir une constante de type `NSString`, par exemple `@"dollar"`. En Objective-C, on utilise beaucoup plus souvent les constantes `NSString` que les chaînes de caractères C et la notation `".."`.

Modifier l'action

La méthode `–changeValue` doit également être modifiée car les deux champs de texte sont susceptibles de l'invoquer. Nous emploierons une autre variante du prototype d'action `–(IBAction) action: (id)sender`. Le paramètre `sender` est l'objet qui a déclenché l'action.

1 Modifiez le fichier *Convertisseur1ViewController.h* pour utiliser ce nouveau prototype d'action :

```
@interface Convertisseur1ViewController : UIViewController
                                        <UITextFieldDelegate> {
   IBOutlet UITextField *labelDollar;
   IBOutlet UITextField *labelEuro;
   Convertisseur *convertisseur;
}
@property (retain,nonatomic) UITextField *labelDollar;
@property (retain,nonatomic) UITextField *labelEuro;
@property (retain,nonatomic) Convertisseur *convertisseur;
- (IBAction) changeValue:(id)sender;
@end
```

2 Modifiez le code de la méthode `changeValue` dans le fichier *Convertisseur1ViewController.m* :

```
- (IBAction) changeValue:sender {
   NSString *textEuro;
   if (sender==labelDollar) {
      textEuro = [NSString localizedStringWithFormat:
                        @"%.2f",self.convertisseur.euro];
      labelEuro.text = textEuro;
   } else {
      textEuro = [NSString localizedStringWithFormat:
                       @"%.2f",self.convertisseur.dollar];
      labelDollar.text = textEuro;
   }
}
```

3 Enregistrez les fichiers modifiés sous XCode. Vous pouvez également reconstruire l'application pour vérifier que le code saisi est correct.

Établir les connexions

Il faut maintenant établir les connexions dans le fichier NIB de la vue principale.

Fichiers NIB localisés

Il faut effectuer les mêmes modifications dans toutes les versions localisées d'un fichier NIB. Il est parfois plus facile de commencer par supprimer les localisations puis effectuer les modifications dans une version unique pour enfin recréer les versions localisées à partir du fichier NIB unique.

Citons aussi l'outil **ibtool** dont la description sort de cadre de cet ouvrage. Il permet d'automatiser en partie la localisation des différentes modifications d'un fichier NIB.

1. Ouvrez successivement chacune des versions localisées du fichier *Convertisseur1ViewController.xib* sous Interface Builder pour établir les connexions des actions et des délégués. Par prudence, puisque nous venons de modifier la classe `Convertisseur1ViewController`, nous pouvons recharger les déclarations des classes utilisées sous Interface Builder.
2. Sélectionnez la commande **Reload All Class Files** du menu **File**.
3. Dans chacune des versions du fichier NIB, sélectionnez successivement les deux champs de texte pour établir les connexions avec le propriétaire du fichier :
   - l'outlet `delegate` ;
   - l'événement `Editing Changed` sur l'action `changeValue:`.
4. Reconstruisez l'application et testez-la. La conversion fonctionne dans les deux sens.

Autres améliorations

Nous terminerons cette nouvelle version de l'application *Convertisseur1* par quelques améliorations simples.

Voir le nom entier

Le nom de l'application *Convertisseur1* est trop long ; il ne s'affiche pas entier sous son logo sur l'iPhone.

Nous allons en changer pour un nom plus court, par exemple *ConvertPro*.

1 Dans la zone **Groups & Files**, sous XCode, sélectionnez la cible *Convertisseur1* dans le groupe **Targets**, cliquez du bouton droit et activez la commande **Get info** pour afficher les informations relatives à l'application *Convertisseur1*.

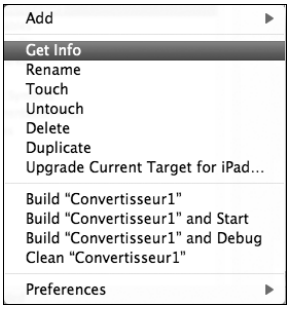


Figure 4.20 : Afficher les informations de l'application

2 Sélectionnez l'onglet **Build** de la fenêtre d'information qui est apparue. Saisissez `name` dans le champ de recherche pour limiter la liste des paramètres affichés et recherchez le paramètre **Product Name**.

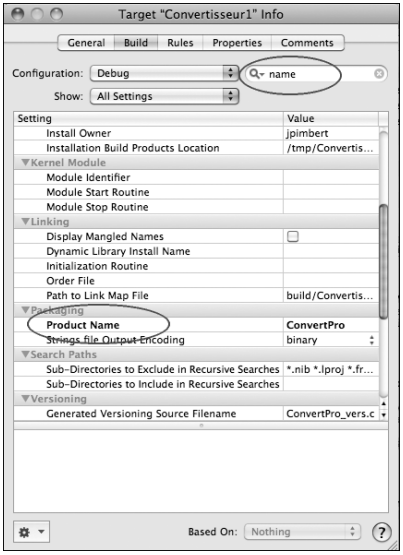


Figure 4.21 : Modifier le nom de l'application

3 Double-cliquez sur ce paramètre et saisissez le nom souhaité, par exemple `ConvertPro`.

4 Reconstruisez l'application pour vérifier que le nom s'affiche sous le logo.

Effacer le clavier

Nous souhaitons que le clavier disparaisse lorsque l'utilisateur presse la touche **Terminé** (*Done* en anglais) sur le clavier.

La frappe de cette touche provoque l'émission de l'événement **Did End On Exit** par le champ de texte. Il nous faut donc créer une nouvelle action, appelons-la `doneEditing`, dans `Convertisseur1ViewController`.

1 Modifiez la classe `Convertisseur1ViewController` pour déclarer cette nouvelle action puis liez l'événement **Did End On Exit** des deux champs de texte à cette action. N'oubliez pas d'enregistrer le fichier modifié sous XCode et de recharger les fichiers de classe sous Interface Builder.

```
- (IBAction) changeValue:(id)sender;
- (IBAction) doneEditing:(id)sender;
@end
```

Figure 4.22 : Champs de texte connectés à la nouvelle action

2 Saisissez le code de l'action. Reconstruisez l'application et testez-la.

```
- (IBAction) doneEditing:(id)sender {
    [sender resignFirstResponder];
}
```

Nous arrivons maintenant à effacer le clavier.

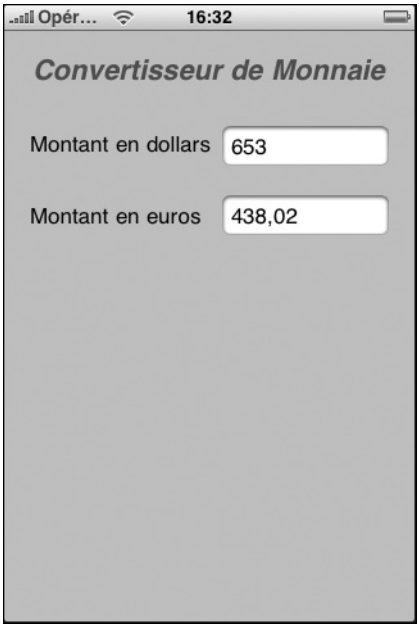


Figure 4.23 : Le clavier est escamotable

La méthode `–resignFirstResponder` est définie dans la classe `UIResponder`. Cette classe gère la chaîne des répondeurs.

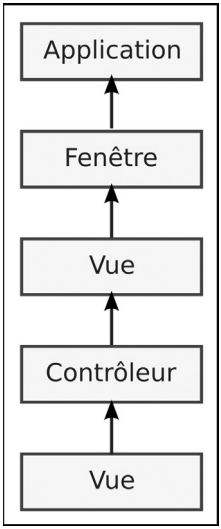


Figure 4.24 : Chaîne des répondeurs

La chaîne des répondeurs est constituée par les vues, les contrôleurs de vue, la fenêtre d'application et l'application. Le **premier répondeur-**

(*First responder*) est généralement la vue, ou le contrôle puisque UIControl hérite de UIView, actif à un moment donné.

Lorsque l'utilisateur touche le champ de texte *dollar*, ce dernier devient le premier répondeur. L'application transmet les actions au premier répondeur. C'est l'objet actif à un moment donné qui est le mieux placé pour répondre à une action. Si le premier répondeur, généralement une vue, ne sait pas traiter l'action, cette dernière est transmise au répondeur suivant dans la chaîne, etc.

La chaîne de répondeurs remonte la hiérarchie des vues à partir du premier répondeur. Une vue transmet une action à son contrôleur (si elle en possède un) avant de la transmettre à sa supervue.

Si un champ de texte reçoit le message `–resignFirstResponder`, il devient inactif et le clavier disparaît alors.

4.3. Motif MVC

Regardons la structure générale de l'application que nous avons produite.

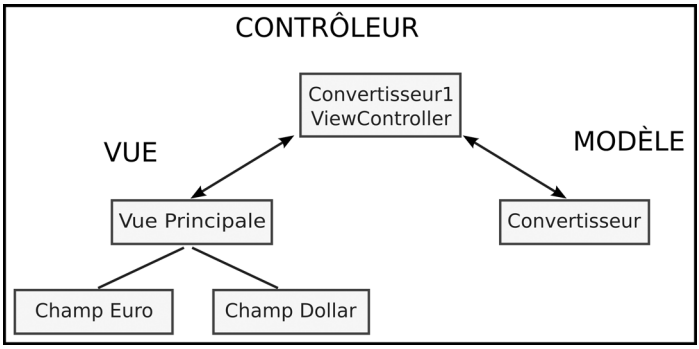


Figure 4.25 : Structure MVC de l'application Convertisseur1

Nous pouvons identifier trois parties distinctes :

- **Le Modèle** ; c'est l'ensemble des classes qui gèrent les données de l'application, on les appelle les objets métiers. Pour l'application *Convertisseur1*, il est constitué de la seule classe *Convertisseur*.
- **La Vue** ; composée de la hiérarchie des vues et de la fenêtre de l'application.
- **Les Contrôleurs** ; qui comportent les contrôleurs de vue et l'application.

Les contrôleurs connaissent les parties *Vue* et *Modèle* ; la classe `Convertisseur1ViewController` connaît les outlets *labelDollar* et *labelEuro* (*Vue*), et la propriété *Convertisseur* (*Modèle*). Il n'y a en revanche aucune communication directe entre les vues et le modèle.

Ce motif de conception est nommé **Modèle-Vue-Contrôleur** (**MVC**, *Model-View-Controller*). Il consiste à identifier le rôle de chaque classe avec l'objectif de faciliter la réutilisation des objets (*Vue* et *Modèle*) ainsi que les tests.

Les objets composant la Vue sont des briques utilisables telles quelles dans toutes les applications. Ces briques sont indépendantes des objets métiers.

Veillez à ce que les classes métiers (celles du *Modèle*) n'aient à connaître ni les vues, ni les contrôleurs. C'est la garantie de pouvoir les réutiliser dans d'autres applications.

4.4. Challenges

Nous avons suffisamment avancé maintenant pour que vous puissiez créer de petites applications simples.

Améliorer encore Convertisseur1

Notre application est déjà d'un bon niveau mais il reste encore un comportement qui peut être désagréable pour l'utilisateur. Lorsqu'on entre en mode Édition dans un champ de texte, son contenu est effacé, le contenu des deux champs n'est plus le résultat d'une conversion de l'un vers l'autre (voir Figure 4.26).

Il existe deux possibilités pour éviter ce petit problème :

- éviter que les champs soient effacés en début de saisie ;
- effacer les deux champs lorsque la saisie débute dans l'un des deux.

Votre premier challenge consiste à implémenter l'une de ces deux solutions :

- La première solution nécessite seulement la modification du fichier NIB sous Interface Builder.
- Pour la seconde solution, on utilise :
  - soit la méthode de délégué – `(void)textFieldDidBeginEditing:(UITextField *)textField` ;

Figure 4.26 : Effacement du champ en début de saisie

— soit le mécanisme **cible-action** avec l'événement **Editing Did Begin**.

REMARQUE

Risque de plantage

Il ne faut pas modifier un champ de texte par programmation pendant son édition par l'utilisateur.

Explorer les contrôles simples

Nous connaissons les contrôles de classe `UILabel` et `UITextField`. Nous avons exploré les techniques fondamentales de programmation Cocoa Touch : **cible-action**, **délégation**, **gestion de la mémoire**, **KVC** et **MVC**. Vous en savez maintenant suffisamment pour utiliser d'autres contrôles simples et développer votre propre application.

Autres contrôles simples

Les contrôles les plus simples sont regroupés dans la rubrique **Inputs & Values** de la bibliothèque d'objets d'Interface Builder.

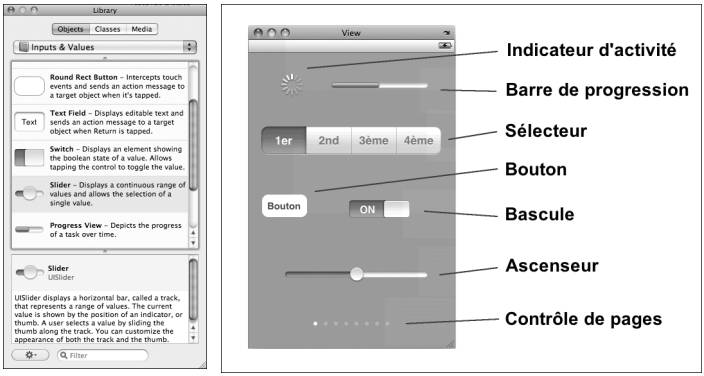


Figure 4.27 : Contrôles simples de Inputs & Values

On utilise ces contrôles de la façon suivante :

- **Label** (*Label*) ; ce contrôle statique affiche un texte. Le programme peut modifier le texte affiché par la propriété *text*.
- **Barre de progression** (*Progress View*) ; ce contrôle statique est une barre de progression. Le programme doit modifier la propriété `progress` (un nombre compris entre 0 et 1) pour visualiser la progression.
- **Indicateur d'activité** (*Activity Indicator*) ; ce contrôle statique affiche une roue qui tourne indiquant à l'utilisateur qu'une tâche est en cours. Le programme émet les messages `startAnimating` et `stopAnimating` pour démarrer et arrêter le mouvement tournant. La propriété `hidesWhenStopped` doit prendre pour valeur `YES` si l'on souhaite que l'indicateur soit masqué lorsqu'il n'est pas actif.
- **Sélecteur** (*Segmented Control*) ; ce contrôle est un bouton à plusieurs valeurs possibles, l'utilisateur sélectionne l'une de ces valeurs. Le programme utilise la propriété `numberOfSegments` pour connaître le nombre de choix possibles, et la propriété `selectedSegmentIndex` pour connaître le choix actuellement sélectionné (nombre entier à partir de 0). Les cibles-actions doivent être connectées sur l'événement **Value Changed**.
- **Bouton** (*Round Rect Button*) ; ce contrôle est un bouton simple. Les cibles-actions doivent être connectées sur l'événement **Touch Up Inside**.
- **Bascule** (*Switch*) ; ce contrôle est un bouton à deux états. Le programme utilise la propriété `on` pour connaître l'état `YES` ou `NO` de la bascule. Les cibles-actions doivent être connectées sur l'événement **Value Changed**.

- **Ascenseur** (*Slider*) ; ce contrôle est un ascenseur horizontal ou vertical. Les valeurs min et max sont précisées sous Interface Builder. Le programme utilise la propriété `value` pour connaître la position de l'ascenseur. Les cibles-actions doivent être connectées sur l'événement **Value Changed**.
- **Champ de Texte** (*TextField*) ; ce contrôle est un champ de texte. Nous l'avons abondamment expliqué. C'est le contrôle le moins facile à utiliser de cette liste ; il émet plusieurs événements et il est le seul à posséder un délégué.
- **Contrôle de pages** (*Page Control*) : ce contrôle permet à l'utilisateur de visualiser le nombre de pages et le numéro de la page en cours. Le programme indique le nombre de pages par la propriété `numberOfPages`, il utilise la propriété `currentPage` pour connaître le numéro (à partir de 0) de la page visualisée. La propriété `hidesForSinglePage` permet de masquer le contrôle s'il n'y a qu'une page. Les cibles-actions doivent être connectées sur l'événement **Value Changed**.

Tous les contrôles dynamiques héritent successivement de `NSObject`, `NSResponder`, `UIView` et `UIControl`. Les contrôles statiques n'héritent pas de `UIControl` ; un "contrôle statique" est une vue.

Le tableau résume les classes, les propriétés, les méthodes et les événements actifs pour chaque contrôle simple.

Tableau 4.2 : Utilisation des contrôles simples

Nom du contrôle	Classe d'objet	Propriétés	Méthodes	Événements
Label	`UILabel`	`text`	-	-
Progress View	`UIProgressView`	`progress`	-	-
Activity Indicator	`UIActivityIndicatorView`	`hidesWhenStopped`	`startAnimating` `stopAnimating`	-
Segmented Control	`UISegmentedControl`	`numberOfSegments` `selectedSegmentIndex`	-	**Value Changed**
Round Rect Button	`UIButton`	-	-	**Touch Up Inside**
Switch	`UISwitch`	`on`	-	**Value Changed**
Slider	`UISlider`	`value`	-	**Value Changed**

Tableau 4.2 : Utilisation des contrôles simples

Nom du contrôle	Classe d'objet	Propriétés	Méthodes	Événements
TextField	`UITextField`	`text`	Possède un délégué.	**Did End On Exit** **Editing Changed** **Editing Did Begin** **Editing Did End**

Page Control	`UIPage Control`	`numberOfPages`
		`currentPage`
		`hidesForSingle Page`
			-	**Value Changed**

Créez votre propre application

Limitez-vous à des applications sur une seule vue, nous verrons les techniques qui permettent de développer des applications multivues au chapitre suivant.

Si vous n'avez pas d'idée, vous pouvez essayer d'ajouter des ascenseurs à *Convertisseur1*. L'utilisateur pourra ainsi effectuer des conversions très rapidement, sans avoir besoin de saisir un nombre dans un champ de texte. Pour que l'application soit pratique, il faut que les trois autres contrôles changent de valeur, quel que soit le moyen utilisé pour indiquer un montant.

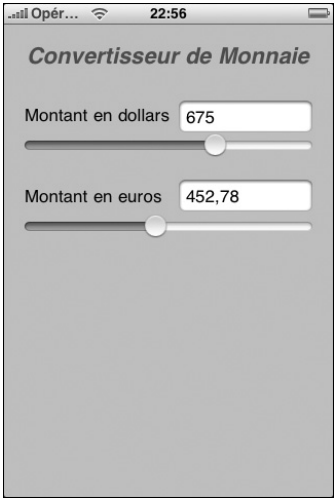


Figure 4.28 : Convertisseur1 avec des ascenseurs

Trouver plus d'informations

Les frameworks Cocoa Touch sont très riches. Nous n'avons présenté que l'utilisation la plus courante des contrôles simples. Nous

verrons d'autres techniques par la suite mais vous pouvez d'ores et déjà accéder à des informations plus détaillées dans l'abondante documentation fournie par Apple sur le site des développeurs (**http://developer.apple.com/**).

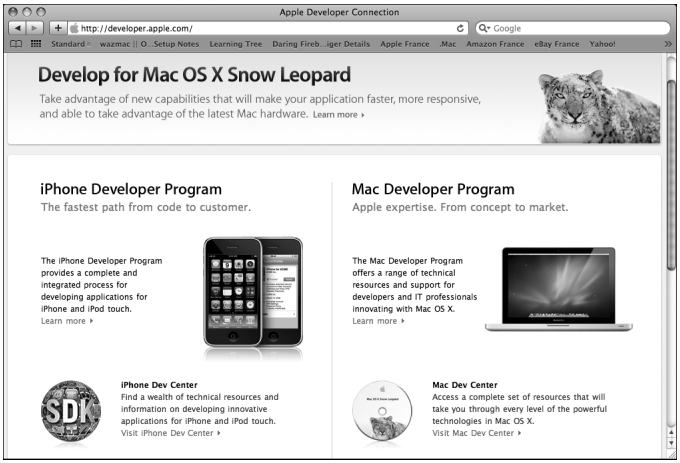


Figure 4.29 : Site des développeurs d'Apple

Sélectionnez **iPhone Dev Center**. Saisissez votre identifiant et votre mot de passe pour accéder aux ressources de développement iPhone ; documentation, exemples, vidéos, etc.

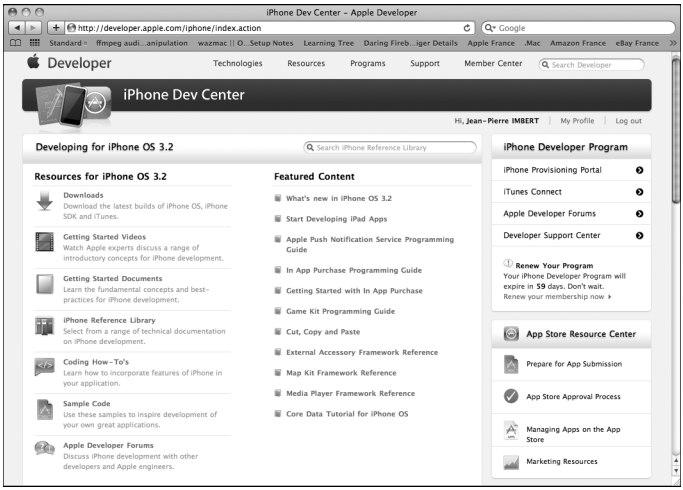


Figure 4.30 : Centre de développement iPhone

Identifiant Apple

Il est indispensable d'être inscrit comme développeur Apple pour accéder aux ressources du centre des développeurs. Vous avez déjà un identifiant si vous avez téléchargé le SDK iPhone, utilisez le même.

L'obtention d'un identifiant, le téléchargement du SDK iPhone et l'accès aux ressources du centre des développeurs sont gratuits.

4.5. Check-list

Nous connaissions déjà le mécanisme Cible-Action. Dans ce chapitre, nous avons continué notre découverte des motifs de conception fondamentaux : **Délégation**, **MVC** et **KVC**. Nous savons créer nos propres classes d'objets. Nous avons maintenant des bases solides pour aborder les techniques plus complexes.

Nous avons vu comment structurer les **fichiers NIB** et les **contrôleurs de vue** d'une application. Nous avons appris dans quelle partie du code il faut introduire le comportement spécifique désiré : **délégué d'application**, **délégués** et **contrôleurs** de vue.

Plus précisément, nous avons avancé dans notre connaissance des frameworks Cocoa Touch :

- Nous connaissons les protocoles de délégué d'application et de champ de texte.
- Nous avons utilisé les classes `NSCharacterSet` et `NSScanner`.
- Nous savons localiser les formats de nombre et les fichiers NIB.
- Nous connaissons la chaîne de répondeurs et la classe `UIResponder`.
- Nous savons utiliser tous les contrôles simples de la bibliothèque **Inputs & Values**.

Concernant le langage Objective C, nous savons maintenant :

- définir un protocole, avec les clauses `@protocol` et `@optional` ;
- utiliser la clause `@class` ;
- utiliser la méthode `–respondsToSelector` et la fonction `SEL` ;
- utiliser le type `BOOL` et les constantes `NSString`.

Nous avons découvert quelques commandes supplémentaires dans les outils du SDK :

- nettoyer les cibles sous XCode après avoir supprimé ou restructuré des ressources ;
- changer le nom de l'application sous XCode ;
- recharger les fichiers de classe sous Interface Builder pour prendre en compte les modifications effectuées sous XCode ;
- changer la langue et les paramètres régionaux du simulateur d'iPhone pour tester nos applications internationales.

APPLICATIONS MULTIVUES

Dans ce chapitre, nous créerons une version 2 de notre convertisseur, avant de délaisser momentanément les dollars et les euros dans les chapitres suivants qui nous porteront vers d'autres types d'applications.

Notre objectif est de comprendre le fonctionnement général des applications multivues : celles dans lesquelles l'utilisateur peut changer de vue principale.

5.1. Application de type utilitaire

Le taux de change des devises varie tous les jours à midi (le fixing). Si notre utilisateur est pointilleux sur les centimes ou s'il est très riche et veut convertir de grosses sommes, il voudra pouvoir modifier le taux de conversion utilisé par notre convertisseur. Nous allons donc lui proposer cette fonctionnalité.

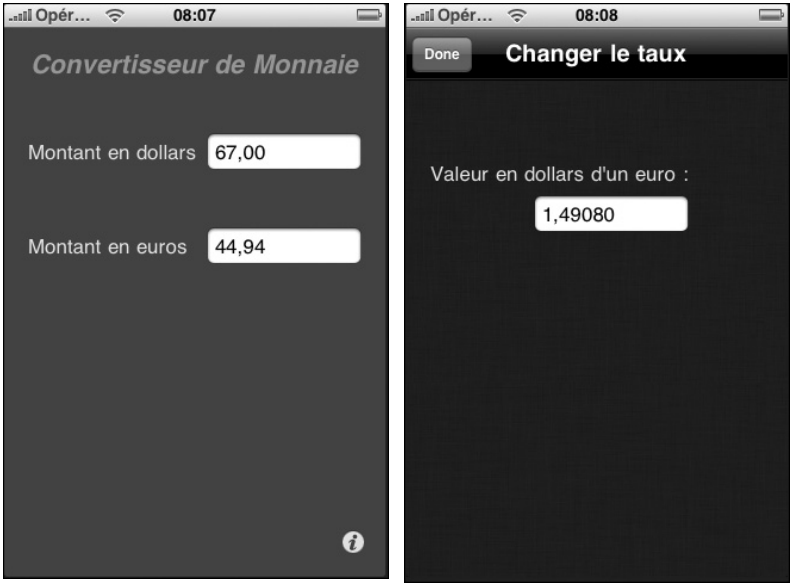


Figure 5.1 : Application Convertisseur2

Comprendre le fonctionnement d'un utilitaire

Créer le projet Convertisseur2

1 Ouvrez XCode et créez un nouveau projet. Nous choisirons le modèle *Utility Application* (*Utilitaire*). Laissez la case *Use Core Data for storage* décochée. *Nommez ce projet Convertisseur2.*

Figure 5.2 : Créer un projet de type Utility Application

2 Ajoutez un logo et nommez l'application `ConvertPro`. Vous savez comment faire maintenant. N'oubliez pas d'activer la case à cocher *Copy items into destination group's folder (if needed)* pour que le logo soit copié dans le dossier du projet.

3 Construisez et lancez l'application (⌘+R) pour voir ce que cela donne.

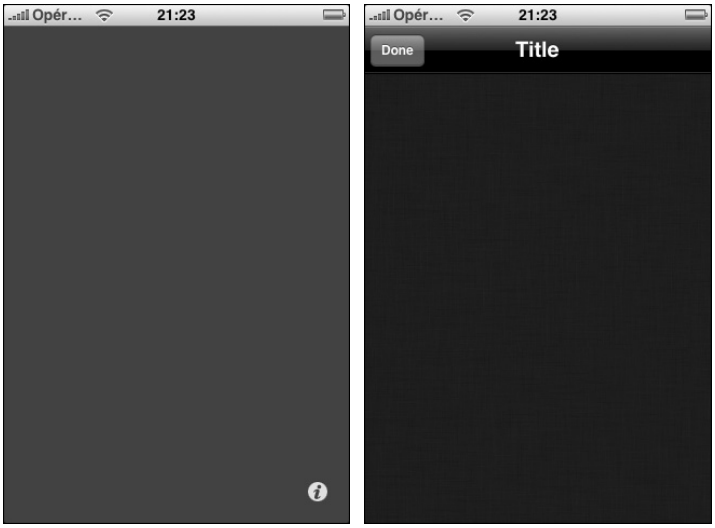


Figure 5.3 : Un utilitaire à l'œuvre

Notre application se comporte comme si la Vue était à double face :

- Au recto, un petit bouton est affiché en bas à droite. Si on le touche, la vue semble se retourner.
- Au verso, une barre de titre est affichée en haut, avec un bouton **Done** qui nous permet de revenir au recto.

Examinons le détail du fonctionnement de ce squelette d'application.

Structure de base d'un utilitaire

Regardons la liste des fichiers créés par XCode. Nous avons d'abord des classes Objective-C :

- *MainViewController* ;
- *FlipsideViewController* ;
- *Convertisseur2AppDelegate*.

Puis les fichiers *NIB* (les autres fichiers sont les mêmes que pour *Convertisseur1*) :

- *MainWindow.xib* ;
- *MainView.xib* ;
- *FlipsideView.xib*.

Le navigateur de Classes

Sous XCode, la commande *Class Browser* du menu **Project** (Maj+⌘+C) affiche une fenêtre qui vous permet de visualiser la hiérarchie des classes, la liste des méthodes de chaque classe et leur code source. Vous pouvez filtrer les classes définies dans le projet ou voir l'ensemble des classes. On obtient la documentation des classes des frameworks en cliquant sur l'icône en forme de livre à côté du nom de la classe.

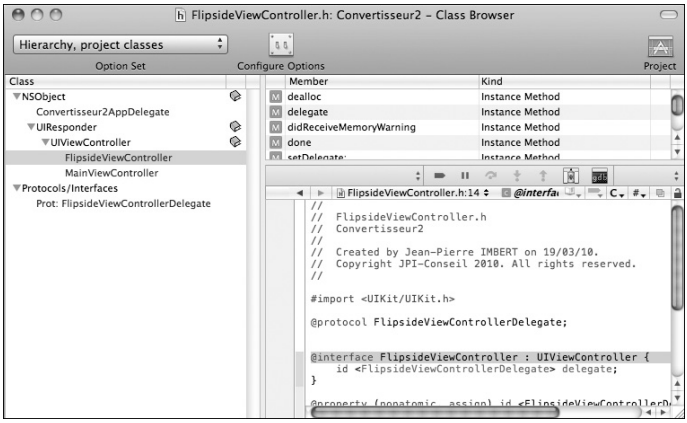


Figure 5.4 : Navigateur de classes

Fonctionnement de base d'un utilitaire

Examinez le code source des classes créées par XCode et les fichiers *NIB* sous Interface Builder pour identifier les liens. Représentons la structure de l'application obtenue.

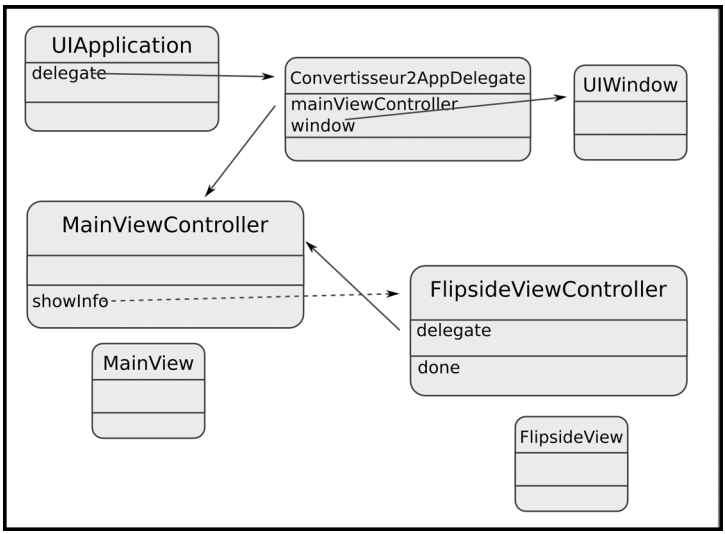


Figure 5.5 : Structure d'une application de type Utilitaire

Nous reconnaissons une structure d'application classique : l'application a un délégué `Convertisseur2AppDelegate` qui possède une fenêtre et un contrôleur de vue principal `MainViewController`. Ce dernier possède à son tour un fichier NIB *MainView.xib*.

La nouveauté avec ce type d'application, c'est un contrôleur `FlipsideViewController` qui possède un fichier NIB *FlipsideView.xib*. Nous verrons comment cela fonctionne mais auparavant, nous allons nous pencher sur une autre petite différence.

Attacher un contrôleur à un fichier NIB

Souvenez-vous comment étaient liés le contrôleur de vue principal et son fichier NIB ; dans le ficher NIB *MainWindow.xib* du délégué de l'application (voir Figure 5.6).

Dans notre application *Convertisseur2*, une autre méthode est utilisée. Regardez la méthode `–applicationDidFinishLaunching:` du délégué d'application, en particulier les lignes qui ne se trouvent pas dans le code du délégué de l'application *Convertisseur1* :

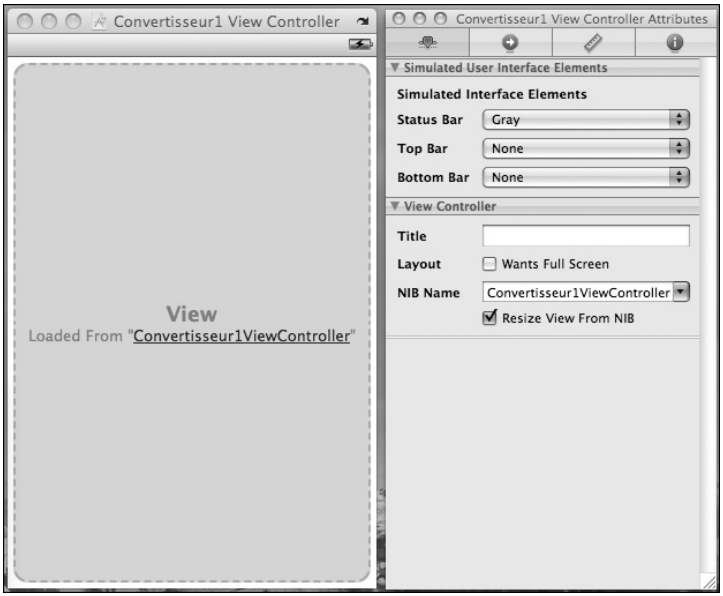


Figure 5.6 : Contrôleur de vue dans le fichier NIB du délégué d'application

```
- (void)applicationDidFinishLaunching:
                                (UIApplication *)application {
    MainViewController *aController =
                    [[MainViewController alloc]
                    initWithNibName:@"MainView" bundle:nil];
    self.mainViewController = aController;
    [aController release];
    mainViewController.view.frame =
                    [UIScreen mainScreen].applicationFrame;
    [window addSubview:[mainViewController view]];
    [window makeKeyAndVisible];
}
```

La méthode `-initWithNibName:bundle:` permet d'initialiser un contrôleur de vue en donnant le nom d'un fichier NIB (sans l'extension *.xib* ou *.nib*) et un paquet (*Bundle*) dans lequel se trouve ce fichier NIB. Par défaut (`nil`), le fichier NIB est recherché dans le paquet de l'application courante.

Paquet (*Bundle*)
Un paquet est un dossier contenant du code exécutable, des fichiers NIB et des ressources diverses.

Le cadre (*frame*) de la vue principale (`mainViewController.view.frame`) est ensuite défini comme étant le cadre dédié à l'application sur l'écran (`[UIScreen mainScreen].applicationFrame`). Par défaut, une barre d'état est affichée en haut de l'écran, la zone restante est dédiée à l'application.

Cadre (*frame*)

Le cadre d'une vue est le rectangle dans lequel cette vue est affichée dans la vue ou la fenêtre qui la contient. L'origine et la taille du rectangle sont exprimées en pixels relativement au cadre de la vue qui contient.

Cadre dédié à l'application

L'écran a une taille de 480 x 320 pixels sur un iPhone et un iPod Touch, et de 1 024 x 768 pixels sur un iPad. La hauteur de la barre d'état est de 20 pixels. La zone dédiée à l'application est la totalité de l'écran excepté la barre d'état.

Il est recommandé d'utiliser la classe `UIScreen` plutôt que d'écrire la taille de la vue principale "en dur" dans son code. Cela facilitera l'adaptation de votre application sur différents appareils, voire de futurs appareils dont la taille de l'écran serait différente.

Nous venons de voir que l'on peut définir et initialiser un contrôleur de vue :

- soit dans un fichier NIB comme dans *Convertisseur1* ;
- soit par programmation comme dans *Convertisseur2*.

Cette alternative est toujours vraie : il n'y a rien que l'on puisse faire dans un fichier NIB et que l'on ne puisse obtenir par programmation.

Les deux méthodes donnent le même résultat. Le fichier NIB est sans doute plus facile à réaliser pour le développeur ; quelques clics au lieu de plusieurs lignes de code. La programmation est beaucoup plus souple et plus puissante car elle permet d'adapter la vue et son contrôleur au contexte en cours, alors que la définition dans un fichier NIB est faite a priori, avant l'exécution de l'application, elle est donc figée.

Animer le changement de vue

Intéressons-nous maintenant à la façon dont le changement de vue s'effectue. Lorsque l'utilisateur touche le bouton de la vue principale, le message `showInfo` est transmis au contrôleur de vue.

```
- (IBAction)showInfo {
    FlipsideViewController *controller =
                [[FlipsideViewController alloc]
                initWithNibName:@"FlipsideView" bundle:nil];
    controller.delegate = self;
    controller.modalTransitionStyle =
                        UIModalTransitionStyleFlipHorizontal;
    [self presentModalViewController:controller
                            animated:YES];
    [controller release];
}
```

Nous connaissons déjà la méthode `–initWithNibName:bundle:` qui nous permet ici de créer une instance de `FlipsideViewController` et de l'attacher au fichier NIB *FlipsideView.xib*.

Ensuite, l'instance de contrôleur de la vue principale est définie comme déléguée du contrôleur nouvellement créé. Cela servira pour revenir à la vue principale, nous en examinerons le mécanisme plus loin.

Le changement de vue est déclenché par le message `–presentModalViewController:animated:`. La propriété `modalTransitionStyle` d'un contrôleur de vue permet de spécifier le type d'animation souhaitée. Testez l'application en utilisant successivement les 4 styles proposés par Apple.

Tableau 5.1 : Type énuméré UIModalTransitionStyle

Constante	Signification
`UIModalTransitionStyleCoverVertical`	Style par défaut, la vue modale recouvre la vue courante en glissant vers le haut.
`UIModalTransitionStyleFlipHorizontal`	La vue modale s'affiche comme si elle était au verso de la vue courante.
`UIModalTransitionStyleCrossDissolve`	La vue courante se dissout lors de l'affichage de la vue modale.
`UIModalTransitionStylePartialCurl`	La vue courante est "relevée" et reste apparente sur un angle de la vue modale. Disponible uniquement à partir de la version 3.2.

Testez également les différents types de boutons proposés dans l'inspecteur des attributs (⌘+1) sous Interface Builder.

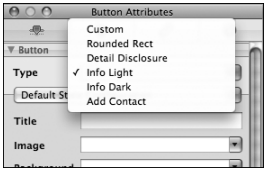


Figure 5.7 : Différents types de bouton sous Interface Builder

Vue modale

Une vue est dite modale lorsqu'elle remplace temporairement une autre vue. Cette autre vue sera à nouveau présentée à l'utilisateur lorsque la vue modale disparaîtra.

Une vue modale peut à son tour passer le contrôle à une autre vue modale.

L'instruction `[controller release];` mérite une petite explication complémentaire. Il faut respecter la règle de gestion de la mémoire. L'instance `controller` de la classe `FlipsideViewController` vient d'être créée avec `+alloc`, il faut donc s'occuper de sa libération. Mais si on la libère tout de suite à la fin de la méthode `showInfo`, ne va-t-elle pas être détruite ? La vue *Flipside* va-t-elle s'afficher correctement ? En fait, tout va fonctionner correctement car le message `–presentModalViewController:animated:` retient le paramètre `controller` dans la propriété `modalViewController` du contrôleur de la vue principale. La libération de l'instance `controller` ne va donc pas provoquer sa destruction immédiate.

Revenir à la vue principale

Examinons maintenant comment le contrôleur de la vue modale revient à la vue principale. Lorsque l'utilisateur touche le bouton **Done** le message `–done` est envoyé au propriétaire du fichier NIB ; vérifiez la cible-ation du bouton en ouvrant *FlipsideView.xib* sous InterfaceBuilder.

Le contrôleur de vue propriétaire de ce fichier NIB (`FlipsideViewController`) définit la méthode `–done`.

```
- (IBAction)done {
   [self.delegate flipsideViewControllerDidFinish:self];
}
```

Souvenez-vous, le délégué de `FlipsideViewController` est le contrôleur de la vue principale ; il est affecté dans sa méthode `–showInfo`. Regardez le code de la méthode `–flipsideViewControllerDidFinish:` dans le fichier *MainViewController.m* :

```
- (void)flipsideViewControllerDidFinish:
                    (FlipsideViewController *)controller {
   [self dismissModalViewControllerAnimated:YES];
}
```

Lorsque l'utilisateur touche le bouton **Done** pour refermer la vue modale, le contrôleur de la vue modale informe le contrôleur de la vue principale. Ce dernier doit désactiver la vue modale puisque c'est lui qui l'a activée.

Activer une vue modale

Résumons le principe de gestion des vues modales :

- La vue modale est définie dans un fichier NIB et attachée à un contrôleur de vue spécifique.
- L'activation de la vue modale est déclenchée par l'émission du message `–presentModalViewController:animated:` sur le contrôleur de la vue principale.
- Le retour à la vue principale est déclenché par l'émission du message `–dismissModalViewControllerAnimated:` sur le contrôleur de la vue principale.

Utilisation de la délégation

Vous avez sans doute remarqué le mécanisme de délégation mis en œuvre :

- Un protocole `FlipsideViewControllerDelegate` est défini dans la classe `FlipsideViewController`.
- Ce protocole déclare la méthode `–flipsideViewControllerDidFinish:` (vérifiez dans le fichier *FlipSideViewController.h*).
- Le contrôleur de la vue principale `MainViewController` adopte le protocole `FlipsideViewControllerDelegate`.
- Le contrôleur de la vue principale se définit comme délégué du contrôleur de la vue modale à la création de ce dernier.
- Lorsqu'il souhaite que la vue modale soit fermée, son contrôleur de vue émet le message défini dans le protocole vers son délégué.
- Lorsque le contrôleur de la vue principale reçoit ce message, il désactive la vue modale.

Ce fonctionnement relativement complexe pourrait être remplacé par le code suivant de la méthode `–done` de la classe `FlipsideViewController` :

```
- (IBAction)done {
    [self.parentViewController
                    dismissModalViewControllerAnimated:YES];
}
```

Le code sans utiliser la délégation serait donc plus simple. La délégation est tout de même une bonne pratique de programmation ; le contrôleur de vue principale déclenche la vue modale puis reprend le contrôle lorsque l'utilisateur souhaite revenir à la vue principale. Celui qui déclenche une action en récupère le résultat.

5.2. Application Convertisseur2

Composer la vue principale

Nous souhaitons que la vue principale de l'application *Convertisseur2* soit analogue à celle de *Convertisseur1*. Nous allons donc récupérer les éléments dans le fichier NIB.

1 Ouvrez simultanément les deux fichiers NIB sous Interface Builder :

- fichier *Convertisseur1ViewController.xib* du projet *Converisseur1* ;
- fichier *MainView.xib* du projet *Convertisseur2*.

2 Dans le premier fichier, sélectionnez les éléments à récupérer (utilisez la touche Maj pour étendre la sélection) :

- le titre "`Convertisseur de Monnaie`" ;
- les labels "`Montant en dollars`" et "`Montant en euros`" ;
- les deux champs de texte.

3 Copiez la sélection du premier fichier pour la coller dans le second fichier. Modifiez les couleurs des labels pour améliorer la visibilité. Vous pouvez également changer la couleur de fond de la vue principale si vous préférez.

Figure 5.8 : Composition de la vue principale

4 Enregistrez le fichier *MainView.xib* et fermez le fichier *Convertisseur1ViewController.xib*.

Paramétrer le taux de conversion

Composition de la vue modale

1 Ouvrez le fichier *FlipsideView.xib* du projet *Convertisseur2* sous Interface Builder pour y ajouter un label et un champ de texte.

2 Modifiez le titre de la barre de navigation (en haut de la vue).

Figure 5.9 : Composition de la vue modale

Adapter la classe Convertisseur

Ajoutez les fichiers *Convertisseur.h* et *Convertisseur.m* du projet *Convertisseur1* au projet *Convertisseur2*. N'oubliez pas de copier ces fichiers dans le dossier du projet ; nous allons les modifier.

La classe `Convertisseur` doit prendre en compte le fait que l'utilisateur peut changer le taux de conversion à tout moment. La propriété `dollarsPourUnEuro` est donc modifiable durant l'exécution de l'application mais les propriétés `dollar` et `euro` ne seront plus le résultat d'une conversion de l'une vers l'autre. Il faudrait recalculer l'une des deux valeurs, ou plus simplement les annuler lorsque la propriété `dollarsPourUnEuro` est modifiée.

Ajoutez la méthode `-setDollarsPourUnEuro:` dans le fichier *Convertisseur.m* :

```
-(void) setDollarsPourUnEuro:(float)newValue {
   if (dollarsPourUnEuro != newValue) {
      dollarsPourUnEuro = newValue;
      self.euro = 0.;
   }
}
```

Remarquez l'emploi de l'instruction `self.euro=0.;`. Elle provoque l'émission du message `-setEuro:` qui modifie simultanément les propriétés `euro` et `dollar`.

On remet à zéro les propriétés `euro` et `dollar` uniquement si le nouveau taux de conversion est différent de l'ancien.

Factoriser le délégué de champ de texte

Dans l'application *Convertisseur1*, le contrôleur de la vue principale est aussi le délégué des champs de texte ; il est chargé de vérifier que l'utilisateur ne saisit que des nombres.

Nous avons besoin du même mécanisme pour vérifier que le taux de conversion saisi par l'utilisateur est un nombre. Nous allons donc créer un objet spécifique pour cette délégation, plutôt que de confier cette mission aux deux contrôleurs de vue.

Objet délégué réutilisable

Il nous faut un objet qui implémente le protocole `UITextFieldDelegate` et définisse la méthode `-textField:shouldChangeCharactersInRange:replacementString:`. Dans la classe `Convertisseur1ViewController`, cette méthode était utilisée pour définir les valeurs en euros ou en dollars suivant le cas. Notre nouvel objet délégué devant être réutilisable dans d'autres contextes, il n'est pas de sa responsabilité de modifier des propriétés d'autres objets.

Appelons cette classe `NumericFieldDelegate`. Sa responsabilité sera de vérifier que le champ de texte contient uniquement des nombres et de retenir ce nombre pour éviter que la conversion soit réalisée plusieurs fois.

1 Créez les fichiers *NumericFieldDelegate.h* et *NumericFieldDelegate.m* dans le projet *Convertisseur2*.

2 Déclarez l'utilisation du protocole `UITextFieldDelegate` et la propriété `value` dans l'interface de la classe :

```
#import <Foundation/Foundation.h>
@interface NumericFieldDelegate : NSObject
                                  <UITextFieldDelegate> {
   float value;
}
@property (nonatomic,assign) float value;
@end
```

3 Modifiez le fichier *NumericFieldDelegate.m* :

```
#import "NumericFieldDelegate.h"
@implementation NumericFieldDelegate
@synthesize value;
- (BOOL)textField:(UITextField *)textField
        shouldChangeCharactersInRange:(NSRange)range
                    replacementString:(NSString *)string {
   NSString *resultingString = [textField.text
                stringByReplacingCharactersInRange:range
                                        withString:string];
   NSScanner *scan = [NSScanner
             localizedScannerWithString:resultingString];
   [scan scanFloat:&value];
   return [scan isAtEnd];
}
@end
```

Le travail de conversion d'une chaîne de caractères en valeur numérique effectué dans la méthode `–setValueForKey:WithString:` de la classe `Convertisseur` est maintenant réalisé par la classe `NumericFieldDelegate`. Vous pouvez supprimer cette méthode de la classe `Convertisseur`, elle ne nous servira plus.

Connecter le délégué dans le fichier NIB

Nous allons effectuer chaque connexion entre un champ de texte et son délégué dans les fichiers NIB. Nous créerons même les objets délégués dans les fichiers NIB.

1 Ouvrez le fichier *MainView.xib* et faites glisser un objet de type `NSObject` dans la fenêtre du fichier NIB (attention : pas dans la fenêtre de la vue principale) (voir Figure 5.10).

2 Sélectionnez l'objet nouvellement créé et affichez l'inspecteur d'identité (⌘+4) pour définir sa classe (`NumericFieldDelegate`) et son nom (`Dollar Field Delegate`) (voir Figure 5.11).

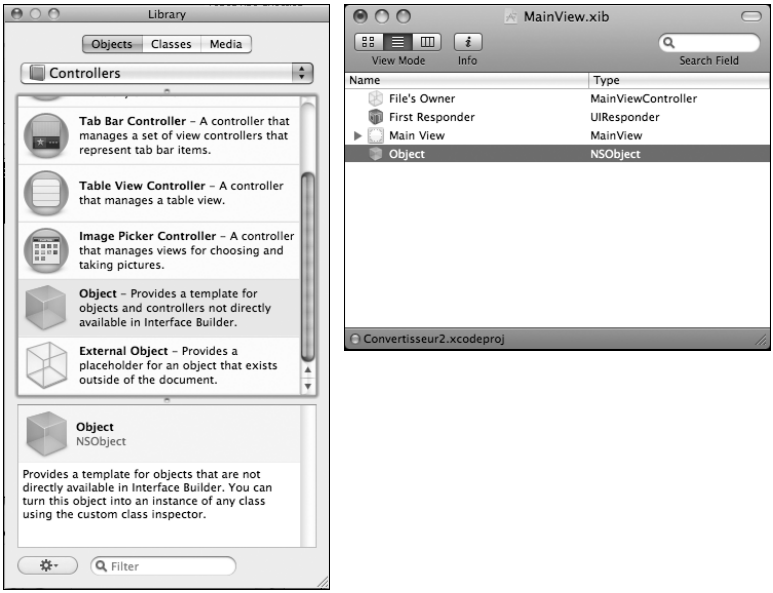


Figure 5.10 : Créer un objet dans MainWindow.xib

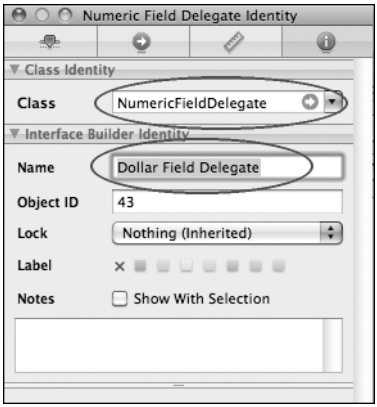


Figure 5.11 : Définition du type d'objet

3 Connectez l'outlet *delegate* du champ de texte destiné à contenir la valeur en dollars à l'objet *Dollar Field Delegate*. Utilisez l'inspecteur de connexions (⌘+2) (voir Figure 5.12).

4 Procédez de même avec les deux autres champs de texte de l'application :

— `Euro Field Delegate` dans *MainView.xib* ;

— `Rate Field Delegate` dans *FlipsideView.xib*.

5 Enregistrez les deux fichiers NIB.

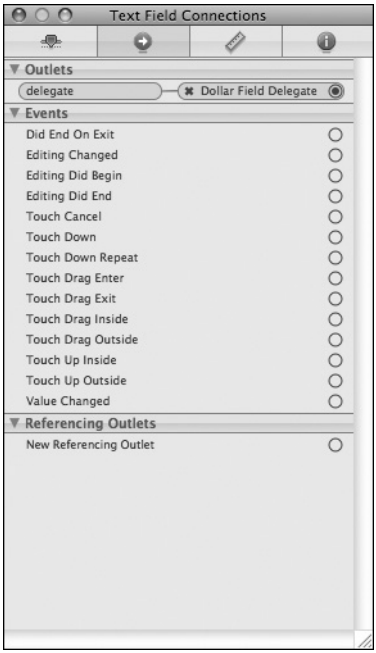


Figure 5.12 : Connexion du champ de texte à son délégué

Finaliser les contrôleurs de vue

Contrôleur de la vue principale

Déclaration

Déclarez les outlets et les actions dans le fichier *MainViewController.h* :

```
#import "FlipsideViewController.h"
#import "Convertisseur.h"
#import "NumericFieldDelegate.h"
@interface MainViewController : UIViewController
                        <FlipsideViewControllerDelegate> {
    IBOutlet UITextField *    dollarField;
    IBOutlet NumericFieldDelegate * dollarFieldDelegate;
    IBOutlet UITextField *    euroField;
    IBOutlet NumericFieldDelegate * euroFieldDelegate;
    IBOutlet Convertisseur *    convertisseur;
}
@property (nonatomic,retain) UITextField *dollarField;
@property (nonatomic,retain)
                NumericFieldDelegate *dollarFieldDelegate;
@property (nonatomic,retain) UITextField *euroField;
@property (nonatomic,retain)
                  NumericFieldDelegate *euroFieldDelegate;
```

```
@property (nonatomic,retain) Convertisseur *convertisseur;
- (IBAction) changeValue:(id)sender;
- (IBAction) beginEditing:(id)sender;
- (IBAction) doneEditing:(id)sender;
- (IBAction)showInfo;
@end
```

Nous déclarons la propriété `convertisseur` comme un *outlet*. Cela nous permettra d'établir la connexion sous Interface Builder plutôt qu'en modifiant le code de la classe `MainViewController`.

Propriété pour les délégués

Les délégués `dollarFieldDelegate` et `euroFieldDelegate` sont déjà connectés aux champs de texte correspondants par leur propriété `delegate` mais cette propriété est définie avec l'attribut `assign` au lieu de `retain`. Pour éviter les problèmes de gestion de mémoire, il faut que ces délégués soient définis comme des propriétés avec l'attribut `retain` dans un autre objet ; c'est la raison pour laquelle nous les définissons dans le contrôleur de la vue principale.

Définition

1 Définissez les accesseurs de propriétés et les méthodes spécifiques dans le fichier *MainViewController.m* :

```
@implementation MainViewController
@synthesize dollarField;
@synthesize dollarFieldDelegate;
@synthesize euroField;
@synthesize euroFieldDelegate;
@synthesize convertisseur;
- (IBAction) changeValue:sender {
   if (sender==dollarField) {
    self.convertisseur.dollar = dollarFieldDelegate.value;
    euroField.text = [NSString localizedStringWithFormat:
                        @"%.2f",self.convertisseur.euro];
   } else {
      self.convertisseur.euro = euroFieldDelegate.value;
      dollarField.text = [NSString
                      localizedStringWithFormat:
                       @"%.2f",self.convertisseur.dollar];
   }
}
- (IBAction) beginEditing:(id)sender {
   if (sender==dollarField) {
      euroField.text = @"";
   } else {
      dollarField.text = @"";
   }
```

```
}
- (IBAction) doneEditing:(id)sender {
    [sender resignFirstResponder];
}
```

2 Libérez les outlets du contrôleur de vue :

```
- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    self.dollarField = nil;
    self.dollarFieldDelegate = nil;
    self.euroField = nil;
    self.euroFieldDelegate = nil;
    self.convertisseur = nil;
}
- (void)dealloc {
    [self viewDidUnload];
    [super dealloc];
}
```

Connexions

1 Ouvrez le fichier *MainView.xib* sous Interface Builder. Ajoutez-y un objet que vous définissez de type Convertisseur.

2 Établissez les connexions du contrôleur de la vue principale : champs de texte, délégués de champ de texte, convertisseur et actions.

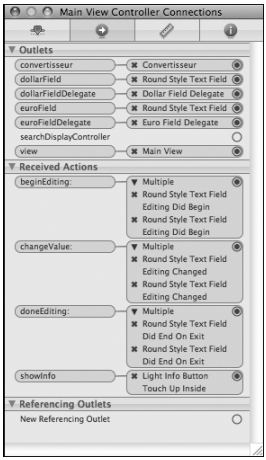


Figure 5.13 : Connexions du contrôleur de la vue principale

3 Construisez l'application et testez la vue principale. Vous devez retrouver le comportement de *Convertisseur1*.

Il nous reste à faire fonctionner la modification du taux de conversion dans la vue modale.

Contrôleur de la vue modale

Nous allons définir le contrôleur de la vue modale *FlipsideViewController* selon le même principe que le contrôleur de la vue principale :

- un outlet pour le champ de texte ;
- un outlet pour le délégué du champ de texte ;
- une propriété `rate` pour communiquer avec le contrôleur de la vue principale ;
- l'action `–doneEditing:`.

L'action `–changeValue:` ne sera pas utile pour ce contrôleur ; nous n'avons pas de mise à jour à faire sur la vue pendant l'édition. L'action `–beginEditing:` sera inutile également.

Déclaration

Effectuez les déclarations dans le fichier *FlipsideViewController.h* :

```
@protocol FlipsideViewControllerDelegate;
#import "NumericFieldDelegate.h"
@interface FlipsideViewController : UIViewController {
    id <FlipsideViewControllerDelegate> delegate;
    IBOutlet UITextField *    rateField;
    IBOutlet NumericFieldDelegate *    rateFieldDelegate;
    float    rate;
}
@property (nonatomic, assign)
            id <FlipsideViewControllerDelegate> delegate;
@property (nonatomic, retain) UITextField *rateField;
@property (nonatomic, retain)
                  NumericFieldDelegate *rateFieldDelegate;
@property (nonatomic, assign) float rate;
- (IBAction)doneEditing:(id)sender;
- (IBAction)done;
@end

@protocol FlipsideViewControllerDelegate
- (void)flipsideViewControllerDidFinish:
                      (FlipsideViewController *)controller;
@end
```

Définition

1 Modifiez le fichier *FlipsideViewController.m* pour définir les accesseurs et l'action :

```
@synthesize delegate;
@synthesize rateField;
@synthesize rateFieldDelegate;
```

```
@synthesize rate;

- (IBAction) doneEditing:(id)sender {
    [sender resignFirstResponder];
}
```

Nous souhaitons que le champ de texte `rateField` soit initialisé avec la valeur de la propriété `rate` lorsque la vue s'affiche.

2 Modifiez la méthode `–viewDidLoad` :

```
- (void)viewDidLoad {
    [super viewDidLoad];
    rateField.text = [NSString localizedStringWithFormat:
                                          @"%.5f",self.rate];
    self.view.backgroundColor =
                    [UIColor viewFlipsideBackgroundColor];
}
```

Inversement, la propriété `rate` doit prendre la valeur du champ de texte lorsque la vue est refermée par l'utilisateur.

3 Modifiez la méthode `–done` :

```
- (IBAction)done {
    if (self.rateFieldDelegate.isModified) self.rate =
                                self.rateFieldDelegate.value;
    [self.delegate flipsideViewControllerDidFinish:self];
}
```

Nous avons besoin de savoir si le contenu du champ de texte a été modifié par l'utilisateur ; si ce n'est pas le cas, la propriété `value` du délégué du champ de texte est nulle. Il faudra penser à définir une propriété `isModified` dans notre classe `NumericFieldDelegate`.

4 Libérez les outlets du contrôleur de la vue modale :

```
- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    self.rateField = nil;
    self.rateFieldDelegate = nil;
}
- (void)dealloc {
    [self viewDidLoad];
    [super dealloc];
}
```

Propriété isModified

1 Ajoutez la propriété `modified` dans le fichier *NumericFieldDelegate.h*, en spécifiant qu'elle est en lecture seule et que son accesseur est `isModified` :

```
@interface NumericFieldDelegate : NSObject
                                  <UITextFieldDelegate> {
   float    value;
   BOOL     modified;
}
@property (nonatomic,assign) float value;
@property (nonatomic,readonly,getter=isModified)
                                                BOOL modified;
@end
```

La propriété doit prendre la valeur `NO` à la création de chaque instance et la valeur `YES` à chaque modification.

2 Ajoutez une méthode `-init` dans le fichier *NumericFieldDelegate.m* :

```
- (id) init {
   if (self == [super init]) {
      modified = NO ;
   }
   return self;
}
```

3 Modifiez la méthode `-textField:shouldChangeCharactersInRange:replacementString:` dans le même fichier :

```
   NSString *resultingString = [textField.text
               stringByReplacingCharactersInRange:range
                                       withString:string];
   NSScanner *scan = [NSScanner
             localizedScannerWithString:resultingString];
   [scan scanFloat:&value];
   modified = YES;
   return [scan isAtEnd];
```

Connexions

1 Ouvrez le fichier *FlipsideView.xib* sous Interface Builder.

2 Établissez les connexions du contrôleur de la vue modale : champ de texte, délégués de champ de texte et actions (voir Figure 5.14).

3 Décochez la case *Clear When Editing Begin* dans les attributs du champ de texte (⌘+1) afin d'empêcher que le champ de texte ne soit remis à zéro au début de l'édition (voir Figure 5.15).

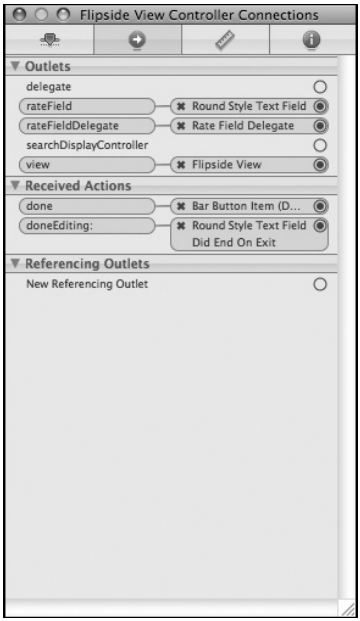


Figure 5.14 : Connexions du contrôleur de la vue modale

Figure 5.15 : Décochez la case Clear When Editing Begin

4 Construisez l'application et testez la vue modale. Elle semble fonctionner mais le taux de conversion saisi par l'utilisateur n'est pas pris en compte dans la vue principale.

Communiquer entre les deux contrôleurs

Le contrôleur de la vue modale étant piloté par le contrôleur de la vue principale, il revient logiquement à ce dernier d'établir la communication :

- La propriété `rate` doit être initialisée à la création du contrôleur de la vue modale.
- La valeur de cette propriété doit être récupérée à la fermeture de la vue modale.

Communiquer le taux de conversion

1 Modifiez la méthode `-showInfo` dans le fichier *MainViewController.m* pour y initialiser la propriété `rate` :

```
- (IBAction)showInfo {
    FlipsideViewController *controller =
              [[FlipsideViewController alloc]
              initWithNibName:@"FlipsideView" bundle:nil];
    controller.delegate = self;
    controller.rate = self.convertisseur.dollarsPourUnEuro;
    controller.modalTransitionStyle =
                    UIModalTransitionStyleFlipHorizontal;
    [self presentModalViewController:controller
                            animated:YES];
    [controller release];
}
```

2 Modifiez également la méthode `-flipsideViewControllerDidFinish:` :

```
- (void)flipsideViewControllerDidFinish:
                  (FlipsideViewController *)controller {
    self.convertisseur.dollarsPourUnEuro = controller.rate;
    [self dismissModalViewControllerAnimated:YES];
}
```

3 Construisez et testez l'application *Convertisseur2*. Son fonctionnement devrait être satisfaisant.

Éviter le blocage du clavier

Peut-être vous êtes-vous aperçu que parfois, le clavier ne disparaît pas lorsque l'utilisateur touche le bouton **Terminé** (*Done*). Ce défaut se manifeste lorsque le curseur de saisie n'apparaît pas après le dernier caractère dans le champ de texte actif.

Figure 5.16 : blocage du clavier

On peut demander au champ de texte d'adopter le même comportement quelle que soit la position du curseur. Vous savez déjà comment on peut modifier le comportement d'un champ de texte ; il faut agir sur son délégué. Ajoutez la méthode `–textFieldShouldReturn:` dans le fichier *NumericFieldDelegate.m* :

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField{
    return YES;
}
```

Lorsque cette méthode retourne la valeur `YES`, une pression sur la touche **Terminé** et interprétée comme si le curseur était après le dernier caractère.

Reconstruisez l'application et vérifiez que le défaut est corrigé.

Garder des montants cohérents

Lorsque le taux de conversion est modifié, il faudrait effacer le contenu des champs de texte de la vue principale. Actuellement, l'utilisateur voit deux montants qui sont le résultat de la conversion en utilisant l'ancien taux. Le plus simple est d'afficher le contenu de l'objet Convertisseur au retour de la fenêtre modale ; nous savons que ses propriétés sont toujours cohérentes.

Modifier le code

Modifiez la méthode `-flipsideViewControllerDidFinish:` dans le fichier *MainViewController.m* :

```
- (void)flipsideViewControllerDidFinish:
                    (FlipsideViewController *)controller {
   self.convertisseur.dollarsPourUnEuro = controller.rate;
   euroField.text = [NSString localizedStringWithFormat:
                          @"%.2f",self.convertisseur.euro];
   dollarField.text = [NSString localizedStringWithFormat:
                         @"%.2f",self.convertisseur.dollar];
   [self dismissModalViewControllerAnimated:YES];
}
```

Factoriser

C'est la quatrième fois que nous écrivons une instruction contenant `[NSString localizedStringWithFormat: @"%.2f",xxx]`. Il est temps de mettre en œuvre la factorisation ; cela nous permettrait de modifier seulement une ligne de code si nous souhaitons modifier le format d'affichage par exemple.

Pour cette factorisation nous avons le choix, soit écrire une nouvelle méthode, soit simplement une fonction Objective-C ou encore une macro-instruction. Utilisons cette dernière possibilité. Définissez une macro `stringWithCurrency()` au début du fichier *MainViewController.m* :

```
#import "MainViewController.h"
#define stringWithCurrency(currency)
     [NSString localizedStringWithFormat: @"%.2f",currency]
@implementation MainViewController
```

Vous pouvez utiliser cette macro dans les méthodes `-changeValue:` et `-flipsideViewControllerDidFinish:`, par exemple : `euroField.text = stringWithCurrency(self.convertisseur.euro);`.

Tester

Reconstruisez et testez l'application pour vérifier son comportement. Effectivement les champs *dollars* et *euros* prennent la valeur `0.00` lorsque l'utilisateur modifie le taux de conversion, et uniquement dans ce cas ; leur valeur est inchangée si l'utilisateur n'a pas édité le taux de conversion.

5.3. Messages d'alerte

Rien n'interdit à l'utilisateur de *Convertisseur2* de saisir un taux de conversion négatif ou totalement anormal ; `123456,54` par exemple.

Pour corriger ce léger défaut, nous pourrions enrichir la classe `NumericFieldDelegate` de façon à pouvoir en paramétrer le comportement, en fixant des bornes *min* et *max*, par exemple.

Une autre possibilité, celle que nous allons adopter, consiste à signaler à l'utilisateur que le taux de conversion saisi est erroné.

Afficher une alerte

Nous souhaitons afficher une alerte si le taux de conversion est erroné. Il faut donc définir une fonction qui en vérifie la validité.

Définir la validité du taux de conversion

Ajoutez une fonction `verifyRate` dans le fichier *FlipsideViewController.m* ; nous utilisons une fonction C cette fois, pour changer et explorer une autre possibilité :

```
#import "FlipsideViewController.h"

BOOL verifyRate (float rate) {
   return (rate >= 0.5) && (rate <= 2.);
}

@implementation FlipsideViewController
```

Un taux de conversion compris entre `0,5` et `2` est correct.

Nous avons créé cette fonction à l'extérieur de la définition de la classe `FlipsideViewController` (avant la clause `@implementation`). Ainsi nous n'avons pas besoin de la déclarer dans l'interface de la classe. De ce fait, elle est inaccessible pour les autres objets ; nous avons défini une fonction privée à la classe `FlipsideViewController`.

Tester la validité du taux saisi

Nous allons tester la validité du taux saisi par l'utilisateur :

- Lorsque l'utilisateur veut revenir à la vue principale, il doit rester dans la fenêtre modale tant que le taux est incorrect.
- Lorsque l'utilisateur touche le bouton **Terminé** pour effacer le clavier ; si le taux est incorrect un message d'alerte est affiché et le champ de texte doit rester en édition.

Modifiez la méthode `-done` dans le fichier *FlipsideViewController.m* :

```
- (IBAction)done {
   if ( !(self.rateFieldDelegate.isModified &&
 !verifyRate(self.rateFieldDelegate.value)) ) {
      if (self.rateFieldDelegate.isModified) self.rate =
                                 self.rateFieldDelegate.value;
```

```
        [self.delegate flipsideViewControllerDidFinish:self];
    }
}
```

Afficher une fenêtre d'alerte

Modifiez la méthode `-doneEditing:` dans le fichier *FlipsideViewController.m* :

```
- (IBAction) doneEditing:(id)sender {
    if (self.rateFieldDelegate.isModified &&
 !verifyRate(self.rateFieldDelegate.value)) {
        UIAlertView *alert = [[UIAlertView alloc]
                    initWithTitle:@"Taux incorrect" message:
  @"Le taux de conversion doit être compris entre 0,5 et 2"
                         delegate:nil
                cancelButtonTitle:@"OK"
                otherButtonTitles:nil];
        [alert show];
        [alert release];
        [sender becomeFirstResponder];
    } else {
        [sender resignFirstResponder];
    }
}
```

Remarquez l'émission du message `-becomeFirstResponder` sur le champ de texte afin qu'il reste en mode Edition.

Nous faisons connaissance ici avec la classe `UIAlertView` et ses deux méthodes principales :

- `-initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:` :
  - Le titre est une chaîne de caractères qui sera affichée en haut de la fenêtre d'alerte.
  - Le message est une chaîne de caractères contenant des informations plus détaillées également affichées.
  - La possibilité est donnée d'affecter un délégué ; ici, c'est inutile donc on met la valeur `nil`.
  - Une chaîne de caractères contenant le titre du bouton principal.
  - Il est possible aussi d'ajouter d'autres boutons, si besoin on indique ici une liste de chaînes séparées par une virgule et terminée par `nil`.
- `-show` qui affiche la fenêtre au milieu de l'alerte. Ensuite, l'instance peut être libérée et la fenêtre sera détruite dès que l'utilisateur aura touché un bouton.

Les principales méthodes et propriétés de la classe `UIAlertView` sont résumées dans le tableau ci-après.

Tableau 5.2 : Méthodes et propriétés principales de la classe UIAlertView

Type	Titre	Objet
Méthodes	`-(id) initWithTitle: (NSString *)title message: (NSString *)message delegate:(id)delegate cancelButtonTitle:(NSString *)cancelButtonTitle otherButtonTitles: (NSString *)otherButton Titles, ...`	Crée une fenêtre d'alerte avec un titre et en fixant le délégué. La liste des autres boutons doit se terminer par `nil`.
	`- (NSString *) buttonTitle AtIndex:(NSInteger)button Index`	Retourne le titre du bouton dont le numéro d'indice est passé en paramètre. Les indices sont numérotés à partir de `0`.
	`- (void)show`	Affiche le récepteur avec une animation.
Propriétés	`@property(nonatomic, readonly) NSInteger numberOfButtons`	Retourne le nombre de boutons du récepteur.
	`@property(nonatomic) NSInteger cancelButtonIndex`	Indice du bouton principal. Ou `-1` si aucun bouton n'est défini.

Construisez l'application et testez-la. Un message d'alerte s'affiche lorsque vous essayez de sortir du mode d'édition du taux de conversion avec une valeur erronée.

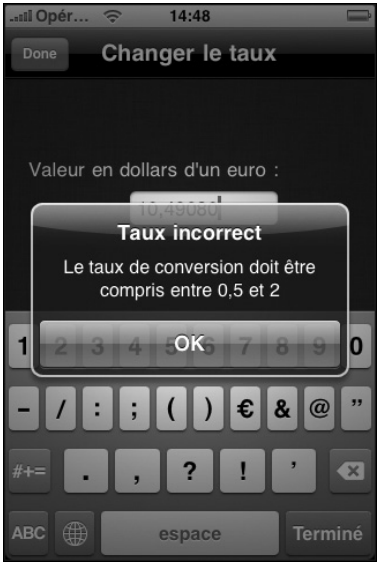


Figure 5.17 : Message d'alerte

Feuilles d'action

Le fonctionnement que nous avons adopté pour l'application *Convertisseur2* interdit à l'utilisateur d'employer un taux de conversion jugé erroné. Nous pourrions préférer un comportement plus souple : signaler que le taux de conversion paraît incorrect et laisser l'utilisateur choisir de l'éditer à nouveau ou de l'utiliser tel quel.

Les **feuilles d'action** gérées par la classe `UIActionSheet` permettent d'implémenter ce mécanisme ; une alerte est affichée à l'écran avec deux boutons ou plus :

- un bouton d'**annulation** (*Cancel Button*) permettant à l'utilisateur d'annuler l'opération en cours ;
- un bouton d'**action** (*Destructive Button*) permettant à l'utilisateur d'effectuer l'action.

Remarquez le nom anglais du bouton d'action (**Destructive**). Il évoque l'utilité des feuilles d'actions. L'utilisateur est prévenu que l'action peut être dangereuse. D'ailleurs, le bouton d'action est rouge par défaut.

Figure 5.18 : Feuille d'action

Les feuilles d'action s'utilisent de la même façon que les fenêtres d'alerte. Leur comportement diffère sur les points suivants :

- Une feuille d'action s'affiche par-dessus une vue particulière au lieu de s'afficher au milieu de l'écran.

- Elle est généralement activée par un contrôleur de vue, par l'émission du message `showInView:self.view`.
- Une feuille d'action offre un titre mais pas de message détaillé.
- Par défaut, une feuille d'action propose une alternative à l'utilisateur, c'est-à-dire deux boutons au lieu d'un.

Le tableau ci-après résume les principales méthodes et propriétés de la classe `UIActionSheet`.

Tableau 5.3 : Méthodes et propriétés principales de la classe UIActionSheet

Type	Titre	Objet
Méthodes	`– (id) initWithTitle: (NSString *)title delegate: (id < UIActionSheet Delegate >)delegate cancel ButtonTitle:(NSString *) cancelButtonTitle destructiveButtonTitle: (NSString *)destructive ButtonTitle otherButton Titles:(NSString *)other ButtonTitles, ...`	Crée une feuille d'action avec un titre en précisant le délégué. La feuille présente un bouton d'annulation et un bouton d'action. La liste des autres boutons doit se terminer par `nil`.
	`– (NSString *) buttonTitle AtIndex:(NSInteger)button Index`	Retourne le titre du bouton dont le numéro d'indice est passé en paramètre. Les indices sont numérotés à partir de `0`.
	`– (void)showInView: (UIView *)view`	Affiche le récepteur avec une animation à partir de la vue passée en paramètre. Il est recommandé d'utiliser une vue racine (vue principale dans une fenêtre).
Propriétés	`@property(nonatomic, readonly) NSInteger numberOfButtons`	Nombre de boutons du récepteur
	`@property(nonatomic) NSInteger cancelButtonIndex`	Indice du bouton d'annulation ou `–1` s'il n'est pas défini
	`@property(nonatomic) NSInteger destructive ButtonIndex`	Indice du bouton d'action ou `–1` s'il n'est pas défini

L'utilisateur pouvant toucher l'un ou l'autre bouton pour sortir de la feuille d'action, il faut que l'application puisse déterminer quel bouton a été touché. Vous avez certainement déjà deviné le mécanisme mis en œuvre : c'est encore la délégation.

Délégué de feuille d'action

Le protocole de délégué pour la classe `UIActionSheet` est `UIActionSheetDelegate`. Dans l'utilisation la plus courante, le contrôleur de vue qui active une feuille d'action se définit comme son délégué en passant `self` comme paramètre `delegate:` lors de l'initialisation de la feuille d'action.

Le délégué implémente généralement la méthode `–actionSheet:clickedButtonAtIndex:` de la façon suivante :

```
- (void)actionSheet:(UIActionSheet *) actionSheet
              clickedButtonAtIndex:(NSInteger)buttonIndex {
   if (buttonIndex == [actionSheet cancelButtonIndex]) {
      // le bouton d'annulation a été touché
   }
   else if (buttonIndex ==
                 [actionSheet destructiveButtonIndex]) {
      // le bouton d'action a été touché
   }
}
```

Challenge

Modifiez l'application *Convertisseur2* en utilisant une feuille d'action à la place d'une fenêtre d'alerte. Ainsi, l'utilisateur pourra forcer un taux de conversion qui nous paraît anormal.

Délégué d'alerte

De la même façon que la feuille d'action, le protocole `UIAlertViewDelegate` permet de définir des délégués pour les fenêtres d'alerte de la classe `UIAlertView`.

La principale méthode de ce délégué est `–alertView:clickedButtonAtIndex:` qui se programme de la même façon que la méthode équivalente du protocole `UIActionSheetDelegate`.

Dans son comportement par défaut, une fenêtre d'alerte n'a pas besoin de délégué puisqu'elle ne comporte qu'un bouton. Mais nous pouvons créer une fenêtre d'alerte avec plusieurs boutons et donc un délégué. Inversement, il est possible de définir une feuille d'action sans délégué avec un seul bouton.

En fait, les classes `UIAlertView` et `UIActionSheet` se programment exactement de la même façon, seule leur apparence visuelle diffère.

5.4. Barre d'onglets

Nous avons vu comment créer une application de type utilitaire qui présente une vue principale et une vue secondaire (vue modale) à l'utilisateur, et les mécanismes pour passer de l'une à l'autre.

Si nous voulons produire une application offrant trois vues ou plus, la navigation par vue modale peut ne pas se révéler satisfaisante pour l'utilisateur. La navigation par **barre d'onglets** est plus adaptée ; l'utilisateur a toujours la possibilité d'accéder à n'importe quelle vue en touchant l'**onglet** correspondant. Les copies d'écran montrent notre application Convertisseur2 si elle avait été développée avec une barre d'onglets.

Figure 5.19 : Convertisseur2 avec une barre d'onglets

Créer une barre d'onglet

La barre d'onglets se situe en bas de l'écran, c'est un objet de la classe `UITabBar` qui hérite de la classe `UIView`.

Figure 5.20 : Barre d'onglets comprenant 2 éléments

On peut placer de 2 à 6 **éléments** sur une barre d'onglets ; il n'est pas interdit d'en mettre plus, mais au-delà ils risquent de se chevaucher. Chaque élément est un objet de la classe `UITabBarItem` affiché avec un logo et un titre. Vous pouvez utiliser un des 12 éléments pour lesquels le logo et le titre sont prédéfinis ou créer votre propre logo. Il est également possible d'ajouter un **badge** contenant généralement une valeur numérique sur un élément de barre d'onglets, mais on peut y inscrire une chaîne de caractère quelconque ; il est conseillé d'en limiter la taille à deux ou trois caractères.

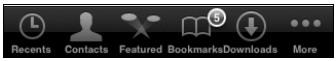


Figure 5.21 : Éléments prédéfinis de barre d'onglets, dont l'un avec un badge

Pour afficher un badge, il suffit d'affecter une chaîne de caractères à la propriété `badgeValue` (de type `NSString`) de l'élément de barre d'onglets.

Les autres éléments sont définis dans l'un des fichiers NIB de l'application ; nous allons détailler cela bientôt.

Pour créer une barre d'onglets, il suffit sous Interface Builder de faire glisser un *contrôleur de barre d'onglets* (*Tab Bar Controller*) dans la fenêtre du fichier NIB.

Figure 5.22 : Contrôleur de barre d'onglets dans la bibliothèque d'Interface Builder

Une autre possibilité consiste à créer une **application à barre d'onglets** (*Tab Bar Application*) sous XCode. Dans ce cas, vous pouvez

choisir le produit pour lequel l'application est développé : iPhone (utilisable aussi sur iPod Touch et iPad) ou iPad.

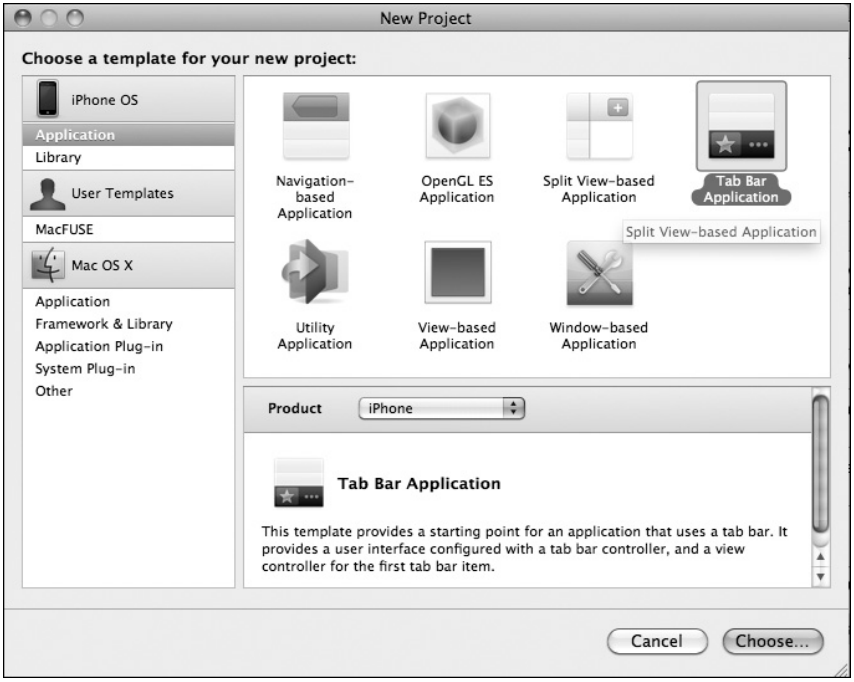


Figure 5.23 : Création d'une application à barre d'onglets

Utiliser un contrôleur de barre d'onglets

Le contrôleur de barre d'onglets, instance de la classe `UITabBarController`), prend en charge la navigation entre les onglets. Son utilisation nécessite peu d'effort de la part du programmeur.

Dès que l'on ajoute un contrôleur de barre d'onglets à un fichier NIB, il est associé à :

- une barre d'onglets ; nous n'aurons généralement pas à nous en occuper ;
- une liste de contrôleurs de vue (`UIViewController`) qui contiennent chacun :
  - un élément de barre d'onglets ;
  - éventuellement une vue, sauf si le contrôleur de vue est associé à un fichier NIB spécifique.

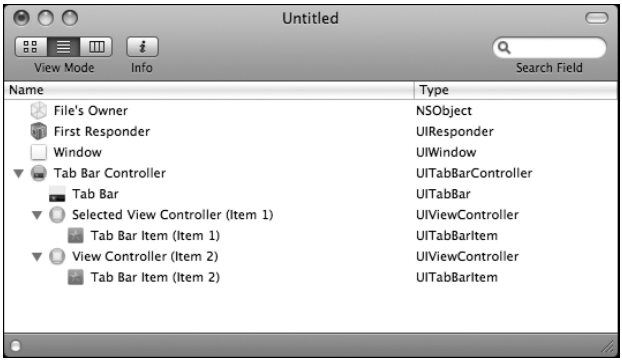


Figure 5.24 : Ajout d'un contrôleur de barre d'onglets dans un fichier NIB

Les opérations le plus courantes pour définir une application par onglets sont décrites ci-après.

Ajouter un onglet

Pour ajouter un onglet sous Interface Builder, il suffit d'ajouter un contrôleur de vue dans le contrôleur de barre d'onglets. Un élément de barre d'onglets sera automatiquement ajouté au nouveau contrôleur de vue.

Adapter l'élément de barre d'onglet

L'inspecteur de l'élément de barre d'onglets, sous Interface Builder, permet de définir son icône et son titre.

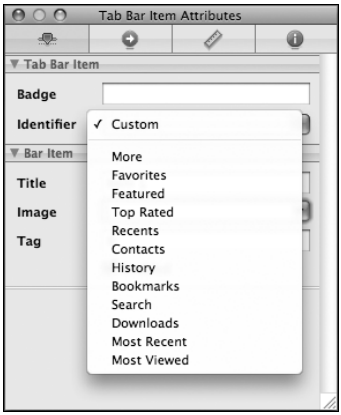


Figure 5.25 : Liste des éléments de barre d'onglets prédéfinis

Vous pouvez également définir vos propres éléments de barre d'onglets en saisissant son titre et le nom du fichier à utiliser comme icône. Ce fichier doit être au format *PNG* et d'une taille de

30 x 30 pixels. Seule la couche alpha (transparence) de l'image sera utilisée lors de l'affichage.

Définir la vue de chaque onglet

Chaque onglet dispose de son propre contrôleur de vue. Vous savez déjà comment utiliser un contrôleur de vue :

- Il faut créer une nouvelle classe qui dérive de la classe `UIViewController` pour y définir ses propres outlets et actions.
- Il faut associer une vue (`UIView`) à ce contrôleur pour y ajouter les contrôles de l'interface utilisateur ; boutons, champs de texte, etc. Cette vue peut être associée.
  - soit en faisant glisser un objet `View` dans le contrôleur de vue ;
  - soit en donnant le nom du fichier NIB qui décrit la Vue, dans l'inspecteur du contrôleur de vue sous Interface Builder.
- Pour finir, il faut établir les connexions entre les contrôles définis dans la vue et les outlets et actions du contrôleur de vue.

Les contrôleurs de vue associés à un contrôleur de barre d'onglets ne dérogent pas à ce mode opératoire. Qu'un contrôleur de vue appartienne à un contrôleur de barre d'onglets est presque transparent pour l'utilisateur.

La propriété `tabBarItem` (de type `UITabBarItem`) contient l'élément de barre d'onglets associé au contrôleur de vue. Elle est définie dans la classe `UIViewController` et donc disponible dans toutes les classes dérivées, par exemple pour y afficher un badge.

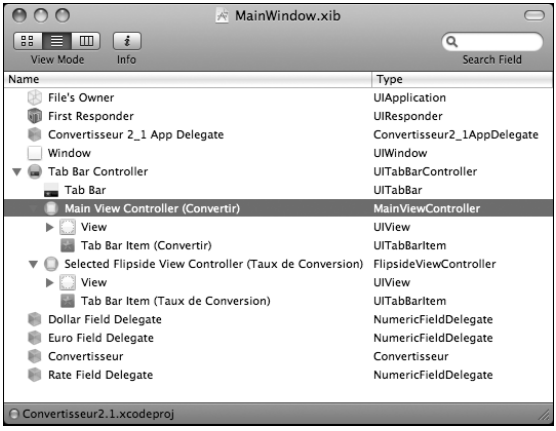


Figure 5.26 : Exemple de fichier NIB contenant les vues pour tous les onglets

Modifier la navigation par onglets

Insérer un contrôleur de vue dans un contrôleur de barre d'onglets suffit pour le fonctionnement de la navigation entre les différents onglets. Il n'est pas nécessaire de dériver la classe `UITabBarController`, elle est utilisée telle quelle.

Si besoin, on peut en modifier le comportement en utilisant le délégué de la classe `UITabBarController`. Il doit adopter le protocole `UITabBarControllerDelegate` dont les méthodes principales sont données dans le tableau.

Tableau 5.4 : Principales méthodes du protocole UITabBarControllerDelegate

Signature de la méthode	Objet de la méthode
`– (BOOL) tabBarController: (UITabBarController *) tabBarController should SelectViewController:(UIView Controller *)viewController`	Demande au délégué si le contrôleur de vue peut être activé.
`– (void) tabBarController: (UITabBarController *)tabBar Controller didSelectView Controller:(UIViewController *) viewController`	Informe le délégué qu'un contrôleur vient d'être sélectionné. Ce peut être le même que celui qui est déjà sélectionné.

Par exemple, dans une application *Convertisseur2* basée sur une barre d'onglets, nous pourrions utiliser la méthode `–tabBar Controller:shouldSelectViewController:` afin de mettre à jour l'affichage des champs de texte de la vue principale, lorsque l'utilisateur a modifié le taux de conversion.

```
- (void) tabBarController:(UITabBarController *)
                 tabBarController didSelectViewController:
                      (UIViewController *)viewController {
   if (viewController==self) {
      dollarField.text =
            stringWithCurrency(self.convertisseur.dollar);
      euroField.text =
              stringWithCurrency(self.convertisseur.euro);
   }
}
```

Challenge

Inspirez-vous des éléments contenus dans cette section pour réécrire l'application *Convertisseur2* avec une barre d'onglets plutôt qu'avec une fenêtre modale.

Le code de cette nouvelle version est plus simple ; les méthodes permettant d'activer et d'effacer la vue modale ne sont plus nécessaires.

5.5. Barres de navigation

Les **barres de navigation** (*Navigation Bar*) sont principalement utilisées pour parcourir une structure hiérarchique de données. L'application *Contacts* en est un exemple :

- La **vue racine** (*Root View*) contient la liste des groupes.
- Lorsqu'on sélectionne un groupe, on accède à une vue contenant la liste des contacts de ce groupe.
- Lorsqu'on sélectionne un contact, on affiche une vue contenant les informations détaillées de ce contact.
- Chacune de ces vues contient une barre de navigation, en haut de l'écran, contenant le **titre** de la vue.
- La barre de navigation de toutes les vues, sauf celle de la vue racine, offre un **bouton de retour** (*Back Button*) qui permet de revenir à la vue précédente ; la vue précédente et le bouton de retour ont le même titre.
- La barre de navigation peut offrir un bouton supplémentaire à droite.

Figure 5.27 : Navigation dans l'application Contacts

Spécificité iPhone/iPod Touch

L'application avec barre de navigation est spécifique à l'iPhone et à l'iPod Touch du fait de leur écran de taille réduite. L'équivalent sur iPad est l'application à vue fractionnée (*Split View*) détaillée dans le chapitre qui décrit les spécificités de cet appareil.

Créer une barre de navigation

À l'instar de la barre d'onglets, pour créer une barre de navigation sous Interface Buider, il faut faire glisser un *contrôleur de navigation* (*Navigation Controller*) dans la fenêtre du fichier NIB.

Le contrôleur de navigation, instance de la classe `UINavigationController`), prend en charge la navigation entre les vues. Son utilisation nécessite peu d'effort de la part du programmeur.

Dès que l'on ajoute un contrôleur de navigation à un fichier NIB, il est associé à :

- une barre de navigation ; nous n'aurons généralement pas à nous en occuper ;
- une contrôleur de vue racine (`UIViewController`) qui contient un élément de navigation.

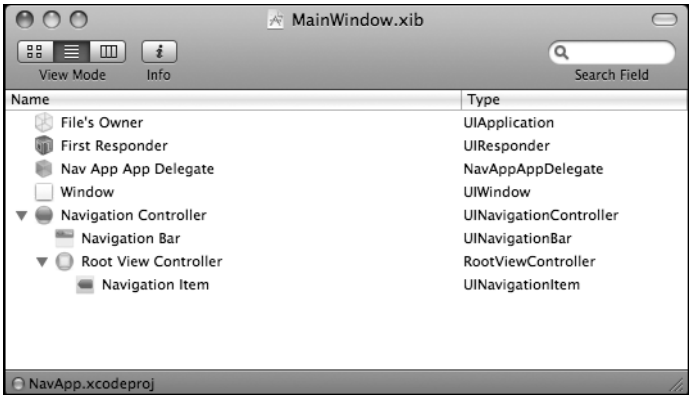


Figure 5.28 : Contrôleur de navigation dans un fichier NIB

À la différence d'un contrôleur de barre d'onglets, qui contient tous les contrôleurs de vue accessibles à l'utilisateur, le contrôleur de navigation contient seulement le contrôleur de la vue racine. Les autres vues devront être ajoutées par programmation.

Utiliser une barre de navigation

La navigation par barre de navigation est adaptée pour présenter des vues contenant des informations de plus en plus détaillées. À partir d'une vue, on peut soit ajouter une vue contenant une information plus détaillée, soit revenir à la vue précédente qui contient des informations moins détaillées.

On parle de **pile de navigation** pour désigner tous les contrôleurs de vues gérés par le contrôleur de navigation, ceux qui doivent être conservés car l'utilisateur doit pouvoir y revenir.

Pile

Une **pile** est une collection dans laquelle seul le dernier objet ajouté est accessible. On empile un objet pour l'ajouter à la collection, on le dépile pour l'en retirer. Une pile d'objets fonctionne comme une pile d'assiettes.

La pile est initialisée avec le contrôleur de vue racine. Pour changer de vue, il faut empiler un contrôleur de vue dans la **pile de navigation**, en envoyant un message `–pushViewController:animated:` au contrôleur de navigation. Nous verrons un exemple de mise en œuvre au chapitre suivant.

Lorsqu'un contrôleur de vue est empilé, la vue associée est affichée avec une barre de navigation et un bouton de retour. Lorsque l'utilisateur touche le bouton de retour, le contrôleur de vue est dépilé et le contrôleur de vue suivant dans la pile est affiché.

5.6. Checklist

Nous avons vu dans ce chapitre les principaux types d'applications multivues :

- utilitaire, avec une vue principale et une vue modale ;
- application à barre d'onglets, avec le contrôleur de barre d'onglets `UITabBarController` ;
- application à barre de navigation et le contrôleur de navigation `UINavigationController`.

Nous avons détaillé le fonctionnement des vues modales et du contrôleur de barre d'onglets et réalisé une version 2 de notre convertisseur permettant à l'utilisateur de modifier le taux de conversion.

Nous mettrons en application le principe de fonctionnement du contrôleur de navigation aux chapitres suivants : nous créerons une application pour naviguer dans une structure de données.

Nous avons également examiné le fonctionnement des alertes (`UIAlertView`) et des feuilles d'action (`UIActionSheet`).

CONTRÔLES COMPLEXES

Dans ce chapitre, nous allons examiner le fonctionnement des contrôles visuels qui dépendent d'un ensemble de données :

- vues en table, qui permettent de présenter une liste de données ;
- sélectionneurs, qui permettent à l'utilisateur de sélectionner une valeur.

Nous en profiterons pour apprendre à manipuler des dates ainsi que les conteneurs utilisés en Objective-C : tableaux et dictionnaires. Nous utiliserons ces éléments pour débuter l'application *Emprunts1*, un aide-mémoire pour nous souvenir des objets que nous avons prêtés à nos amis, ce qui nous permettra aussi de mettre en œuvre les barres de navigation.

6.1. Utiliser un sélectionneur

Un **sélectionneur** (*picker*) est un contrôle visuel en forme de tambour ; l'utilisateur le fait tourner pour choisir une valeur.

Figure 6.1 : Exemples de sélectionneurs

Sélectionneur de date

Nous allons commencer par une mise en pratique du cas le plus simple : le **sélectionneur de date** (*date picker*).

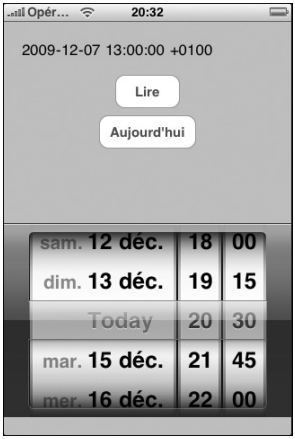


Figure 6.2 : Mise en œuvre du sélectionneur de date

Exemple de mise en œuvre

Créez un nouveau projet de type *View-based Application* sous XCode. Appelez-le `Picker1`.

Création de l'interface

Ouvrez le fichier *Picker1ViewController.xib* et composez l'interface utilisateur avec :

- un *Label* ;
- deux *boutons* dont vous changez le titre : `Lire` et `Aujourd'hui` ;
- un *sélectionneur de date*.

Figure 6.3 : Composition de l'interface

Création du contrôleur de vue

1 Modifiez le fichier *Picker1ViewController.h* :

```
#import <UIKit/UIKit.h>
@interface Picker1ViewController : UIViewController {
  IBOutlet UILabel * label;
  IBOutlet UIDatePicker * datePicker ;
}
@property (nonatomic,assign) UILabel * label;
@property (nonatomic,assign) UIDatePicker * datePicker;
-(IBAction) readPicker ;
-(IBAction) setPicker ;
@end
```

2 Établissez les connexions sous Interface Builder. L'action *readPicker* doit être connectée à l'événement *Touch Up Inside* du bouton

Lire. L'action `setPicker` doit être connectée à l'événement *Touch Up Inside* du bouton **Aujourd'hui**.

3 Ouvrez le fichier *Picker1ViewController.m*, modifiez la méthode `–viewDidUnload` et ajoutez les méthodes `–readPicker` et `–setPicker` :

```
@synthesize label;
@synthesize datePicker;
- (IBAction) readPicker {
  label.text = [[datePicker date] description];
}
- (IBAction) setPicker {
    [datePicker setDate:[NSDate date] animated:YES] ;
}
- (void)viewDidUnload {
  // Release any retained subviews of the main view.
  // e.g. self.myOutlet = nil;
  self.label = nil;
  self datePicker = nil;
}
```

4 Construisez l'application et testez-la sur le simulateur.

Nous détaillerons bientôt les classes `UIDatePicker` et `NSDate` utilisées dans cette application. Nous avons employé par ailleurs une méthode `–description`. Cette méthode est définie dans la classe `NSObject` ; elle est donc disponible dans toutes les classes et retourne une chaîne de caractères qui décrit le récepteur.

Décrivez vos instances

Pensez à définir la méthode `–description` dans les classes que vous définissez, vous pourrez ainsi utiliser le descripteur `%@` pour inclure vos instances dans une chaîne de caractères.

La classe UIDatePicker

Testez l'application Picker1 en essayant plusieurs configurations de paramètres sous Interface Builder.

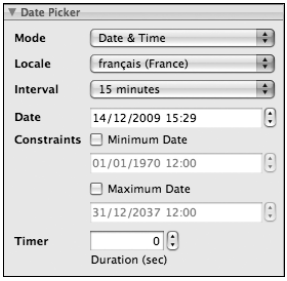


Figure 6.4 : Paramètres d'un sélectionneur de date

Le paramètre **mode** permet de modifier la présentation visuelle du sélectionneur en fonction de l'usage que l'on veut en faire. Un sélectionneur de date est un objet de la classe `UIDatePicker`. Ses différents paramètres peuvent être définis sous Interface Builder ou par programmation à l'aide des propriétés des instances de la classe.

Tableau 6.1 : Présentations visuelles du sélectionneur de date

Présentation	Mode sous Interface Builder	Propriété datePickerMode
	Date&Time	`UIDatePickerModeDate AndTime`
	Time	`UIDatePickerModeTime`
	Date	`UIDatePickerModeDate`
	Timer	`UIDatePickerModeCount DownTimer`

Tableau 6.2 : Principales propriétés de la classe UIDatePicker

Thème	Propriété	Objet de la propriété
Date et calendrier	`@property(nonatomic, retain) NSDate *date`	Date affichée ou 0 si le sélectionneur est en mode Timer.
Mode	`@property(nonatomic) UIDatePickerMode datePickerMode`	Mode d'affichage du sélectionneur.
Attributs temporels	`@property(nonatomic, retain) NSDate *maximumDate`	Date maximale affichée ou `nil` s'il n'y a pas de maximum.
	`@property(nonatomic, retain) NSDate *minimumDate`	Date minimale affichée ou `nil` s'il n'y a pas de minimum.
	`@property(nonatomic) NSInteger minuteInterval`	Intervalle minimum affiché, en minutes. Doit être un diviseur de `60`. Valeur minimum `1` (par défaut) et valeur maximum `30`.
	`@property(nonatomic) NSTimeInterval countDownDuration`	Durée affichée comprise entre `0` et `23h59`, ou `0` lorsque le sélectionneur n'est pas en mode Timer.

La classe `UIDatePicker` définit également une méthode `–setDate: animated:` qui permet de changer la date affichée avec une animation. Pour changer la valeur affichée, on peut aussi modifier la propriété `date`. Dans ce cas, il n'y a pas d'animation. Essayez ces deux procédés dans la méthode `–setPicker` de l'application *Picker1*.

Vous pouvez également connecter l'action `–datePicker` du contrôleur de vue *Picker1ViewController* à l'événement *Value Changed* du sélectionneur de date ; le texte du *label* évoluera dès que l'utilisateur changera la valeur affichée.

NSTimeInterval

La propriété `countDownDuration` de la classe `UIDatePicker` est de type `NSTimeInterval`, équivalent au type `double`. Il représente un intervalle de temps exprimé en secondes.

Gestion des dates

La gestion des dates est un réel défi sur un appareil destiné à être utilisé dans le monde entier, qui doit donc prendre en compte les différentes façons de représenter les dates et les différents calendriers. Le système mis en place par Apple permet de simplifier la tâche des développeurs qui souhaitent la plus large diffusion de leurs applications. Elle pourra vous paraître un peu complexe si vos ambitions sont plus limitées.

Le mot *date* désigne simplement un instant précis mesuré à partir d'une référence absolue. Un *calendrier* est une structuration du temps en jour-mois-année. Un instant donné est toujours représenté par la même date, indépendante de la localisation, alors que le calendrier dépend de la culture et du lieu géographique. Outre le calendrier grégorien utilisé en occident, iPhone OS propose les calendriers hébreu, islamique, chinois, bouddhiste et japonais.

Pour exprimer un instant en jour-mois-année, il faut la combinaison d'un instant (la date) et d'un calendrier. On obtient alors les *composants* d'une date (le jour, le mois et l'année) dans un calendrier donné.

Concentrons-nous sur les classes les plus utilisées :

- `NSDate` qui représente une date ;
- `NSDateFormatter` qui permet d'effectuer les conversions entre chaîne de caractères et date.

NS et UI

Le nom de chaque classe commence par deux caractères majuscules qui identifient le framework dans lequel la classe est définie. Par exemple UI pour *UIKit* et NS pour *NextStep*. Ce système est un ancêtre de Mac OS X. Les classes NS que nous utilisons sont communes aux environnements iPhone OS et Mac OS X. Le framework *UIKit* est disponible uniquement dans l'environnement iPhone OS.

La classe NSDate

Les dates, ou instants particuliers, sont représentées par des instances de la classe `NSDate` dont les principales méthodes sont résumées dans le tableau. Elles permettent de réaliser l'arithmétique de base sur les dates :

- comparer deux dates ;
- calculer la durée espaçant deux dates ;
- définir une nouvelle date en ajoutant une durée à une date.

Bien entendu, les durées peuvent être positives ou négatives. Elles sont du type `NSTimeInterval` qui n'est rien d'autre qu'un `double` exprimant une durée en secondes.

Tableau 6.3 : Principales méthodes de la classe NSDate

Thème	Méthode	Objet de la méthode
Création et initialisation	`+ (id)date`	Méthode de classe qui retourne une instance initialisée à l'instant présent
	`– (id)init`	Initialise le récepteur à l'instant présent.
	`+ (id) dateWithTimeIntervalSinceNow: (NSTimeInterval)seconds`	Méthode de classe qui retourne une instance initialisée à un nombre donné de secondes à partir de l'instant présent
	`– (id) initWithTimeIntervalSinceNow: (NSTimeInterval)seconds`	Initialise le récepteur à un nombre donné de secondes à partir de l'instant présent.
Comparaisons	`– (NSDate *) earlierDate:(NSDate *) anotherDate`	Retourne la date la plus précoce entre le récepteur et la date donnée en paramètre.
	`– (NSDate *) laterDate: (NSDate *)anotherDate`	Retourne la date la plus tardive entre le récepteur et la date donnée en paramètre.
	`– (NSComparisonResult) compare:(NSDate *) anotherDate`	Compare le récepteur à la date donnée en paramètre. Retourne `NSOrderedSame` lorsque les dates sont identiques, `NSOrderedDescending` lorsque le récepteur est plus tardif que la date donnée en paramètre, et `NSOrderedAscending` si elle est plus précoce.
Obtenir des durées	`– (NSTimeInterval) timeIntervalSinceDate: (NSDate *)anotherDate`	Retourne la durée entre le récepteur et la date passée en paramètre.
	`– (NSTimeInterval) timeIntervalSinceNow`	Retourne la durée entre le récepteur et l'instant présent.
Ajouter une durée	`– (id) addTimeInterval: (NSTimeInterval)seconds`	Crée une nouvelle date initialisée à un nombre donné de secondes à partir du récepteur.

La classe NSDateFormatter

La classe `NSDateFormatter` permet de convertir une chaîne de caractères en une date et vice-versa. Chaque instance de cette classe contient un calendrier, un fuseau horaire et une localisation qui sont par défaut ceux réglés dans l'appareil. Le format de conversion doit être spécifié par la méthode `–setDateFormat:`. On utilise ensuite l'une des deux méthodes `–dateFromString:` et `–stringFromDate:` pour effectuer les conversions.

1 Modifiez la méthode `readPicker` de la classe `Picker1ViewController` dans l'application *Picker1* :

```
- (IBAction) readPicker {
    formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"EEEE dd MMMM HH:mm"];
    label.text=[formatter stringFromDate:[datePicker date]];
    [formatter release];
}
```

2 Testez l'application sur le simulateur d'iPhone. Changez la localisation (**Réglages**->**Général**->**International**->**Format régional**) et vérifiez que le texte affiché et le sélectionneur de date répercutent la localisation par défaut de l'appareil.

Figure 6.5 : Picker1 sous différentes localisations

Tableau 6.4 : Principales méthodes de la classe NSDateFormatter

Thème	Méthode	Objet de la méthode
Initialisation	`– (id) init`	Initialise le récepteur avec les paramètres par défaut de l'appareil (calendrier, fuseau horaire, localisation).
Conversion	`– (NSDate *) dateFromString: (NSString *)string`	Convertit une chaîne de caractères en date.
	`– (NSString *) stringFromDate: (NSDate *)date`	Convertit une date en chaîne de caractères.
Formats	`– (void) setDateFormat:(NSString *) string`	Définit le format de conversion selon le standard technique n°35 de l'Unicode.
Gestion des symboles	`– (void) setWeekdaySymbols:(NSArray *) array`	Définit la représentation des jours de la semaine à utiliser. Le premier élément de tableau est le dimanche.
	`– (void) setMonthSymbols:(NSArray *) array`	Définit la représentation des mois de l'année à utiliser.

Vous pouvez consulter le standard technique n°35 de l'*Unicode* sur le site de l'organisation (**http://unicode.org/reports/tr35/tr35-6.html#Date_Format_Patterns**) pour connaître toutes les possibilités de formatage des dates. Un *format de date* est une chaîne de caractères contenant des codes qui représentent les différentes composantes d'une date.

Tableau 6.5 : Codes de formatage de date les plus courants

Code	Représente	Exemple pour le 12/12/2010 à 15:30
`yy`	Les 2 derniers chiffres de l'année	`10`
`yyyy`	L'année	`2010`
`MM`	Le mois numérique	`12`
`MMMM`	Le mois littéral	`décembre`
`dd`	Le jour dans le mois	`12`
`EEEE`	Le jour dans la semaine littéral	`samedi`
`HH`	L'heure (de 0 à 23)	`15`
`mm`	Les minutes	`30`

En utilisant un sélectionneur `UIDatePicker` et un formateur `NSDateFormatter`, le développeur a l'assurance que les dates seront toujours affichées en employant le réglage régional décidé par l'utilisateur de l'appareil.

Challenge

Les utilisateurs pointilleux auront remarqué que les langues régionales (provençal, breton, occitan…) ne sont pas disponibles sur l'iPhone. Heureusement, la classe `NSDateFormatter` est pleine de ressources. Les méthodes `–setWeekdaySymbols:` et `–setMonthSymbols:` permettent de définir la représentation des jours de la semaine et des mois de l'année.

Votre objectif est d'afficher la date en *breton* dans le label de l'application *Picker1*.

Figure 6.6 : Affichage de la date en breton

Vous pourrez utiliser un formateur de date qui pourrait être initialisé dans la méthode `–viewDidLoad` du contrôleur de vue :

```
- (void)viewDidLoad {
    [super viewDidLoad];
    formatter = [[NSDateFormatter alloc] init];
    NSArray * mois = [NSArray arrayWithObjects:@"Genver",
                      @"C'hwevrer",@"Meurzh",@"Ebrel",@"Mae",
                      @"Mezheven",@"Gouere",@"Eost",
                      @"Gwengolo",@"Here",@"Du",@"Kerzu",nil];
    NSArray * jours = [NSArray arrayWithObjects:@"Sul",
                      @"Lun",@"Meurzh",@"Merc'her",@"Yaou",
                      @"Gwener",@"Sadorn",nil];
    [formatter setMonthSymbols:mois];
    [formatter setWeekdaySymbols:jours];
```

```
    [formatter setDateFormat:@"EEEE d MMMM HH:mm"];
}
```

Sélectionneur standard

Un sélectionneur standard ressemble visuellement à un sélectionneur de date mais il fonctionne différemment. Nous allons commencer par un exemple simple pour découvrir la classe `UIPickerView`.

Application Picker2

L'application Picker2 va simplement présenter un sélectionneur à l'utilisateur, pour lui permettre de choisir un pays. Le pays choisi sera affiché dans un label.

Figure 6.7 : Application Picker2

Créez un nouveau projet de type *View-based Application* sous XCode et appelez-le `Picker2`.

Création de l'interface

Ouvrez le fichier *Picker2ViewController.xib* et composez l'interface utilisateur avec :

- un *Label* ;
- un *sélectionneur standard (Picker View)*.

Interface du contrôleur de vue

1 Modifiez le fichier *Picker2ViewController.h* :

```
#import <UIKit/UIKit.h>
@interface Picker2ViewController : UIViewController
<UIPickerViewDelegate,UIPickerViewDataSource>{
    IBOutlet UILabel * label;
    NSArray * valeurs;
}
@property (nonatomic,assign) UILabel * label;
@end
```

2 Remarquez les différences avec le contrôleur de vue de Picker1 :

- Il n'y a pas d'outlet sur le sélectionneur.
- Nous avons besoin d'un tableau de valeurs.
- Le contrôleur adopte les protocoles `UIPickerViewDelegate` et `UIPickerViewDataSource`.

Le sélectionneur standard nécessite un délégué pour fonctionner. Deux protocoles sont définis et donc on pourrait même dire qu'il lui faut deux délégués. En pratique, ces deux protocoles seront généralement adoptés par un seul contrôleur de Vue. C'est donc le sélectionneur qui connaît le contrôleur de vue, son délégué, et ce dernier n'a donc généralement pas besoin de connaître le sélectionneur ; il n'y a pas d'outlet sur le sélectionneur.

Nous avons besoin d'un tableau de valeur dans le contrôleur de vue car c'est lui, en tant que délégué du sélectionneur, qui doit gérer les valeurs à afficher ; nous allons expliquer cela.

Connexions

Établissez les connexions sous Interface Builder :

- L'outlet *label* du contrôleur de vue doit être connecté au champ *Label* de l'interface.
- Les outlets *delegate* et *dataSource* du sélectionneur doivent être connectés au contrôleur de vue (*File's owner*).

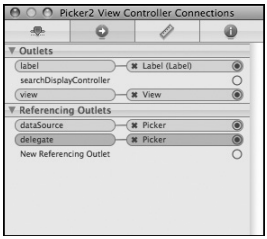


Figure 6.8 : Connexions du contrôleur de vue Picker2ViewController

Code du contrôleur

1 Ouvrez le fichier *Picker2ViewController.m*, modifiez la méthode `–viewDidUnload` et `–viewDidLoad` puis ajoutez les méthodes définies dans les protocoles :

```
@synthesize label;
- (NSInteger)numberOfComponentsInPickerView:
                              (UIPickerView *)pickerView{
   return 1;
}
- (NSInteger)pickerView:(UIPickerView *)pickerView
         numberOfRowsInComponent:(NSInteger)component{
   return [valeurs count];
}
- (NSString *)pickerView:(UIPickerView *)pickerView
             titleForRow:(NSInteger)row
            forComponent:(NSInteger)component{
   return [valeurs objectAtIndex:row];
}
- (void)pickerView:(UIPickerView *)pickerView
       didSelectRow:(NSInteger)row
        inComponent:(NSInteger)component{
   self.label.text = [valeurs objectAtIndex:row];
}
// Implement viewDidLoad to do additional setup after
loading the view, typically from a nib.
- (void)viewDidLoad {
   [super viewDidLoad];
   valeurs = [[NSArray alloc] initWithObjects:@"France",
      @"Allemagne",@"Italie",@"Espagne",@"Portugal",nil];
   self.label.text = [valeurs objectAtIndex:0];
}
- (void)viewDidUnload {
   // Release any retained subviews of the main view.
   // e.g. self.myOutlet = nil;
   self.label = nil;
   [valeurs release];
}
```

2 Construisez l'application et testez-la sur le simulateur.

Nous avons construit un tableau `valeurs` initialisé avec une liste de pays. Au moment où il s'affiche, le sélectionneur demande un certain nombre d'informations à son délégué ; elles seront puisées dans ce tableau :

- Combien y a-t-il de lignes au total ?
- Que dois-je afficher sur telle ou telle ligne ?

Le sélectionneur va également envoyer un message à son délégué chaque fois que l'utilisateur le manipulera.

Nous détaillerons tout cela. Auparavant, étudions la classe `UIPickerView`.

Classe UIPickerView

Un sélectionneur est un objet qui permet d'afficher un ou plusieurs tambours. Chaque tambour contient une liste de valeurs ; l'utilisateur choisit l'une de ces valeurs en le faisant tourner. Les tambours sont les **composants** (*components*) du sélectionneur, chaque composant contient plusieurs **lignes** ou rangées (*rows*).

Figure 6.9 : Sélectionneur à deux composants

Les développeurs ont une tendance naturelle à écrire du code qui décide ce qui doit être affiché à l'écran. La programmation Cocoa est différente, on parle de **contrôle inversé** :

- Les contrôleurs envoient des instructions simples aux vues :
  - Où doivent-elles s'afficher ?
  - Quelles sont leurs propriétés ?
- Les vues prennent en charge toute la partie visuelle : affichage et animation.
- Les vues sous-traitent les activités qui ne relèvent pas strictement du comportement visuel à leur délégué.

Les sélectionneurs `UIPickerView` savent s'afficher et faire tourner les tambours sous l'impulsion de l'utilisateur mais ils ne connaissent pas leur contenu. Pour savoir ce qu'ils doivent afficher, ils consultent leur délégué et leur source de données (voir section suivante).

Tableau 6.6 : Principales méthodes de la classe UIPickerView

Thème	Méthode	Objet de la méthode
Recharger les données	`– (void) reloadComponent: (NSInteger)component`	Notifie au récepteur que les valeurs du composant ont été modifiées.
	`– (void) reloadAllComponents`	Notifie au récepteur que les valeurs de tous les composants ont été modifiées.
Sélection	`– (void) selectRow: (NSInteger)row inComponent: (NSInteger)component animated:(BOOL)animated`	Sélectionne une ligne pour un composant, éventuellement avec une animation visuelle.
	`– (NSInteger) selectedRowIn Component:(NSInteger) component`	Retourne la ligne sélectionnée pour un composant du sélectionneur.

Un contrôle qui n'en est pas un

Contrairement au sélectionneur de date, le sélectionneur standard n'est pas un contrôle. La classe `UIPickerView` ne dérive pas de la classe `UIControl`. On ne peut donc pas utiliser le mécanisme **cible-action** avec un sélectionneur standard. C'est le délégué qui est notifié des changements d'états.

Source de données

Un sélectionneur standard nécessite deux délégués :

- `delegate` qui joue le rôle de **délégué** traditionnel : contrôle des vues, positionnement, réponse aux actions, il doit répondre au protocole `<UIPickerViewDelegate>` ;
- `dataSource`, **source de données**, qui fournit des informations relatives aux valeurs à afficher et répond au protocole `<UIPicker ViewDataSource>`.

Dans l'application *Picker2*, le contrôleur de vue est à la fois, comme souvent, le délégué et la source de données du sélectionneur. Nous y avons donc défini :

- la méthode `–numberOfComponentsInPickerView:` qui retourne 1 ; nous informons le sélectionneur qu'il doit afficher un seul composant (un seul tambour) ;
- la méthode `–pickerView:numberOfRowsInComponent:` qui retourne `[valeurs count]` pour informer le sélectionneur que le composant doit comporter autant de lignes que d'éléments dans le tableau `valeurs` ; tableau qui contient la liste des pays à afficher ;
- la méthode `–pickerView:titleForRow:forComponent:` qui retourne la chaîne de caractères à afficher sur la rangée `row` : `[valeurs objectAtIndex:row]` ;
- la méthode `–pickerView:didSelectRow:inComponent:` qui est appelée lorsque l'utilisateur vient de manipuler l'un des tambours.

Ces méthodes admettent un paramètre `pickerView`, ce qui permet à un même délégué et une même source de données de gérer plusieurs sélectionneurs.

Tableau 6.7 : Méthodes du protocole UIPickerViewDataSource

Méthode	Objet de la méthode
`– (NSInteger) numberOfComponents InPickerView:(UIPickerView *) pickerView`	Doit retourner le nombre de composants du sélectionneur passé en paramètre.
`– (NSInteger) pickerView: (UIPickerView *)pickerView numberOfRowsInComponent: (NSInteger)component`	Doit retourner le nombre de lignes pour le composant et le sélectionneur passés en paramètres.

Tableau 6.8 : Méthodes du protocole UIPickerViewDelegate

Thème	Méthode	Objet de la méthode
Dimensions de la vue	`– (CGFloat) pickerView: (UIPickerView *)picker View rowHeightFor Component:(NSInteger) component`	Doit retourner la hauteur en pixels dans laquelle doit s'afficher une ligne pour le composant et le sélectionneur passés en paramètres.
	`– (CGFloat) pickerView: (UIPickerView *)picker View widthForComponent: (NSInteger)component`	Doit retourner la largeur en pixels dans laquelle doit s'afficher une ligne pour le composant et le sélectionneur passés en paramètres.

Tableau 6.8 : Méthodes du protocole UIPickerViewDelegate

Thème	Méthode	Objet de la méthode
Contenu de la vue Une de ces méthodes est obligatoire	`– (NSString *) pickerView:(UIPickerView *) pickerView titleForRow:(NSInteger)row forComponent:(NSInteger) component`	Retourne une chaîne de caractères à afficher sur la ligne du composant du sélectionneur passés en paramètres.
	`– (UIView *) pickerView:(UIPickerView *)pickerView viewForRow:(NSInteger)row forComponent:(NSInteger) component reusingView:(UIView *)view`	Retourne une Vue à afficher sur la ligne du composant du sélectionneur passé en paramètres. La vue passée en paramètre peut être réutilisée.
Sélection d'une ligne	`– (void) pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)row inComponent:(NSInteger)component`	L'utilisateur vient de sélectionner la ligne du composant du sélectionneur passé en paramètres.

Adapter le sélectionneur au contexte

Les protocoles `UIPickerViewDelegate` et `UIPickerViewDataSource` permettent une grande diversité d'utilisation des sélectionneurs.

Figure 6.10 : Sélectionneur à deux composants

Dans le sélectionneur représenté ici, le composant gauche contient une liste de pays et le composant droit une liste de villes. Il est souhaitable que la liste des villes change lorsque l'utilisateur change de pays.

Structure de données

Les données que nous devons manipuler ont une structure plus complexe que celles que nous avons manipulées jusqu'ici. Nous avons besoin :

- d'un tableau contenant la liste des pays ;
- pour chaque pays, d'un tableau contenant la liste des villes de ce pays.

En programmation Cocoa, lorsqu'on veut établir une mise en correspondance de deux listes d'objets – ici une liste de pays et une liste de listes de villes –, on utilise un **dictionnaire** de la classe `NSDictionary`. Le bon endroit pour initialiser cette structure de données est la méthode `–viewDidLoad` du contrôleur de vue :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   pays = [[NSArray alloc] initWithObjects:@"France",
       @"Allemagne",@"Italie",@"Espagne",@"Portugal",nil];
   villes = [[NSDictionary alloc] initWithObjects:
                  [NSArray arrayWithObjects:
                      [NSArray arrayWithObjects:@"Paris",
                         @"Marseille",@"Lyon",@"Toulouse",
                         @"Bordeaux",nil],
                      [NSArray arrayWithObjects:@"Berlin",
                         @"Hambourg",@"Munich",
                         @"Stuttgart",nil],
                      [NSArray arrayWithObjects:@"Rome",
                         @"Florence",@"Naples",@"Venise",
                         @"Milan",nil],
                      [NSArray arrayWithObjects:@"Madrid",
                         @"Barcelone",@"Séville",nil],
                      [NSArray arrayWithObjects:@"Lisbonne",
                         @"Porto",nil],nil]
                  forKeys:pays];
   self.paysChoisi = @"France";
   self.label.text =
   [[villes objectForKey:self.paysChoisi] objectAtIndex:0];
}
```

Les classes et méthodes utilisées seront expliquées plus loin dans ce chapitre.

Source de données

Nous devons écrire les 2 méthodes du protocole `UIPickerView DataSource` qui retournent le nombre de composants et le nombre de lignes pour chaque composant du sélectionneur.

Nous voulons afficher deux composants :

```
- (NSInteger)numberOfComponentsInPickerView:
                        (UIPickerView *)pickerView{
   return 2;
}
```

Le nombre de lignes est fixe pour le premier composant, c'est le nombre de pays. Il dépend du pays choisi pour le deuxième composant, il faudra donc penser à ajouter une propriété `paysChoisi` dans notre contrôleur de vue :

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
             numberOfRowsInComponent:(NSInteger)component{
   if (component == 0) {
      return [pays count];
   } else {
      return [[villes objectForKey:self.paysChoisi] count];
   }
}
```

Délégué

Dans le premier composant, le titre à afficher en fonction du numéro de ligne est le nom du pays. Dans le second composant, il faut afficher le nom de la ville et nous devons ici aussi prendre en compte le pays choisi.

```
- (NSString *)pickerView:(UIPickerView *)pickerView
             titleForRow:(NSInteger)row
            forComponent:(NSInteger)component{
   if (component == 0) {
      return [pays objectAtIndex:row];
   } else {
      return [[villes objectForKey:self.paysChoisi]
                                      objectAtIndex:row];
   }
}
```

Lorsque l'utilisateur change de pays (lorsqu'il touche le composant numéro 0), il faut :

- modifier la propriété `paysChoisi` du contrôleur de vue ;
- indiquer au sélectionneur que la liste des villes à afficher n'est plus la même.

```
- (void)pickerView:(UIPickerView *)pickerView
      didSelectRow:(NSInteger)row
       inComponent:(NSInteger)component{
   if (component == 0) {
      self.paysChoisi = [pays objectAtIndex:row];
      self.label.text = self.paysChoisi;
      [pickerView reloadComponent:1];
   } else {
      self.label.text = [[villes
       objectForKey:self.paysChoisi] objectAtIndex:row];
   }
}
```

Remarquez l'instruction `[pickerView reloadComponent:1];` qui indique au sélectionneur qu'il doit modifier le contenu du composant numéro 1 (la liste des villes).

Challenge

Vous avez maintenant tous les éléments pour réaliser l'application Picker3 qui présente un sélectionneur à deux tambours, un pour les pays et un pour les villes, et qui affiche dans le label la dernière sélection de l'utilisateur. Bien sûr, la liste des villes dépend du pays sélectionné.

Si vous souhaitez comprendre plus précisément le fonctionnement des tableaux et des dictionnaires, reportez-vous à la section suivante avant de réaliser l'application *Picker3*.

6.2. Utiliser les conteneurs Cocoa

Les **conteneurs** sont des structures de données de base, indispensables pour réaliser des applications au modèle de données complexe. Dans ce chapitre, nous décrivons :

- les **tableaux**, qui sont des instances de la classe `NSArray` ;
- les **dictionnaires**, qui sont des instances de la classe `NSDictionary`.

Ces structures de données sont particulièrement intégrées au langage *Objective-C* ; on peut les parcourir avec l'instruction `for in` :

```
NSString * pays;
NSArray * toutLesPays=[NSArray arrayWithObjects:@"France",
        @"Allemagne",@"Italie",@"Espagne",@"Portugal",nil];
for ( pays in toutLesPays ) {
   // la boucle est exécutée avec les éléments du tableau
}
```

Tableaux NSArray

Un **tableau** est une liste ordonnée de pointeurs vers d'autres objets, indexée par un entier compris entre *0* et *n-1* si le tableau contient *n* objets. Les différents objets peuvent être de classes différentes mais toutes les positions de *0* à *n-1* doivent être occupées ; il est interdit d'avoir un pointeur `nil` au milieu du tableau.

Objet nul

On ne peut pas insérer `nil` dans un conteneur mais on peut utiliser l'objet nul `[NSNull null]`. L'objet nul ne répond à aucune méthode excepté celles de `NSObject` dont il dérive.

Les méthodes principales de la classe `NSArray` sont résumées dans le tableau. Les méthodes les plus utilisées sont `–objectAtIndex:` qui retourne l'objet associé à l'indice passé en paramètre, et `–count` qui retourne le nombre d'éléments dans le tableau.

Tableau 6.9 : Principales méthodes de la classe NSArray

Thème	Méthode	Objet de la méthode
Créer un tableau	`+ (id) arrayWithObjects: (id)firstObj, ...`	Crée un tableau constitué de la liste des objets passés en paramètre. Les éléments de la liste sont séparés par une virgule. Le dernier élément de la liste doit être `nil`.
Initialiser un tableau	`– (id) initWithObjects: (id)firstObj, ...`	Initialise un tableau constitué de la liste des objets passés en paramètre. Les éléments de la liste sont séparés par une virgule. Le dernier élément de la liste doit être `nil`.
Interroger un tableau	`– (BOOL) containsObject: (id)anObject`	Retourne `YES` si `anObject` est dans le tableau, `NO` sinon. Les éléments du tableau sont comparés à `anObject` par la méthode `-isEqual:` déclarée dans `NSObject`.
	`– (NSUInteger)count`	Retourne le nombre d'éléments du tableau.
	`– (id)lastObject`	Retourne le dernier élément du tableau.
	`– (id) objectAtIndex: (NSUInteger)index`	Retourne l'élément dont l'indice est passé en paramètre. `index` doit être compris entre `0` et `count–1`.
Trouver un objet	`– (NSUInteger) index OfObject:(id)anObject`	Retourne l'indice de l'élément égal à `anObject`. Les éléments du tableau sont comparés à `anObject` par la méthode `-isEqual:`. Si plusieurs éléments sont égaux à `anObject`, l'indice le plus petit est retourné. Si aucun élément n'est égal à `anObject`, la méthode retourne `NSNotFound`.
	`– (NSUInteger) index OfObject:(id)anObject inRange:(NSRange)range`	Retourne l'indice de l'élément égal à `anObject`. Les éléments du tableau sont comparés à `anObject` par la méthode `-isEqual:`. Si plusieurs éléments sont égaux à `anObject`, l'indice le plus petit est retourné. Si aucun élément n'est égal à `anObject`, la méthode retourne `NSNotFound`. La recherche est limitée aux indices compris dans l'intervalle `range`.

Pour créer un intervalle de type `NSRange`, on peut employer la fonction utilitaire `NSMakeRange`, par exemple :

```
NSRange range = NSMakeRange(4, 8);
```

Dictionnaires NSDictionary

Un tableau permet de retrouver un objet par son indice qui est obligatoirement un entier. Il est parfois intéressant d'utiliser des indices quelconques, une chaîne de caractères par exemple. C'est le rôle d'un dictionnaire.

Un **dictionnaire** est une liste d'entrées. Chaque entrée est constituée :

- d'une **clé**, un objet quelconque, souvent de la classe `NSString` ;
- d'une valeur associée à la clé, également un objet quelconque.

Aucune entrée ne doit présenter de clé ou de valeur `nil` (l'objet nul est autorisé). Une clé doit être unique dans le dictionnaire.

Les méthodes principales de la classe `NSDictionary` sont résumées dans le tableau. Les méthodes les plus utilisées sont `–objectForKey:` qui retourne la valeur associée à la clé passée en paramètre et `–count` qui retourne le nombre d'éléments dans le dictionnaire.

Tableau 6.10 : Principales méthodes de la classe NSDictionary

Thème	Méthode	Objet de la méthode
Créer un dictionnaire	`+ (id) dictionaryWithObjects:(NSArray *)objects forKeys:(NSArray *)keys`	Crée un dictionnaire constitué des objets contenus dans le tableau `objects` avec les clés contenus dans le tableau `keys`. Les deux tableaux doivent contenir le même nombre d'éléments.
Initialiser un dictionnaire	`– (id) initWithObjects:(NSArray *)objects forKeys:(NSArray *)keys`	Initialise un dictionnaire constitué des objets contenus dans le tableau `objects` avec les clés contenus dans le tableau `keys`. Les deux tableaux doivent contenir le même nombre d'éléments.
Accéder aux clés et aux valeurs	`– (NSUInteger)count`	Retourne le nombre de paires (clé, objet) du dictionnaire.
	`– (id) objectForKey:(id)aKey`	Retourne l'objet associé à la clé `aKey`, ou `nil` si la clé n'est pas dans le dictionnaire.
	`– (NSArray *)allKeys`	Retourne un tableau constitué de l'ensemble des clés du dictionnaire, ou un tableau vide si le dictionnaire est vide.

Conteneurs mutables

Les objets des classes `NSArray` et `NSDictionary` sont **immuables** ; une fois créés, on ne peut les modifier. Il existe des versions modifiables (*mutable*) de ces classes qui, en pratique, sont peu utilisées.

La classe `NSMutableArray` dérive de `NSArray`. Elle définit en particulier les méthodes supplémentaires suivantes :

- `-addObject:` qui permet d'ajouter un élément à la fin du tableau ;
- `-insertObject:atIndex:` qui permet d'insérer un élément dans le tableau ;
- `-removeObjectAtIndex:` pour supprimer un élément connaissant son indice ;
- `-replaceObjectAtIndex:withObject:` pour remplacer un des éléments du tableau.

Les méthodes les plus utilisées de la classe `NSMutableDictionary` sont :

- `-setObject:forKey:` qui permet d'ajouter une entrée dans le dictionnaire ;
- `-removeObjectForKey:` pour supprimer une entrée connaissant sa clé.

6.3. Utiliser les Vues en table

La **vue en table** (*TableView*) est le principal outil de navigation au sein d'une structure de données arborescente. Vous avez maintenant les connaissances suffisantes pour en comprendre le fonctionnement et pour la mettre en œuvre dans vos applications :

- pattern Modèle-Vue-Contrôleur ;
- délégation ;
- source de données ;
- conteneurs.

Pour illustrer le fonctionnement d'ensemble des classes mises en jeu, nous créerons l'application *Emprunts1*. Il s'agit d'un aide-mémoire pour nous souvenir des objets que nous avons prêtés à nos amis. À qui les avons-nous prêtés et à quelle date ?

Présentation générale

Avec les vues en table, l'utilisateur visualise une liste de données puis il choisit un élément de cette liste pour visualiser une autre liste liée à cet élément, etc. Il parcourt ainsi la structure de données, visualise les informations détaillées ou édite les données.

L'exemple typique de ce mode de navigation est l'application *Contacts* :

- L'application présente une liste de groupes.
- L'utilisateur choisit un groupe ; il visualise alors la liste des contacts appartenant à ce groupe.
- Il parcourt la liste des contacts du groupe. Lorsqu'il en choisit un, il en visualise les informations détaillées qu'il peut éditer s'il le souhaite.

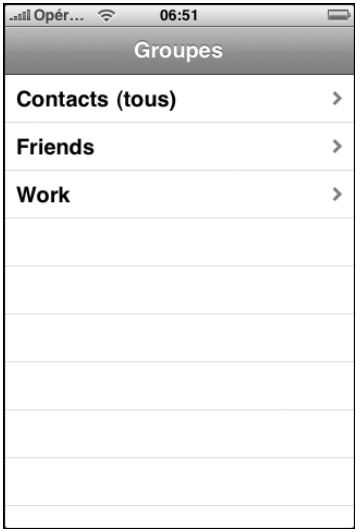


Figure 6.11 : Liste des groupes de contacts

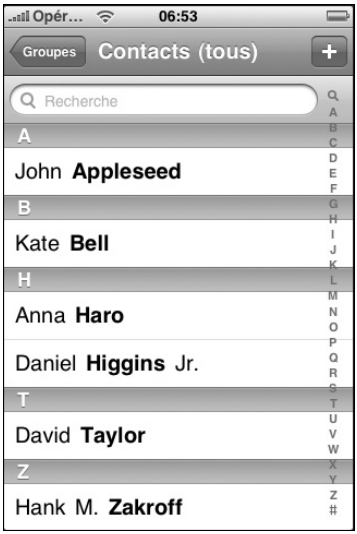


Figure 6.12 : Liste des contacts

Figure 6.13 : Informations détaillées d'un contact

Chacune de ces vues est une vue en table de style différent :

- vue en table simple ;
- vue en table indexée (par les lettres de l'alphabet) ;
- vue en table par groupe (téléphones, adresses de courriel, adresses postales, etc.).

Le fonctionnement de l'application *Emprunts1* est analogue à celui de *Contacts* :

- Le sommet de la structure de données est la liste des catégories d'objet (CD, DVD, Livre).
- Lorsqu'une catégorie et choisie, l'utilisateur accède à la liste des objets de cette catégorie.
- Il peut ajouter un objet ou éditer les informations pour un objet.

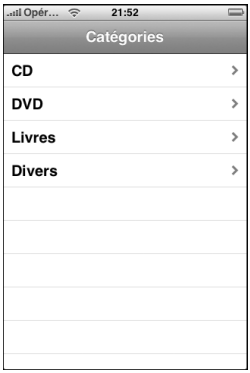


Figure 6.14 : Application Emprunts1

Paradoxalement, nous utiliserons peu les objets de la classe `UITableView` pour la programmation des vues en table. Nous manipulerons surtout les protocoles et classes associés :

- la classe `UITableView`, bien sûr ;
- la classe `UITableViewController`, que nous dériverons et qui répond aux protocoles :
  - `<UITableViewDelegate>` ;
  - `<UITableViewDataSource>`.
- la classe `UITableViewCell`, qui est la vue devant être affichée dans une **cellule** ou ligne de la table ;
- la classe `NSIndexPath` dont les instances permettent de repérer une cellule particulière de la table.

La classe `UITableViewCell` méritera une attention particulière car elle permet toute la richesse d'affichage de la vue en table.

La programmation d'une vue en table consiste principalement à définir un contrôleur de vue qui hérite de `UITableViewController`. Cette classe prend en charge, en particulier, la présentation générale de la table et le défilement vertical lorsque la liste ne tient pas entièrement sur l'écran.

Créer une vue en table

Pour créer une vue en table, il suffit de créer et d'initialiser un contrôleur de vue qui hérite de la classe `UITableViewController`.

Par programmation, l'initialisation du contrôleur de vue crée automatiquement la vue en table :

- pour créer une vue en table simple :
  - `init` ;
  - `initWithStyle:UITableViewStylePlain`.
- pour créer une vue en table par groupe :
  - `initWithStyle:UITableViewStyleGrouped`.

Sous *Interface Builder*, il faut faire glisser un objet de type *Table View Controller* dans la fenêtre du document NIB. Le contrôleur de vue ainsi ajouté contient une instance de la classe `UITableView`.

Sous XCode, lorsqu'on crée une application de type *Navigation-based*, une barre de navigation et un contrôleur de vue en table sont automatiquement créés.

La vue en table associée au contrôleur est accessible par sa propriété `tableView` définie dans la classe `UITableViewController`.

1 Ouvrez *XCode* et créez une application de type *Navigation-based*.

2 Laissez la case **Use Core Data for storage** décochée et nommez l'application *Emprunts1*.

Afficher la table

Préparer la structure de données

La classe `RootViewController` est le contrôleur de vue racine, elle dérive de `UITableViewController`. C'est cette classe que nous allons modifier pour apporter le comportement souhaité à notre application.

Nous commencerons par créer le tableau des catégories à afficher sur la première table.

1 Ajoutez une propriété `categories` dans l'interface de la classe `RootViewController` :

```
@interface RootViewController : UITableViewController {
   NSArray * categories;
}
@property(nonatomic,retain) NSArray * categories;
@end
```

2 Modifiez la méthode `–viewDidLoad` dans le fichier *RootViewController.m* pour initialiser le tableau des catégories :

```
@synthesize categories;
- (void)viewDidLoad {
   [super viewDidLoad];
   self.categories = [NSArray arrayWithObjects: @"CD",
                      @"DVD",@"Livres",@"Divers",nil];
}
```

3 Enlevez les marques de commentaires autour de la méthode `–viewDidLoad`.

Dimensionner la table

À l'instar du sélectionneur standard, la vue en table interroge sa **source de données** pour dimensionner son affichage.

Si la table est de style groupé, la méthode `–numberOfSections InTableView:` doit être implémentée dans le contrôleur de vue et retourner le nombre de sections. La table des catégories de l'application *Emprunts1* n'étant pas décomposée en section, vérifiez que cette méthode retourne `1`.

La méthode `-tableView:numberOfRowsInSection:` est obligatoire. Modifiez-la pour retourner le nombre de lignes dans la table (le nombre de catégories) :

```
- (NSInteger)tableView:(UITableView *)tableView
                numberOfRowsInSection:(NSInteger)section {
   return [self.categories count];
}
```

Afficher les titres

Le titre inscrit dans la barre de navigation est, par défaut, le titre du contrôleur de vue actif. Modifiez la méthode `-viewDidLoad` pour indiquer le titre *Catégories*.

```
- (void)viewDidLoad {
   [super viewDidLoad];
   self.categories = [NSArray arrayWithObjects: @"CD",
                      @"DVD",@"Livres",@"Divers",nil];
   self.title = @"Catégories";
}
```

Si la table est créée en mode Groupé et si l'on souhaite donner un titre à chaque section, il faut que le contrôleur de vue implémente la méthode `-tableView:titleForHeaderInSection:` qui retourne une instance de `NSString`.

Délégué et Source de données

Dans un but de simplification, nous écrivons qu'un contrôleur de vue en table doit implémenter telle ou telle méthode sans préciser si cette méthode est définie dans le protocole de **délégué** ou dans celui de **source de donnée** de la vue en table.

Décrire une ligne

La méthode retenue par Apple pour afficher une ligne de la table répond à plusieurs problématiques :

- Il faut pouvoir repérer la ligne en question, lui donner une identification.
- Il ne faut pas brider le développeur dans sa créativité.
- Il faut que la programmation d'affichages simples reste simple.
- La mémoire est limitée et il faut tenir compte du fait que seules quelques lignes sont réellement affichées à l'écran à un instant donné.

Repérer la ligne

Une ligne dans une table est repérée par une instance de la classe `NSIndexPath`. Cette classe fournit deux propriétés `section` et `row` qui permettent d'identifier une **ligne** (*row*) dans une **section** de la table (*section*).

Avec une table ne contenant qu'une section, seule la propriété `row` est utilisée. La section et la ligne dans la section sont numérotées à partir de `0`. (La première section a le numéro `0`.)

NSIndexPath

Les propriétés `section` et `row` sont définies dans une extension de la classe `NSIndexPath` qui la rend plus facile à utiliser avec les vues en table.

Cellules de table

Chaque ligne de la table est affichée dans une vue de type `UITableViewCell`, appelée **cellule** (*cell*). Cette classe définit un comportement par défaut qui facilite la programmation dans les cas simples. Elle peut aussi être dérivée ou peut inclure d'autres vues ou d'autres contrôles afin d'obtenir un comportement enrichi.

C'est le contrôleur de vue de la table qui est chargé de fabriquer les cellules pour chaque ligne à la demande de la vue en table. Le contrôleur doit implémenter la méthode `-tableView:cellForRowAtIndexPath:` et retourner la cellule initialisée qui doit être affichée sur la ligne repérée par le paramètre `indexPath`.

Nous allons avancer dans la compréhension des cellules avant de voir un exemple pratique de mise en œuvre.

Recycler les cellules

À mesure que l'utilisateur parcourt la liste de la vue en table, de nouvelles lignes sont affichées et d'autres disparaissent de l'écran. Nous venons de voir que lorsqu'une ligne est sur le point d'apparaître à l'écran, la vue en table demande à son contrôleur de créer une cellule et de l'initialiser. Mais que se passe-t-il lorsqu'une ligne disparaît de l'écran ?

La mémoire étant limitée, on ne peut se permettre de toutes les conserver, mais ce serait dommage de toutes les détruire. Si une ligne disparaît, c'est qu'une autre apparaît à l'écran et qu'il faudra créer une cellule pour cette nouvelle ligne. Cela prend du temps de

créer une cellule et il faut aussi économiser la batterie. L'idée est donc de *recycler* les cellules qui disparaissent de l'écran.

Les instances de `UITableView` entretiennent à cet effet une liste de cellules réutilisables. On peut obtenir une cellule en appelant la méthode `–dequeueReusableCellWithIdentifier:`. Cette méthode retourne `nil` s'il n'y a pas de cellule réutilisable. L'identifiant passé en paramètre est une chaîne de caractères. Si la vue en table contient des cellules de différents types, il est important de repérer chaque type par un *identifiant* spécifique.

L'identifiant d'une cellule est défini lors de sa création, à l'aide de la méthode `–initWithStyle:reuseIdentifier:` de la classe `UITableViewCell`.

C'est une bonne pratique de toujours vérifier s'il n'existe pas une cellule réutilisable avant d'en créer une nouvelle du même type. Examinez la méthode `–tableView:cellForRowAtIndexPath:` de la classe `RootViewController`. Le recyclage des cellules y est déjà prévu, le développeur n'a plus qu'à saisir le code pour configurer la cellule :

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
   static NSString *CellIdentifier = @"Cell";
   UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
   if (cell == nil) {
      cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
              reuseIdentifier:CellIdentifier] autorelease];
   }
   // Configure the cell.
   return cell;
}
```

Classe UITableViewCell

La méthode d'initialisation de la classe `UITableViewCell` prend un paramètre `identifier` qui permet de définir l'identifiant de la cellule pour sa réutilisation, et un paramètre `style` qui définit son style de présentation à l'écran. Les styles les plus utilisés sont :

- `UITableViewCellStyleDefault`, style par défaut, une ligne de texte avec une image optionnelle ;
- `UITableViewCellStyleSubtitle`, style avec sous-titre, deux lignes de texte et une image optionnelle.

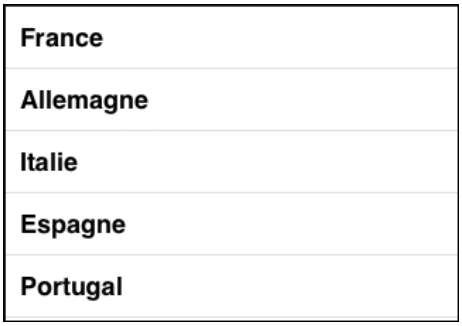


Figure 6.15 : Style par défaut sans image

Figure 6.16 : Style par défaut avec image

Figure 6.17 : Style avec sous-titre et image

Ces différents éléments de la cellule sont accessibles par les propriétés de la classe `UITableViewCell` :

- `textLabel`, propriété de type `UILabel *` qui contient le texte principal de la cellule ;
- `detailTextLabel`, propriété de type `UILabel *` qui contient le texte secondaire de la cellule (seulement si la cellule est de style avec sous-titre) ;
- `imageView`, propriété de type `UIImageView *` qui contient l'image affichée à gauche de la cellule.

Il est également possible d'agrémenter chaque cellule d'un **accessoire** qui s'affiche sur la droite. Il s'agit d'une icône qui indique à l'utilisateur les opérations qu'il peut réaliser. Le type d'accessoire à afficher est indiqué avec la propriété `accessoryType` de la classe `UITableViewCell`. Les valeurs autorisées pour cette propriété sont précisées dans le tableau.

Tableau 6.11 : Valeurs autorisées pour la propriété accessoryType

Icône	Valeur de la propriété	Utilisation
	`UITableViewCellAccessory DisclosureIndicator`	Indique qu'une touche sur la ligne permet d'accéder à des informations plus détaillées.
	`UITableViewCellAccessory DetailDisclosureButton`	Indique qu'une touche sur l'accessoire permet d'accéder à des informations plus détaillées.
	`UITableViewCellAccessory Checkmark`	Indique que la ligne est sélectionnée.
	`UITableViewCellAccessory None`	Indique que la ligne n'est pas sélectionnée et ne contient pas d'accessoire spécifique.

Nous voulons afficher les catégories d'objets et que l'utilisateur accède à la liste des objets de cette catégorie lorsqu'il touche la ligne correspondante.

1 Modifiez la méthode `-tableView:cellForRowAtIndexPath:` du fichier *RootViewController.m*.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
       dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                 initWithStyle:UITableViewCellStyleDefault
             reuseIdentifier:CellIdentifier] autorelease];
    }
    cell.textLabel.text = [categories
                           objectAtIndex:indexPath.row];
    cell.accessoryType =
             UITableViewCellAccessoryDisclosureIndicator;
    return cell;
}
```

2 Construisez l'application et testez-la. L'affichage est correct mais il faut maintenant que l'application réagisse lorsque l'utilisateur touche une catégorie.

Réagir à une sélection

Préparer la structure de données

Comme nous l'avons fait lors de notre étude des sélectionneurs, avec les pays et les villes, nous allons représenter les listes d'objets prêtés dans un dictionnaire indexé par la catégorie d'objet. Au départ, les listes d'objets sont vides.

1 Ajoutez une propriété `lendObjects` dans l'interface de la classe `RootViewController` :

```
@interface RootViewController : UITableViewController {
   NSArray * categories;
   NSDictionary * lendObjects;
}
@property(nonatomic,retain) NSArray * categories;
@property(nonatomic,retain) NSDictionary * lendObjects;
@end
```

2 Modifiez la méthode `–viewDidLoad` dans le fichier *RootViewController.m* pour initialiser le dictionnaire des objets prêtés. Nous employons ici des tableaux modifiables de la classe `NSMutableArray` car l'utilisateur doit pouvoir modifier les listes d'objets :

```
@synthesize categories,lendObjects;
- (void)viewDidLoad {
   [super viewDidLoad];
   self.title = @"Catégories";
   self.categories = [NSArray
    arrayWithObjects:@"CD",@"DVD",@"Livres",@"Divers",nil];
   self.lendObjects = [NSDictionary dictionaryWithObjects:
          [NSArray arrayWithObjects:[NSMutableArray array],
          [NSMutableArray array],[NSMutableArray array],
                              [NSMutableArray array],nil]
            forKeys:self.categories];
}
```

Afficher la liste détaillée

Lorsque l'utilisateur touche une cellule de la vue en table, le contrôleur de vue reçoit un message `–tableView:didSelectRowAtIndexPath:`. Modifiez cette méthode pour y créer un nouveau contrôleur de vue et l'afficher à l'écran. Ce contrôleur de vue doit connaître la liste d'objets à afficher, il aura donc aussi une propriété `lendObjects` :

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
   ObjectListViewController * objectListViewController =
             [[ObjectListViewController alloc]
```

```
                    initWithStyle:UITableViewStylePlain];
    NSString * category =
              [self.categories objectAtIndex:indexPath.row];
    objectListViewController.title = category;
    objectListViewController.lendObjects =
                  [self.lendObjects objectForKey:category];
    [self.navigationController pushViewController:
                     objectListViewController animated:YES];
    [objectListViewController release];
}
```

Si l'accessoire ajouté dans les cellules avait été du type `UITableViewCellAccessoryDetailDisclosureButton`, ce code aurait du être placé dans la méthode `–tableView:accessoryButtonTappedForRowWithIndexPath:`. Cette méthode est appelée lorsque l'accessoire est un bouton et qu'il est touché par l'utilisateur.

Il faut maintenant créer la classe `ObjectListViewController`. Créez une nouvelle classe sous *XCode* (⌘+N), choisissez un contrôleur de vue qui dérive de `UITableViewController`.

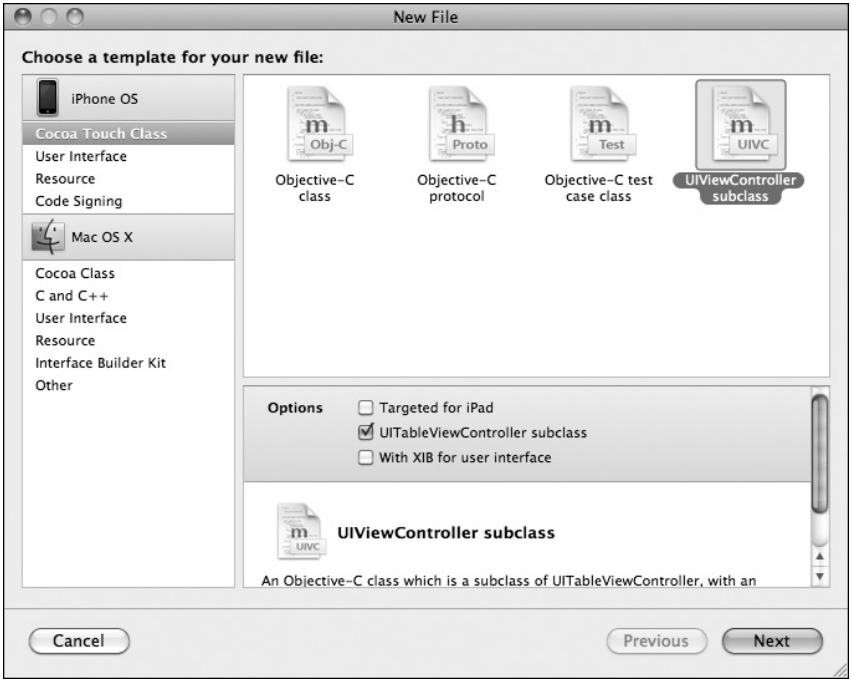


Figure 6.18 : Création d'un contrôleur dérivant de UITableViewController

1 Ajoutez la propriété `lendObjects` dans l'interface de la classe :

```
@interface ObjectListViewController : UITableViewController{
   NSMutableArray * lendObjects;
}
@property(nonatomic,retain) NSMutableArray * lendObjects;
@end
```

2 Modifiez le fichier *ObjectListViewController.m* afin de prendre en compte cette propriété pour l'affichage de la table. Laissez inchangées les autres méthodes du fichier.

```
@synthesize lendObjects;
- (void)viewDidUnload {
   self.lendObjects = nil;
}
- (NSInteger)numberOfSectionsInTableView:
                              (UITableView *)tableView {
   return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section {
   return [lendObjects count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
   static NSString *CellIdentifier = @"Cell";
   UITableViewCell *cell = [tableView
       dequeueReusableCellWithIdentifier:CellIdentifier];
   if (cell == nil) {
      cell = [[[UITableViewCell alloc] initWithStyle:
             UITableViewCellStyleDefault reuseIdentifier:
                         CellIdentifier] autorelease];
    }
   cell.textLabel.text = [[lendObjects
               objectAtIndex:indexPath.row] description];
   return cell;
}
```

3 Ajoutez l'importation de l'interface de la classe `ObjectListViewController` dans le fichier *RootViewController.m* :

```
#import " ObjectListViewController.h"
```

4 Reconstruisez l'application et testez-la (voir Figure 6.19).

Les listes sont vides, ce qui était prévu, et un bouton de retour est automatiquement ajouté par le contrôleur de navigation. Vous pouvez déjà naviguer dans la structure de données.

Il ne nous reste plus qu'à doter notre application d'une fonction pour ajouter des objets dans les listes.

Figure 6.19 : La liste des objets prêtés est vide

Ajouter un élément

Classe LendObject

Nous avons besoin d'une classe pour représenter un objet prêté et ses propriétés :

- le nom de l'objet ;
- le nom de l'emprunteur ;
- la date du prêt.

Cette classe est seulement un réceptacle de données, elle ne contient pas d'autre méthode que les accesseurs de ses propriétés.

1 Sous XCode, créez une nouvelle classe `LendObject` qui dérive de `NSObject`. Déclarez les propriétés dans le fichier *LendObject.h* :

```
@interface LendObject : NSObject {
   NSString * objectName;
   NSString * borrowerName;
   NSDate * lendDate;
}
@property(nonatomic,retain) NSString * objectName;
@property(nonatomic,retain) NSString * borrowerName;
@property(nonatomic,retain) NSDate * lendDate;
@end
```

2 Modifiez le fichier *LendObject.m* pour y définir les accesseurs de propriétés et la méthode `-dealloc`.

```
@implementation LendObject
@synthesize objectName, borrowerName, lendDate;
- (void)dealloc {
    self.objectName = nil;
    self.borrowerName = nil;
    self.lendDate = nil;
    [super dealloc];
}
@end
```

Contrôleur de vue LendObjecViewController

Il nous faut maintenant une Vue et un contrôleur de vue pour éditer les propriétés d'une instance de la classe `LendObject`. Sous *XCode*, créez une nouvelle classe qui dérive de `UIViewController`, décochez la case *UITableViewController subclass* et cochez la case *With XIB for user interface*. Nous aurons besoin en effet d'un fichier NIB pour décrire l'interface utilisateur.

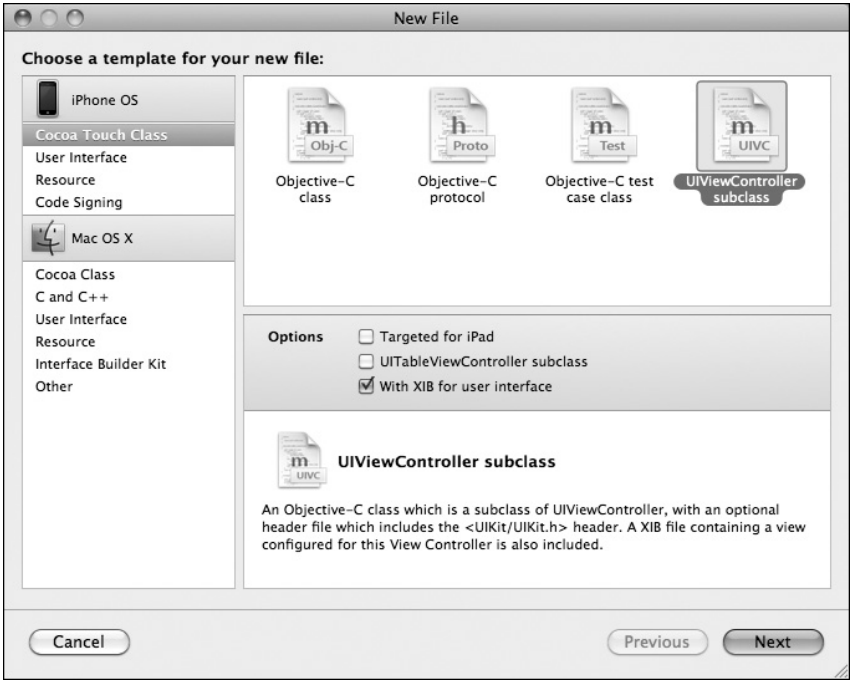


Figure 6.20 : Création de la classe LendObjectViewController

Déclaration de l'interface

1. Intitulez le fichier *LendObjectViewController.m* puis ouvrez le fichier *LendObjectViewController.h* pour déclarer les outlets et les propriétés.
2. Déclarez une action `–doneEditing:` qui nous servira pour effacer le clavier.

```
#import <UIKit/UIKit.h>
#import "LendObject.h"
@interface LendObjectViewController : UIViewController {
   LendObject * lendObject;
   IBOutlet UITextField * objectNameField;
   IBOutlet UITextField * borrowerNameField;
   IBOutlet UIDatePicker * datePicker;
}
@property(nonatomic,retain) LendObject * lendObject;
@property(nonatomic,retain) UITextField * objectNameField;
@property(nonatomic,retain) UITextField *borrowerNameField;
@property(nonatomic,retain) UIDatePicker * datePicker;
- (IBAction) doneEditing:(id)sender;
@end
```

Définition des méthodes

1. Ouvrez le fichier *LendObjectViewController.m* pour définir les accesseurs des outlets et propriétés. Modifiez la méthode `–viewDidUnload` pour libérer les outlets et les propriétés.
2. Créez l'action `–doneEditing:` pour effacer le clavier comme nous l'avons fait précédemment et créez la méthode `–viewWillDisappear:`. Cette dernière méthode est définie dans la classe `UIViewController`, elle est appelée lorsque la vue va disparaître de l'écran ; c'est le bon endroit pour prendre en compte la saisie effectuée par l'utilisateur.

```
@synthesize lendObject, objectNameField, borrowerNameField,
                                                    datePicker;
- (void)viewDidUnload {
   self.lendObject = nil;
   self.objectNameField = nil;
   self.borrowerNameField = nil;
   self.datePicker = nil;
}
- (void)viewWillDisappear:(BOOL)animated{
   self.lendObject.objectName = self.objectNameField.text;
   self.lendObject.borrowerName=self.borrowerNameField.text;
   self.lendObject.lendDate = self.datePicker.date;
   [super viewWillDisappear:animated];
}
```

```
- (IBAction) doneEditing:(id)sender {
    [sender resignFirstResponder];
}
```

Le retour vers l'écran précédent sera pris en charge par le contrôleur de navigation ; nous n'avons pas besoin de nous en occuper ici.

Création du fichier NIB

Ouvrez le fichier *LendObjectViewController.xib*. Disposez les contrôles pour bâtir l'interface utilisateur de saisie d'un prêt.

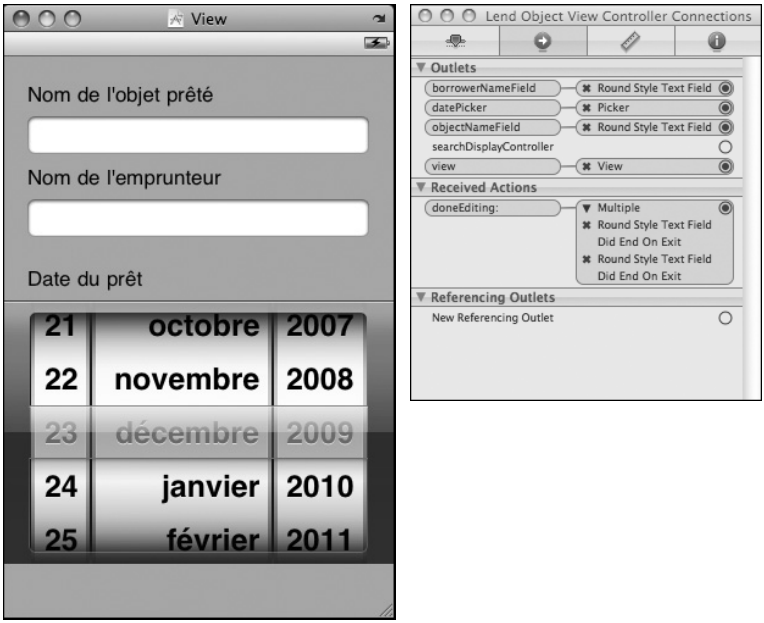


Figure 6.21 : Interface de saisie pour le prêt d'un objet

Établissez les connexions avec les outlets et l'action du contrôleur de vue. Vous pouvez en profiter pour retoucher quelques paramètres des contrôles pour la saisie :

- augmenter la taille de caractère des champs de texte ;
- donner la valeur *Done* à la clé **Return** des champs de texte ;
- régler le sélectionneur de date en mode Date.

Nous avons terminé l'interface utilisateur pour la saisie des nouveaux prêts. Il nous reste à programmer la fonction pour accéder à cette saisie depuis la liste des objets prêtés (toujours vide, pour l'instant).

Activer la fonction d'ajout

L'activation de la fonction d'ajout nécessite deux éléments :

- un bouton pour permettre à l'utilisateur de l'actionner ;
- une action connectée sur ce bouton pour activer l'interface utilisateur de saisie.

Ajoutez une action dans le fichier *ObjectListViewController.h* :

```
- (IBAction) addItem;
```

La barre de navigation de l'interface utilisateur est actuellement occupée :

- à gauche par le bouton de retour ;
- au centre par le titre de l'écran.

Il nous reste une place à droite pour le bouton d'ajout. Les boutons de la barre de navigation sont de la classe `UIBarButtonItem`, ils peuvent comporter une image et un titre. Une vingtaine de boutons sont prédéfinis dans *Cocoa Touch* que vous pouvez visualiser sous *Interface Builder*. Nous utiliserons le bouton d'ajout standard représenté par le signe plus.

Bouton d'ajout

Ajoutez une méthode `–viewDidAppear:` dans le fichier *ObjectListViewController.m* pour y créer un bouton de barre de navigation et l'ajouter à droite. Le bouton est connecté à l'action `addItem` lors de sa création. La méthode `–viewDidAppear:` est définie dans la classe `UIViewController`, elle est appelée lorsque la vue vient de s'afficher à l'écran.

```
- (void)viewDidAppear:(BOOL)animated {
   [super viewDidAppear:animated];
   UIBarButtonItem * addButton = [[UIBarButtonItem alloc]
      initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                  target:self action:@selector(addItem)];
   self.navigationItem.rightBarButtonItem = addButton;
   [addButton release];
}
```

Action addItem

Nous allons maintenant définir l'action `addItem` toujours dans le fichier *ObjectListViewController.m*. Il nous faut :

- créer un contrôleur de vue de la classe `LendObjectViewController`, celle que nous venons de coder pour s'occuper de la saisie ;
- charger le fichier NIB correspondant à cette classe ;

- créer un nouvel objet de la classe `LendObjet` ;
- ajouter ce nouvel objet à la liste des objets prêtés ;
- transmettre ce nouvel objet au contrôleur de vue pour qu'il en effectue la saisie ;
- afficher la vue pour la saisie.

1 Déclarez la classe que nous allons utiliser en tête du fichier :

```
#import "LendObjectViewController.h"
```

2 Créez la méthode `–addItem` :

```
- (void)addItem{
    LendObjectViewController * itemViewController =
          [[LendObjectViewController alloc] initWithNibName:
                @"LendObjectViewController" bundle:nil];
    LendObject * newLendObject = [[LendObject alloc] init];
    itemViewController.lendObject = newLendObject;
    [self.lendObjects addObject:newLendObject];
    [self.navigationController pushViewController:
                        itemViewController animated:YES];
    [itemViewController release];
}
```

3 Construisez l'application et testez-la. Tout semble fonctionner et pourtant, l'objet créé n'apparaît pas dans la liste lorsque l'utilisateur revient de l'écran de saisie ; il faut mettre la liste à jour à ce moment-là.

Mise à jour de la liste

Il faut transmettre un message `reloadData` à la vue en table pour lui indiquer que la liste a évolué.

1 Ajoutez une méthode `–viewWillAppear:` dans le fichier *ObjectListViewController.m* :

```
- (void)viewWillAppear:(BOOL)animated {
    [self.tableView reloadData];
    [super viewWillAppear:animated];
}
```

2 Construisez l'application et testez-la. La liste est mise à jour, en tout cas il se passe quelque chose, mais le résultat n'est pas très esthétique. Nous allons améliorer cela.

Améliorer l'affichage

Nous emploierons le style de cellule avec un sous-titre pour que l'utilisateur puisse voir :

- le nom de l'objet prêté en titre de cellule ;

- le nom de l'emprunteur et la date d'emprunt en sous-titre.

1 Modifiez la méthode `–tableView:cellForRowAtIndexPath:` dans le fichier *ObjectListViewController.m* :

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
          dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
       cell = [[[UITableViewCell alloc] initWithStyle:
                      UITableViewCellStyleSubtitle
              reuseIdentifier:CellIdentifier] autorelease];
    }
    LendObject * lendObject = [lendObjects
                           objectAtIndex:indexPath.row];
    NSDateFormatter * formatter =
                         [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"dd MMMM yyyy"];
    NSString * subTitle = [NSString stringWithFormat:
           @"prêté à %@ le %@",lendObject.borrowerName,
           [formatter stringFromDate:lendObject.lendDate]];
    [formatter release];
    cell.textLabel.text = [lendObject objectName];
    cell.detailTextLabel.text = subTitle;
    return cell;
}
```

2 Construisez l'application et testez-la. Cette fois, c'est parfait.

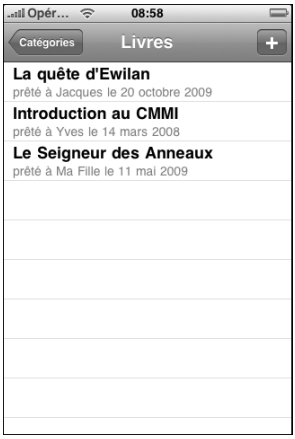


Figure 6.22 : Liste des objets prêtés

Subsiste un petit problème… Il ne faut pas quitter l'application. Autrement, nous perdons tout ce que nous avons saisi. Nous allons régler cela dès le chapitre suivant.

Pour aller plus loin

Nous n'avons malheureusement qu'effleuré le vaste sujet des vues en table, de leur cellules et de la navigation entre les vues. Pour approfondir ces sujets, nous vous invitons à consulter la documentation d'Apple soit sur le site des développeurs, soit directement sous XCode : sélectionnez la commande *Class Browser* du menu **Project** (Maj+⌘+C) puis sélectionnez l'option *Flat, all classes*.

Même si vous ne lisez pas l'anglais, vous pouvez consulter la liste des propriétés et des méthodes disponibles dans chaque classe et procéder à vos propres expérimentations.

Nous vous proposons quelques challenges par difficulté croissante.

Challenges

Challenge 1

Positionner le sélectionneur de date à la date du jour lors de l'affichage de l'écran de saisie d'un nouvel objet.

Challenge 2

Autoriser la modification des prêts déjà saisis. Le même écran peut être utilisé pour la saisie d'un nouveau prêt ou une modification.

Challenge 3

Ne pas autoriser la sortie de l'édition tant que le nom de l'objet et celui de l'emprunteur ne sont pas saisis. Prévoir un bouton d'annulation qui permette de revenir à la liste sans créer d'objet supplémentaire.

Challenge 4

Autoriser la suppression d'un élément d'une liste.

6.4. Checklist

Nous avons mis en œuvre dans ce chapitre des contrôles textuels plus complexes, les sélectionneurs et les vues en table, ainsi que la navigation entre les vues.

Nous avons détaillé le fonctionnement des sélectionneurs de date de la classe `UIDatePicker` et les classes d'objets qui permettent le traitement des dates et leur localisation :

- `NSDate` ;
- `NSDateFormatter` ;
- `NSTimeInterval`.

Nous avons vu comment programmer un sélectionneur standard et les protocoles qui accompagnent la classe `UIPickerView` :

- `UIPickerViewDelegate` ;
- `UIPickerViewDataSource`.

Nous avons exploré les principaux conteneurs utilisés en Objective-C : les tableaux `NSArray` et les dictionnaires `NSDictionary` ainsi que leur version modifiable.

Notre parcours nous a menés enfin vers les vues en table qui permettent la navigation dans des structures de données complexes :

- la classe `UITableView` ;
- les protocoles associés `UITableViewDelegate` et `UITableViewDataSource` ;
- le contrôleur de vue de la classe `UITableViewController` qui prend en charge ces protocoles ;
- les possibilités d'affichage des lignes avec la classe `UITableViewCell`.

Nous avons mis en œuvre ces techniques pour construire l'application *Emprunts1* qui est presque fonctionnelle. Il ne lui manque plus que la mémoire ; une application qui oublie tout dès qu'on la quitte n'est pas très utile. Nous la doterons de souvenance dès le prochain chapitre.

PERSISTANCE DES DONNÉES

Sur un ordinateur, les opérations de sauvegarde des données d'une application sont souvent explicitement demandées par l'utilisateur, qui peut préciser un nom de fichier. Sur un iPhone, l'utilisateur recherche l'immédiateté. Les données doivent s'enregistrer dès que l'application se termine, à l'occasion de la prise d'un appel entrant par exemple, et l'utilisateur souhaite retrouver l'application telle qu'il l'a laissée.

Ce chapitre est consacré à quelques techniques d'enregistrement et de récupération des données utilisées sur iPhone OS :

- *Core Data* est une technologie destinée à gérer des ensembles de données élaborés ; elle prend en charge leur *persistance*.
- Les *Listes de Propriétés* sont une technique très élégante pour conserver de petits ensembles de données.

À l'issue de ce chapitre, nous aurons doté nos applications *Convert-Pro* et *Emprunts* de la persistance des données. Ce sera également l'opportunité de découvrir le motif *Notification* qui est une technique importante de la programmation Cocoa.

7.1. Utiliser le framework Core Data

Core Data comprend un outil de description d'un modèle de données, équivalent à ce que l'on trouve sous Access ou 4D, et un ensemble de classes permettant de manipuler les données modélisées. On peut voir Core Data comme l'encapsulation d'une base de données *SQLite* dans des objets Objective-C. Le framework masque au programmeur la complexité de gestion d'une base de données ; il n'a besoin de connaître ni le langage *SQL*, ni l'administration des bases de données, ni le format d'enregistrement des données, ni l'entretien d'un cache mémoire. Core Data prend tout cela en charge.

SQLite

SQLite est un gestionnaire de base de données léger du domaine public, écrit en C ANSI ; le code est donc portable sur différentes plateformes et systèmes d'exploitation. Une base de données SQLite tient dans un fichier unique, lui-même portable. Outre Cocoa, ce gestionnaire est utilisé en Python, PHP, dans Firefox et il est disponible dans de nombreuses distributions Gnu/Linux.

Décrire le modèle de données

Entités, attributs et relations

Dans la programmation sans Core Data, la partie *Modèle* (au sens du motif MVC) de l'application est constituée d'un ensemble de classes d'objets définies par le développeur. Avec *Core Data*, le développeur décrit le **modèle de données** dans un fichier spécifique au format *xcdatamodel*, ce fichier sera ensuite exploité par le framework pour gérer les données, les enregistrer et les retrouver ; le travail du développeur est grandement facilité.

Dans le modèle de données, le développeur définit des **entités** (*entity*), l'équivalent des *classes* d'objet, puis les *propriétés* de chaque entité. Suivant leur nature, les *propriétés* sont des **attributs** (*attribute*) ou des **relations** (*relationship*) :

- Les *attributs* sont de type scalaire :
  - booléen ;
  - numérique (entier, décimal ou flottant) ;
  - date ;
  - chaîne de caractères.
- Les *relations* sont des références vers d'autres entités.

Lorsqu'une relation est définie dans une entité `A`, il faut préciser vers quelle entité `B` doit être établie la relation, ainsi que sa **cardinalité**, c'est-à-dire les nombres minimum et maximum d'objets de type `B` avec lesquels chaque objet de type `A` peut être en relation.

Dans l'application, les instances d'une entité sont généralement de la classe `NSManagedObject`. Le développeur peut dériver cette classe s'il souhaite donner un comportement spécifique à certaines entités.

Mise en pratique sous XCode

Nous allons créer une nouvelle application sous XCode, *Emprunts2*, dans laquelle la classe `LendObject` sera remplacée par une entité *Core Data* du même nom ; ainsi nous bénéficierons de la persistance des données sans rédiger une ligne de code. Il nous faudra néanmoins adapter le code que nous avions écrit dans l'application *Emprunts1* à l'utilisation du framework Core Data.

Créez un nouveau projet sous XCode, de type **Navigation-based Application**. Cochez la case *Use Core Data for storage* cette fois, et intitulez le projet *Emprunts2*.

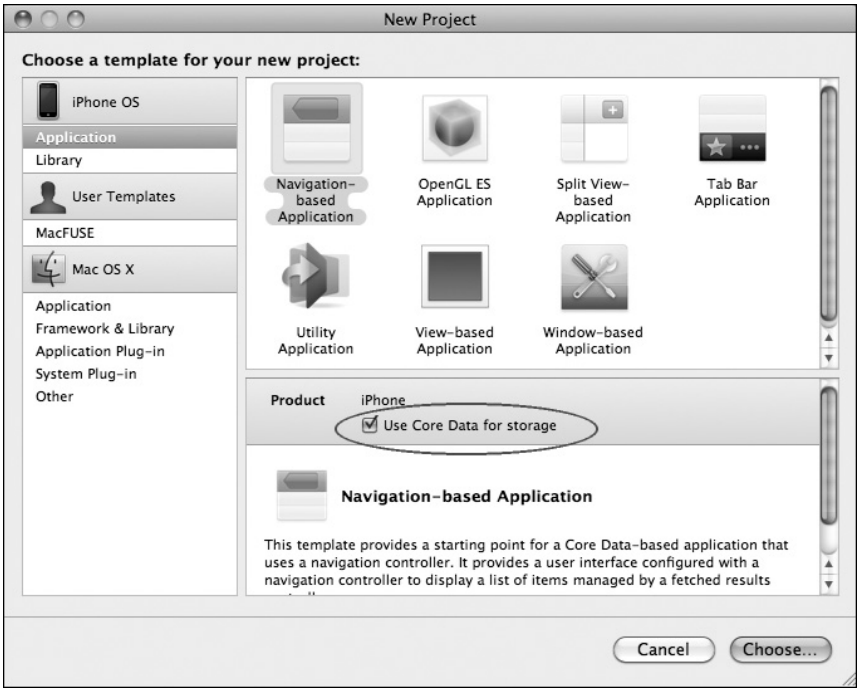


Figure 7.1 : Utilisation de Core Data dans un nouveau projet

Vous constaterez l'existence d'un fichier *Emprunts2.xcdatamodel* dans le groupe *Resources* du projet. Il contient le modèle de données au format Core Data.

Nous allons mettre Core Data en œuvre avec un modèle simple comprenant une relation.

Création des entités

1. Ouvrez ce fichier. Il contient un exemple d'entité, *Event*, avec un attribut *timeStamp*. Vous pouvez soit détruire cette entité, soit la modifier pour créer une entité *LendObject*. Pour détruire l'entité, sélectionnez-la et pressez la touche ⌦.
2. Créez une entité `LendObject` et ses attributs selon le tableau :

Tableau 7.1 : Attributs de l'entité LendObject

Nom	Type	Optional	Transient	Indexed
`objectName`	`String`	Non	Non	Non
`lendDate`	`Date`	Non	Non	Non
`borrowerName`	`String`	Non	Non	Non

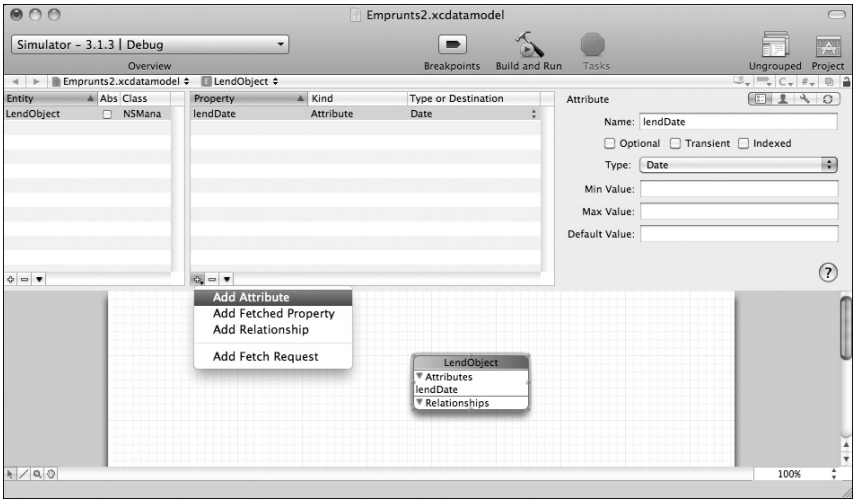


Figure 7.2 : ajout d'attributs dans le modèle de données

3 Pour chaque attribut, décochez toutes les cases *Optional*, *Transient* et *Indexed* :

- *Optional* signifie que l'attribut peut être absent. Au moment de l'enregistrement des données, Core Data vérifie que tous les attributs non optionnels sont présents. Si ce n'est pas le cas, l'enregistrement est refusé.
- *Transient* signifie qu'il n'y a pas de donnée sauvegardée pour cet attribut. Une entité qui possède un attribut éphémère (*Transient*) devrait être d'une classe dérivée de `NSManagedObject` afin d'y définir le comportement relatif à cet attribut.
- *Indexed* est employé sur les attributs que l'on veut utiliser comme critère de recherche.

4 Créez une entité *Category* avec un attribut selon le tableau :

Tableau 7.2 : Attributs de l'entité Category

Nom	Type	Optional	Transient	Indexed
`categoryName`	`String`	Non	Non	Oui

5 Cochez la case *Indexed* pour l'attribut `categoryName`. Nous aurons besoin de retrouver tous les objets appartenant à une catégorie ; c'est donc un critère de recherche.

Création des relations

1 Sélectionnez l'entité *LendObject* et ajoutez une relation (*relationShip*). Paramétrez cette relation de la façon suivante :

- Name ; `category` ;
- Optional ; non ;
- Transient ; non ;
- Destination ; *Category* ;
- Inverse ; *No Inverse Relationship* ;
- To-Many RelationShip ; Non ;
- Delete Rule : *Nullify*.

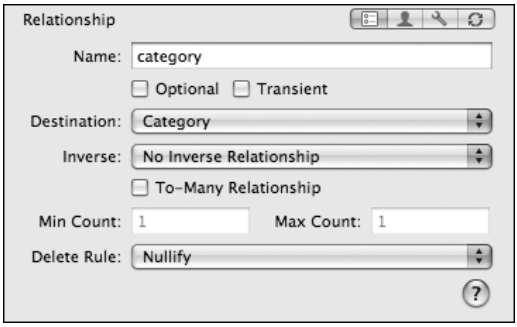


Figure 7.3 : Paramètres de la relation category de l'entité LendObject

Nous venons d'indiquer dans le modèle de données que chaque instance de l'entité *LendObject* doit être liée à une instance (la case *Optional* est décochée donc la relation est obligatoire), et une seule, de l'entité *Category* (la case *To-Many Relationship* est décochée, un objet prêté ne peut avoir qu'une catégorie).

C'est une bonne pratique de définir une relation inverse pour chacune des relations du modèle de données. Cela facilite les vérifications d'intégrité réalisées par Core Data. Nous allons donc créer la relation inverse de *category*.

2 Sélectionnez l'entité *Category* et ajoutez une relation paramétrée de la façon suivante :

- Name ; `lendObjects` ;
- Optional ; non ;
- Transient ; non ;
- Destination ; *LendObject* ;
- Inverse ; *category* ;

— To-Many RelationShip ; Oui ;

— Delete Rule : *Deny*.

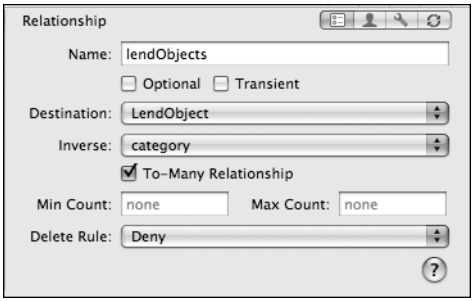


Figure 7.4 : Paramètres de la relation lendObjects de l'entité Category

Cette fois, la case *To-Many RelationShip* est cochée car une catégorie peut contenir plusieurs objets. Nous définissons également la relation inverse. La relation inverse de *category* dans *LendObject* est automatiquement définie à *lendObjects* ; les deux relations sont l'inverse l'une de l'autre.

Le paramètre *Delete Rule* définit le comportement de *Core Data* lorsqu'un objet est détruit. C'est une caractéristique fonctionnelle importante qui permet de garantir l'intégrité des données :

- *No action* ; l'objet est détruit sans autre action.
- *Nullify* ; la destruction de l'objet est prise en compte dans la relation inverse (c'est la valeur par défaut).
- *Cascade* ; les objets liés à l'objet détruit sont également détruits.
- *Deny* ; l'objet ne peut être détruit tant qu'il est en relation avec d'autres objets.

Nous avons fixé le paramètre *Delete Rule* de la relation *lendObjets* à *Deny* car nous souhaitons que l'application nous empêche de détruire une catégorie tant qu'elle contient au moins un objet. Nous allons illustrer ce comportement dans notre application *Emprunts2*.

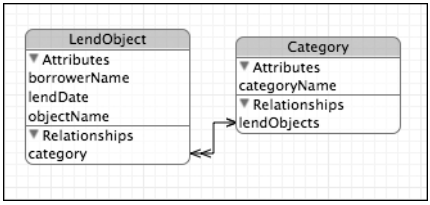


Figure 7.5 : Modèle de données de Emprunts2

3 Enregistrez le fichier *Emprunts2.xcdatamodel* une fois le modèle de données complété.

Programmation du modèle de données

Au lancement de l'application, le fichier *Emprunts2.xcdatamodel* sera lu et son contenu utilisé pour créer une instance de la classe `NSManagedObjectModel`. Cette dernière contiendra l'ensemble des descriptions d'entité du modèle, chacune étant une instance de la classe `NSEntityDescription`.

En pratique, vous n'aurez pas à utiliser directement l'instance du modèle dans vos programmes, XCode a fait le nécessaire lors de la création du projet. Examinez les fichiers de la classe `Emprunts2AppDelegate`, une propriété privée `managedObjectModel` y est définie en lecture seule ; deux autres propriétés relatives à *Core Data* y sont également définies, nous les examinerons plus loin. L'accesseur de cette propriété est défini explicitement dans le fichier *Emprunts2AppDelegate.m* :

```
- (NSManagedObjectModel *)managedObjectModel {
    if (managedObjectModel != nil) {
        return managedObjectModel;
    }
    managedObjectModel = [[NSManagedObjectModel
                           mergedModelFromBundles:nil] retain];
    return managedObjectModel;
}
```

La méthode `+mergedModelFromBundles:` crée un modèle de données en regroupant tous les fichiers au format *.xcdatamodel* contenus dans les paquetages passés en paramètre, ou dans le paquetage de l'application si le paramètre passé est `nil`.

C'est un motif courant pour écrire un accesseur sur une propriété en lecture seule :

- Si la propriété est différente de `nil`, c'est qu'elle a déjà été initialisée. Il suffit de la retourner.
- Si la propriété vaut `nil`, il faut l'initialiser et la retenir, avant de la retourner à l'appelant.

Comprendre le fonctionnement de Core Data

La pile Core Data

Le fonctionnement de Core Data nécessite la collaboration de plusieurs objets qui constituent la **pile Core Data** :

- Le *modèle de données* de la classe `NSManagedObjectModel` contient la description des entités manipulées.

- Les *unités de stockage* de la classe `NSPersistentStore` gèrent les accès aux différents fichiers dans lesquels les données sont conservées.
- Le *coordonnateur des unités de stockage* de la classe `NSPersistentStoreCoordinator` a la responsabilité d'unifier les différentes unités de stockage.
- Les *contextes Core Data*, instances de la classe `NSManagedObjectContext`, sont la principale interface du développeur avec les données Core Data.

La plupart du temps, on utilise une *pile Core Data* offrant une seule *unité de stockage* (donc un seul fichier de données) et un seul *contexte Core Data*.

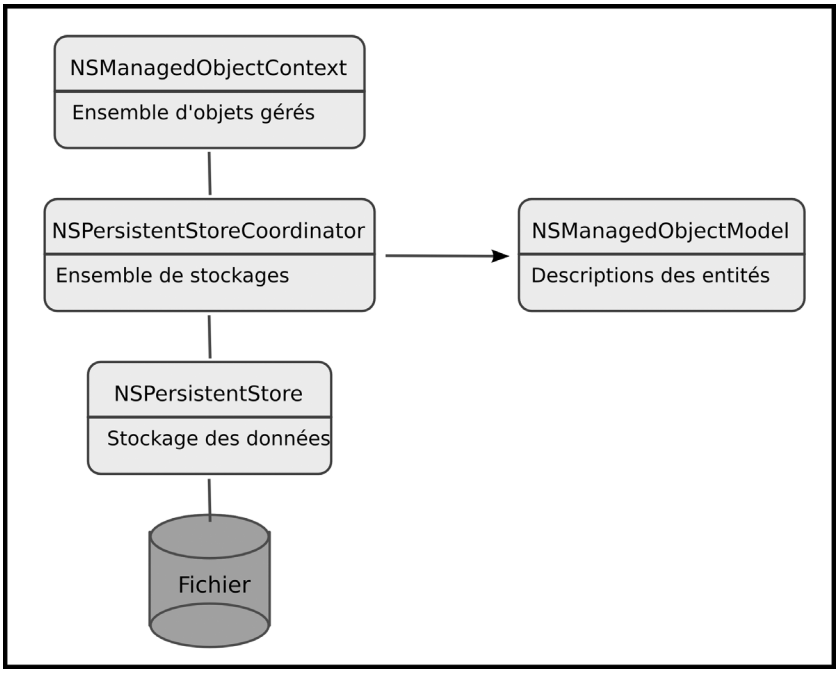


Figure 7.6 : Pile Core Data dans le cas d'une unité de stockage et d'un contexte uniques

Le contexte Core Data est l'objet manipulé par le développeur pour créer, détruire ou rechercher des instances d'entités Core Data. Tous nos contrôleurs de vue auront donc une propriété `managedObjectContext`.

Initialisation d'une pile Core Data

Avant de pouvoir manipuler le *contexte Core Data*, il faut initialiser les différents éléments de la pile. Encore une fois, XCode a réalisé tout le travail lors de la création du projet ; la *pile Core Data* est construite par le délégué de l'application et nous n'aurons pas à ajouter une ligne de code.

1 Ouvrez le fichier *Emprunts2AppDelegate.m* pour voir ce qui se passe au lancement de l'application.

Options de lancement

Les méthodes `–applicationDidFinishLaunching:` et `–application:didFinishLaunchingWithOptions:` sont équivalentes. La seconde permet de récupérer les paramètres de lancement transmis à l'application, par le système des notifications à distance par exemple.

2 Commencez par l'examen de la méthode `–applicationDidFinishLaunching:`. Après avoir créé le contrôleur de vue principal `rootViewController`, la propriété `managedObjectContext` de ce dernier est initialisée avec celle du délégué de l'application :

```
rootViewController.managedObjectContext =
                                    self.managedObjectContext;
```

Cette instruction fait beaucoup plus que donner une valeur à une propriété. Le délégué d'application offre trois propriétés définies en lecture seule :

```
@property (nonatomic, retain, readonly)
                  NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly)
              NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly)
  NSPersistentStoreCoordinator *persistentStoreCoordinator;
```

Nous avons déjà examiné l'accesseur de la propriété `managedObjectModel`. Les deux autres accesseurs sont réalisés selon le même motif : si la propriété vaut `nil`, elle est initialisée par l'accesseur. Regardez le code de l'accesseur `–managedObjectContext`, son premier appel déclenche en cascade l'appel des autres accesseurs et donc l'initialisation de la pile Core Data. Considérons les instructions d'initialisation dans l'ordre selon lequel elles sont exécutées, en supprimant les contrôles d'erreur et les appels successifs, et en ajoutant des étapes intermédiaires, pour nous concentrer sur l'essentiel, la création de chacun des éléments de la pile Core Data :

- Identification du *fichier de données* à utiliser :
  - utilisation d'une fonction du framework *Foundation* pour obtenir le chemin du dossier des documents par défaut :

```
NSString *directory = [NSSearchPathForDirectoriesInDomains
  (NSDocumentDirectory,NSUserDomainMask, YES) lastObject];
```

  - construction du chemin d'accès vers le fichier en ajoutant son nom *Emprunts2.sqlite* au chemin du dossier des documents par défaut :

```
NSString *storeFile = [directory
     stringByAppendingPathComponent: @"Emprunts2.sqlite"];
```

  - construction d'un *URL* permettant d'accéder au fichier de données.

```
NSURL *storeUrl = [NSURL fileURLWithPath: storeFile];
```

- Création et initialisation du *modèle de données* :

```
managedObjectModel = [[NSManagedObjectModel
                      mergedModelFromBundles:nil] retain];
```

- Création et initialisation du *coordonnateur des unités de stockage* et initialisation avec le *modèle de données* :
  - création puis initialisation avec le *modèle de données* :

```
persistentStoreCoordinator =
   [[NSPersistentStoreCoordinator alloc]
   initWithManagedObjectModel:[self managedObjectModel]];
```

  - ajout d'une *unité de stockage* de type SQLite pointant sur le fichier *Emprunts2.sqlite* :

```
NSError *error = nil;
 [persistentStoreCoordinator
           addPersistentStoreWithType:NSSQLiteStoreType
                        configuration:nil
                                  URL:storeUrl
                              options:nil
                                error:&error];
```

- Création et initialisation du *contexte Core Data* :
  - création :

```
managedObjectContext =
                  [[NSManagedObjectContext alloc] init];
```

  - initialisation avec le *coordonnateur des unités de stockage* :

```
 [managedObjectContext
setPersistentStoreCoordinator: persistentStoreCoordinator];
```

Nous n'aurons pas à modifier ces lignes de code mais c'est toujours intéressant de comprendre comment cela fonctionne. Il y a vraisemblablement des aspects de ce code que vous ne maîtrisez pas. Ce n'est pas grave, vous n'en aurez pas besoin pour utiliser Core Data. N'hésitez pas à consulter la documentation afin d'approfondir ces sujets.

Au final, nous disposons d'une propriété `managedObjectContext` qui est un *contexte Core Data* utilisant le modèle de données *Emprunts2.xcdatamodel* de notre application et le fichier de données *Emprunts2.sqlite* dans le dossier des documents de l'application sur l'iPhone.

Utilisation dans une Vue en Table

Le contrôleur de résultats de recherche

Le framework Core Data propose une classe `NSFetchedResultsController`, un *contrôleur de résultats de recherche*, qui facilite l'écriture du délégué et de la source de données d'une vue en table. Vérifiez que le contrôleur de vue principal du projet *Emprunts2* (*RootViewController*) offre une propriété `fetchedResultsController`, instance de cette classe.

Chaque instance de `NSFetchedResultsController` est associée à un contexte Core Data (propriété `managedObjectContext`) et à une requête Core Data (propriété `fetchRequest`). Une requête est une instance de la classe `NSFetchRequest`, elle permet de trouver les instances d'une entité qui répondent à un critère donné ; ces instances sont celles que l'on souhaite afficher dans la vue en table.

Nos vues en table n'offrent qu'une section mais nous pourrions souhaiter que les éléments de la vue en table soient regroupés en fonction d'un critère de tri. Si nous activons cette fonction (nous verrons comment un peu plus loin), la propriété `sections` du contrôleur de résultats de recherche est un tableau dont chaque élément décrit une section. Ce sont des objets qui répondent au protocole `<NSFetchedResultsSectionInfo>`, offrant les propriétés suivantes :

- `numberOfObjects` ; nombre de lignes dans la section ;
- `objects` ; tableau contenant les objets de la section ;
- `name` ; nom de la section, généralement affiché comme titre de la section ;
- `indexTitle` ; titre de l'index, généralement utilisé lorsqu'un index est affiché sur la droite de la vue en table.

Les principales méthodes de la classe `NSFetchedResultsController` sont :

- `-performFetch:` qui exécute la requête associée au contrôleur et retourne `YES` si l'exécution s'est bien déroulée, `NO` autrement. Cette méthode prend en paramètre un pointeur sur une référence d'instance de la classe `NSError` qui nous fournit des informations dans le cas où la requête ne s'est pas bien déroulée.
- `-objectAtIndexPath:` qui retourne l'instance d'entité (sous forme d'instance de `NSManagedObject`) qui doit être affichée sur la ligne identifiée par l'instance de `NSIndexPath` passée en paramètre.

Cette classe contient également un délégué répondant au protocole `<NSFetchedResultsControllerDelegate>` qui est informé de toute modification de la liste des objets. Le contrôleur de vue en table qui contient le contrôleur de résultats de recherche est généralement défini comme son délégué pour informer la vue en table que les données ont été modifiées.

Vérifiez que la classe `RootViewController` du projet *Emprunts2* répond à ce protocole et en implémente notamment les méthodes `-controllerWilChangeContent:` et `-controllerDidChangeContent:`.

Gestion des erreurs

Voici la pratique recommandée par Apple concernant la détection et le traitement des erreurs :

- Les fonctions ou méthodes susceptibles de ne pas s'exécuter correctement retournent `NO` ou `nil` en cas d'erreur.
- La valeur de retour est systématiquement testée dans le code appelant.
- Une référence à un objet `NSError` est passée par référence et initialisée en cas d'erreur.

passage par référence

En *langage C*, donc aussi en *Objective-C*, les fonctions et méthodes ne peuvent pas modifier les paramètres qui leur sont transmis. On dit que les paramètres sont *passés par valeur*. Le seul moyen de modifier un paramètre est donc de passer son adresse, c'est ce que l'on appelle le passage de paramètre *par référence* ou par adresse.

Dans cet exemple, le paramètre de la méthode `-performFetch:` est de type `(NSError **)`. On lui passe l'adresse d'une `error` de type `NSError *` en utilisant l'opérateur de référence du langage C "`&error`".

Les potentialités d'erreur sont nombreuses quand on utilise Core Data : incompatibilité entre un modèle de données et une unité de stockage, entité inexistante dans un modèle de données, erreur de cardinalité dans une relation, etc. Plusieurs méthodes du framework adoptent donc la méthode de détection d'erreur préconisée. Par exemple, la méthode `-performFetch:` de la classe `NSFetchedResultsController` s'utilise de la façon suivante :

```
NSError *error = nil;
if (![fetchedResultsController performFetch:&error]) {
   // Traitement de l'erreur
}
```

Une instance de la classe `NSError` est un conteneur permettant de transmettre des informations plus riches qu'un simple code d'erreur, facilitant ainsi le diagnostic et le traitement de l'erreur. Les principales méthodes de la classe `NSError` sont résumées dans le tableau.

Tableau 7.3 : Principales méthodes de la classe NSError

Thème	Signature	Objet
Création	`+ (id) errorWithDomain: (NSString *)domain code: (NSInteger)code userInfo: (NSDictionary *)dict`	Crée une instance de la classe avec les paramètres `domain`, `code` et `dict`.
Propriétés	`- (NSString *)domain`	Chaîne de caractères identifiant le domaine de l'erreur
	`- (NSInteger)code`	Code d'erreur
	`- (NSDictionary *) userInfo`	Dictionnaire contenant des informations complémentaires sur le contexte de l'erreur

Les erreurs produites par le framework Core Data sont généralement du domaine `NSCocoaErrorDomain`. Une erreur est identifiée par son *domaine* et son *code*. Nous verrons un exemple de traitement d'erreur dans l'application *Emprunts2*.

Enregistrement des données Core Data

La création de la pile *Core Data*, au lancement de l'application, permet de mettre en place la lecture du fichier de données. La structure de données contenue dans le fichier est reproduite au besoin dans le contexte *Core Data*. Toutes les modifications effectuées par l'application (modification des propriétés, création ou suppression d'objets) sont enregistrées dans le contexte. Elles sont enregistrées dans le fichier de données lorsque le contexte reçoit un message `-save:`.

La méthode `–save:` applique la méthode standard pour la détection d'erreur. Elle retourne un booléen et prend en paramètre une variable `NSError` passée par référence.

Cette méthode doit être appelée notamment chaque fois qu'un objet est créé ou détruit car ces opérations sont des sources potentielles d'erreur ; il faut que l'utilisateur soit informé dès que possible s'il réalise une action interdite.

Il faut également enregistrer les modifications effectuées dans le contexte lorsque l'application est sur le point de quitter. XCode génère le code nécessaire dans le délégué d'application, la méthode `–applicationWillTerminate` est appelée juste avant que l'application ne quitte :

```
- (void)applicationWillTerminate:
                              (UIApplication *)application {
   NSError *error = nil;
   if (managedObjectContext != nil) {
      if ([managedObjectContext hasChanges] &&
 ![managedObjectContext save:&error])                        {
         NSLog(@"Unresolved error %@, %@",
                                    error, [error userInfo]);
         abort();
      }
   }
}
```

Dans le code proposé par défaut, un message est affiché dans la console avec la fonction `NSLog`. Ce fonctionnement est suffisant pendant le développement, mais pour une version distribuée, l'affichage d'une alerte serait plus approprié.

Accès aux propriétés des objets Core Data

Les instances d'entités Core Data sont manipulées dans le code comme des instances de la classe `NSManagedObject`, quel que soit le type de l'entité. Les attributs et relations sont accessibles en utilisant les méthodes du motif *KVC* sur cette instance :

- `– (id)valueForKey:(NSString *)key` pour obtenir une propriété ;
- `– (void)setValue:(id)value forKey:(NSString *)key` pour modifier une propriété.

Dans les deux cas, la chaîne de caractères `key` est le nom de la propriété tel qu'il a été défini dans le modèle de données pour l'entité. La classe `NSManagedObject` vérifie que la clé employée est un attribut ou une relation de l'entité considérée.

Formuler des requêtes

L'application *Emprunts2*, comme l'application *Emprunts1* du chapitre précédent, présente à l'utilisateur une liste de catégories, puis dans la vue suivante la liste des objets prêtés qui appartiennent à cette catégorie. Chacune de ces listes est le résultat d'une **requête** Core Data affichée dans une vue en table avec son propre contrôleur de vue :

- La première liste utilise une requête qui porte sur l'entité `Category` et dont le résultat est l'ensemble de toutes les instances de l'entité.
- La seconde liste porte sur l'entité `LendObject`. Nous souhaitons conserver dans cette liste uniquement les objets liés par la relation `category` à la catégorie dont l'attribut `categoryName` est le nom sélectionné par l'utilisateur dans la liste précédente.

Réalisation d'une requête

Examinez la classe `RootViewController` du projet *Emprunts2*, en particulier l'accesseur de la propriété `fetchResultsController`. Cette méthode initialise le contrôleur de résultat de requête, en particulier la requête qui y est associée.

Une requête est représentée par une instance de la classe `NSFetched Request` :

- Sa propriété `entity` représente l'entité concernée par la requête.
- La propriété `fetchBatchSize` prend la valeur 20, pour limiter le nombre d'objets lus en une fois dans le fichier de données. Cela permet d'économiser la mémoire. Seules quelques lignes sont affichées à un instant donné dans une vue en table. Il est donc inutile de lire plus de 20 valeurs à la fois.
- La propriété `sortDescriptors` est un tableau de descripteurs de tri, instances de `NSSortDescriptor`. Un **descripteur de tri** peut être résumé comme l'association du nom d'une propriété de l'entité et d'un ordre de tri, ascendant ou descendant.

Création de la requête pour les catégories

Modifiez la méthode `fetchResultsController` dans le fichier *RootView-Controller.m* pour y définir l'entité et le critère de tri souhaités.

```
NSFetchRequest *fetchRequest =
                                [[NSFetchRequest alloc] init];
NSEntityDescription *entity =
  [NSEntityDescription entityForName:@"Category"
              inManagedObjectContext:managedObjectContext];
[fetchRequest setEntity:entity];
```

```
[fetchRequest setFetchBatchSize:20];
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
                 initWithKey:@"categoryName" ascending:YES];
NSArray *sortDescriptors = [[NSArray alloc]
                      initWithObjects:sortDescriptor, nil];
[fetchRequest setSortDescriptors:sortDescriptors];
```

Création de la requête pour les objets

Nous avons besoin d'un contrôleur de vue en table pour la liste des objets. Ce contrôleur sera analogue à celui des catégories ; aussi, nous allons en faire un clone.

Création du contrôleur de vue

1. Dupliquez les fichiers *RootViewController.h* et *RootViewController.m* sous Finder, dans le dossier du projet. Renommez-les respectivement *ObjectListViewController.h* et *ObjectListViewController.m*. Ajoutez ces 2 nouveaux fichiers au projet *Emprunts2* sous XCode.
2. Dans ces deux fichiers, modifiez toutes les occurrences de texte `RootViewController` en `ObjectListViewController`. Vous pouvez pour cela utiliser la fonction de recherche dans un fichier de XCode (⌘+F).
3. Modifiez la méthode `fetchResultsController` pour y adapter l'entité et le critère de tri souhaités, respectivement `LendObject` et `lendDate`.

Définition d'un critère de recherche

Le critère de recherche est donné par la propriété `predicate` de la requête `NSFetchRequest`. C'est une instance de la classe `NSPredicate` qui doit être initialisée avec une chaîne de caractères.

Ajoutez une propriété `category` de classe `NSManagedObject` à la classe `ObjectListViewController` que nous venons de créer. Ajoutez les lignes suivantes dans la méthode `fetchResultsController` du fichier *ObjectListViewController.m* pour sélectionner seulement les objets de la catégorie choisie :

```
NSPredicate *predicate = [NSPredicate
      predicateWithFormat:@"category.categoryName like %@",
              [self.category valueForKey:@"categoryName"]];
[fetchRequest setPredicate:predicate];
```

Prédicat

Un prédicat est une proposition dont la valeur Vrai ou Faux dépend d'une ou plusieurs variables. Le prédicat est évalué pour chaque instance d'entité. Seuls sont retenus les objets pour lesquels le prédicat est vrai.

Le prédicat le plus simple est une expression de comparaison d'un attribut à une valeur. Un attribut est désigné par son nom. Ici, nous avons utilisé la notation pointée `category.categoryName` ; cela signifie que nous nous intéressons à l'attribut `categoryName` de l'objet lié par la relation `category` à l'entité sur laquelle nous effectuons la recherche (`LendObject`).

On peut employer les opérateurs standard pour effectuer les comparaisons telles que `=`, `<` et `>`. Les chaînes de caractères sont comparées avec l'opérateur `like` qui admet les caractères joker `*` et `?`. Il est également possible de combiner plusieurs expressions avec les opérateurs `AND`, `OR` et `NOT`.

L'écriture des prédicats fait l'objet d'un guide complet dans la documentation Apple (*Predicate Programming Guide*).

Connecter les deux contrôleurs de vue

Le contrôleur de la vue des objets vient d'être ébauché par clonage du contrôleur de vue des catégories. Il faut maintenant établir les connexions entre ces deux contrôleurs.

Classe ObjectListViewController

1 Modifiez la méthode `viewDidLoad` pour définir le titre de la vue. Il faut également supprimer le bouton **Edit** à gauche de la barre de navigation. Il sera remplacé par le bouton de retour.

```
self.title = [self.category valueForKey:@"categoryName"];
//self.navigationItem.leftBarButtonItem=self.editButton
                                                    ✂ Item;
```

2 Modifiez les méthodes `-tableView:cellForRowAtIndexPath:` et `-configureCell:atIndexPath:` pour créer les cellules de la vue en table.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"ObjectCell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
       cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleSubtitle
                reuseIdentifier:CellIdentifier] autorelease];
    [self configureCell:cell atIndexPath:indexPath];
    return cell;
    }
- (void)configureCell:(UITableViewCell *)cell
                  atIndexPath:(NSIndexPath *)indexPath {
    cell.accessoryType =
```

```
            UITableViewCellAccessoryDisclosureIndicator;
    NSManagedObject *managedObject =
     [fetchedResultsController objectAtIndexPath:indexPath];
    cell.textLabel.text =
                [managedObject valueForKey:@"objectName"];
    NSDateFormatter * formatter =
                            [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"dd MMMM yyyy"];
    NSString * subTitle = [NSString stringWithFormat:
                @"prêté à %@ le %@",
                [managedObject valueForKey:@"borrowerName"],
                [formatter stringFromDate:
                  [managedObject valueForKey:@"lendDate"]]];
    [formatter release];
    cell.detailTextLabel.text = subTitle;
}
```

Cette suite d'instructions ressemble à celle que nous avons écrite pour l'application *Emprunts1*. Notez toutefois ces différences :

- Nous avons un identifiant de cellule `ObjectCell`, pour le distinguer de celui que nous utiliserons pour la vue en table des catégories.
- Nous utilisons le contrôleur de résultats de requête afin d'obtenir l'objet à afficher dans la cellule.
- Nous employons le motif *KVC* pour accéder aux attributs de l'objet *Core Data*.

Fichier RootViewController.m

C'est dans le fichier *RootViewController.m* que nous écrirons le code afin d'utiliser le contrôleur de la vue en table pour les objets.

Procédez ainsi :

1. Ajoutez une clause d'importation de la déclaration de ce contrôleur :

```
#import "ObjectListViewController.h"
```

2. Définissez un titre de la vue en table dans la méthode `didLoadView`. Autrement, vous ne distinguerez pas le bouton de retour dans la vue suivante.

```
self.title = @"Catégories";
```

3. Définissez les cellules dans la méthode `–configureCell:atIndexPath:` :

```
- (void)configureCell:(UITableViewCell *)cell
                  atIndexPath:(NSIndexPath *)indexPath {
    NSManagedObject *managedObject =
     [fetchedResultsController objectAtIndexPath:indexPath];
```

```
    cell.textLabel.text =
              [managedObject valueForKey:@"categoryName"];
cell.accessoryType =
             UITableViewCellAccessoryDisclosureIndicator;
```

4 Programmez l'affichage de la liste des cellules lorsqu'une catégorie est sélectionnée. Veillez à initialiser les propriétés `managedObjectContext` et `category` avant d'activer le contrôleur de vue.

```
- (void)tableView:(UITableView *)tableView
      didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
   NSManagedObject *selectedCategory =
    [fetchedResultsController objectAtIndexPath:indexPath];
   ObjectListViewController * objectListViewController =
                      [[ObjectListViewController alloc]
                      initWithStyle:UITableViewStylePlain];
   objectListViewController.managedObjectContext =
           [fetchedResultsController managedObjectContext];
   objectListViewController.category = selectedCategory;
   [self.navigationController
             pushViewController:objectListViewController
                       animated:YES];
   [objectListViewController release];
}
```

5 Construisez l'application pour vérifier qu'il n'y a pas d'erreur dans votre code. Vous pouvez également l'essayer sur le simulateur mais ce premier test sera très frustrant car la liste des catégories est vide. Nous allons maintenant écrire le code pour créer des catégories.

Ajouter un objet

Créer une instance d'entité

La création d'une instance d'entité nécessite le nom de l'entité et un contexte Core Data. Elle se déroule en deux étapes :

- créer une description de *l'entité* ;
- créer une instance de `NSManagedObject` et l'initialiser pour qu'elle devienne une instance de l'entité souhaitée, tout en l'insérant dans le contexte.

Par exemple, si nous souhaitons créer une instance de l'entité *LendObject* :

```
NSEntityDescription *entity = [NSEntityDescription
                     entityForName:@"LendObject"
            inManagedObjectContext: managedObjectContext];
NSManagedObject *newLendObject = [[NSManagedObject alloc]
```

```
                    initWithEntity: entity
insertIntoManagedObjectContext: managedObjectContext];
```

Classes dérivées de NSManagedObject

Même s'il est déclaré de type `NSManagedObject *`, l'objet retourné par la méthode `–initWithEntity:insertIntoManagedObjectContext:` est une instance de la classe définie pour cette entité dans le modèle de données.

Créer une vue détaillée pour les catégories

Lors de sa création, il faut que l'utilisateur puisse saisir le nom de la catégorie. Vous devez donc créer une vue spécifique et son contrôleur :

1. Sous XCode, créez un nouveau fichier pour une classe qui dérive de `UIViewController`. Cochez l'option *With XIB for user interface*. Intitulez cette classe `CategoryViewController`.
2. Ouvrez le fichier *CategoryViewController.xib* pour y tracer l'interface utilisateur. Vous avez besoin uniquement d'un champ de texte pour saisir le nom de la catégorie.

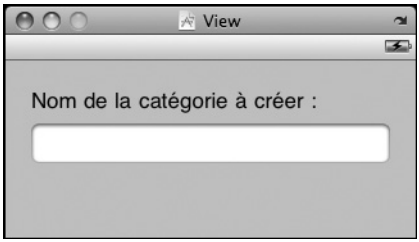


Figure 7.7 : interface utilisateur pour la saisie d'une catégorie

3. Sous XCode, modifiez la classe `CategoryViewController` :
   - Ajoutez une propriété `NSManagedObject * category`.
   - Ajoutez un outlet `UITextField * categoryNameField`.
   - Modifiez la méthode `-viewDidLoad`.
   - Créez une méthode `–viewWillDisappear:` :

```
- (void)viewDidLoad {
    self.categoryNameField.text =
            [self.category valueForKey:@"categoryName"];
    self.title = @"Catégorie";
    [super viewDidLoad];
}
- (void)viewWillDisappear:(BOOL)animated{
    [self.category setValue:self.categoryNameField.text
                                forKey:@"categoryName"];
```

```
    [super viewWillDisappear:animated];
}
```

4 Connectez l'outlet au champ de texte sous Interface Builder.

Modifier le contrôleur de vue principal

Il faut maintenant prévoir l'activation de la vue que nous venons de réaliser. Le modèle d'application que nous avons indiqué lors de la création du projet dispose un bouton d'ajout dans la barre de navigation. L'action connectée à ce bouton est la méthode `–insertNewObject` préparée par XCode.

1 Modifiez cette méthode dans le fichier *RootViewController.m* :

```
- (void)insertNewObject {
    NSManagedObjectContext *context =
          [fetchedResultsController managedObjectContext];
    NSEntityDescription *entity =
         [[fetchedResultsController fetchRequest] entity];
    NSManagedObject *newManagedObject = [NSEntityDescription
             insertNewObjectForEntityForName:[entity name]
                        inManagedObjectContext:context];
    [newManagedObject
     setValue:@"saisissez le nom" forKey:@"categoryName"];
    NSError *error = nil;
    if (![context save:&error]) {
       NSLog(@"Unresolved error %@, %@", error,
                                          [error userInfo]);
       abort();
    }
    CategoryViewController * itemViewController =
                  [[CategoryViewController alloc]
    initWithNibName:@"CategoryViewController" bundle:nil];
    itemViewController.category = newManagedObject;
    [self.navigationController
      pushViewController:itemViewController animated:YES];
    [itemViewController release];
}
```

2 Ajoutez une clause `#import "CategoryViewController.h"` en tête du fichier.

Vous reconnaissez dans le code proposé par XCode les instructions pour créer un objet Core Data. La particularité ici est que le contexte et la description de l'entité sont extraits du contrôleur de résultats de requête. Ainsi le nom de l'entité est inscrit à un seul endroit dans le fichier *RootViewController.m* ; il est plus facile d'assurer la maintenance du code.

Nous initialisons l'objet nouvellement créé puisque la présence de l'attribut est obligatoire, puis nous enregistrons le contexte.

Les autres instructions sont classiques : création du contrôleur de vue pour la saisie du nom de la catégorie, initialisation de ses propriétés et activation.

Challenge

Vous savez maintenant comment ajouter une instance d'entité à un contexte Core Data. Nous vous proposons donc de mettre en œuvre la même méthode pour la fonction d'ajout d'un objet prêté.

Ce challenge est assez facile. Vous pouvez adapter le contrôleur de vue `LendObjectViewController` et son fichier *NIB* que nous avions réalisés pour l'application *Emprunts1*.

Nous rencontrerons des challenges plus complexes dans la suite de ce chapitre.

Supprimer un objet

Si vous avez testé votre application *Emprunts2*, vous avez constaté que la vue principale comprenait un bouton **Edit** à gauche de la barre de navigation. Si vous avez eu la curiosité de toucher ce bouton, vous avez pu voir une vue en table en mode Édition.

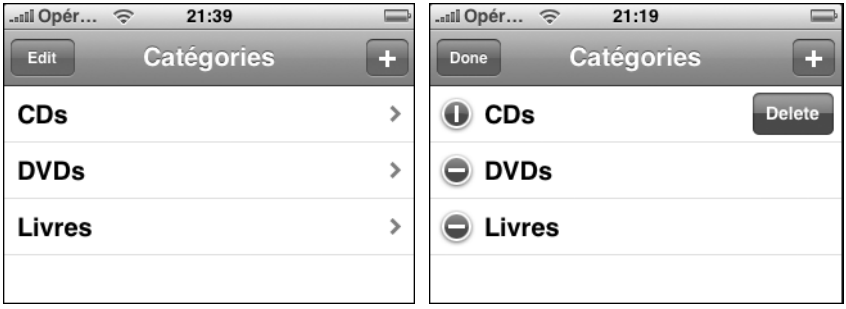


Figure 7.8 : Mode Édition d'une vue en table

Programmation du mode édition

Le bouton **Edit** est activé dans la méthode `–viewDidLoad` du contrôleur de vue principal.

```
self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

Par défaut, en mode Édition, un bouton permettant la suppression est affiché dans chaque cellule. Ce comportement peut être modifié en implémentant la méthode `-tableView:editingStyleForRowAtIndexPath:` dans le délégué de la vue en table (généralement son contrôleur). Cette méthode doit retourner une valeur du type énuméré `UITableViewCellEditingStyle` pour définir le style de bouton à afficher dans la cellule.

Tableau 7.4 : Valeurs du type énuméré UITableViewCellEditingStyle

Bouton	Valeur
	`UITableViewCellEditingStyleNone`
C07-10.png	`UITableViewCellEditingStyleDelete`
C07-11.png	`UITableViewCellEditingStyleInsert`

Lorsque la vue en table est en mode Édition et que l'utilisateur touche le bouton d'édition d'une cellule, la source de données (généralement le contrôleur de la vue en table) reçoit le message `-tableView:commitEditingStyle:forRowAtIndexPath:`. Nous allons modifier cette méthode proposée par XCode.

Vérification à la suppression d'une catégorie

La suppression d'un objet s'effectue par l'envoi du message `-deleteObject:` au contexte Core Data, avec l'objet à supprimer passé en paramètre.

Dans le modèle de données, nous avons donné la valeur *Deny* au paramètre *Delete Rule* de la relation *lendObjects* de l'entité *Category*. Nous souhaitons en effet interdire la suppression d'une catégorie tant qu'elle contient au moins un objet. Cette règle concernant la suppression fait partie d'un ensemble plus global nommé **règles d'intégrité**.

Règles d'intégrité
Ce sont les règles que doivent respecter les données enregistrées dans une base de données afin que cet ensemble de données conserve un sens.

Les règles d'intégrité sont vérifiées par Core Data au moment de l'enregistrement du contexte. Nous allons donc tester les valeurs retournées par la méthode `-save:`, en particulier l'instance de `NSError`, pour savoir si nous sommes dans le cas de la règle de suppression :

- Le domaine de l'erreur est `NSCocoaErrorDomain`.
- Le code d'erreur est `NSValidationRelationshipDeniedDeleteError`.

Le traitement de cette erreur est alors :

- afficher un message d'alerte pour informer l'utilisateur ;
- réinsérer l'objet détruit dans le contexte :

```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
             forRowAtIndexPath:(NSIndexPath *)indexPath {
   if (editingStyle == UITableViewCellEditingStyleDelete) {
      NSManagedObjectContext *context =
           [fetchedResultsController managedObjectContext];
      NSManagedObject *objectToDelete =
    [fetchedResultsController objectAtIndexPath:indexPath];
      [context deleteObject:objectToDelete];
      NSError *error = nil;
      if (![context save:&error]) {
         if (([error.domain
               isEqualToString:NSCocoaErrorDomain]) &&
(error.code == NSValidationRelationshipDeniedDeleteError)){
            UIAlertView * alertView = [[UIAlertView alloc]
               initWithTitle:@"Suppression d'une catégorie"
               message:@"Il n'est pas autorisé de supprimer
   une catégorie pour laquelle il existe des objets prêtés"
                                     delegate:nil
                           cancelButtonTitle:@"OK"
                           otherButtonTitles:nil];
            [alertView show];
            [alertView release];
         }
         [context insertObject:objectToDelete];
      }
   }
}
```

Liste des codes d'erreur
La liste des codes d'erreur de Core Data est dans la documentation **Core Data Constants Reference**.

Construisez et testez l'application. Son comportement devrait être satisfaisant maintenant.

Figure 7.9 : Détection d'erreur lors de la suppression

Challenges

Challenge 1

Finalisez l'application *Emprunts2* pour permettre à l'utilisateur de supprimer un objet prêté.

Challenge 2

Ajoutez une fonctionnalité permettant à l'utilisateur de voir la liste de tous les objets prêtés. Pour que cette fonction soit utilisable, il faut penser à permettre à l'utilisateur de préciser la catégorie d'un objet lors de sa création.

Challenge 3

Un challenge purement technique, les classes `RootViewController` et `ObjectListViewController` se ressemblent beaucoup. Il y a de nombreuses lignes de code dupliquées, ce qui ne favorise pas la maintenance.

Écrivez une classe qui puisse être utilisée pour remplacer ces deux contrôleurs de vue.

7.2. Utiliser les listes de propriétés

Core Data est un très bon framework mais sans doute lourd dans les situations où seules quelques données sont concernées. Les **Listes de propriétés** (*property list*) sont plus faciles à employer.

Format des listes de propriétés

Vous avez déjà utilisé une liste de propriété : le fichier au format *.plist* que l'on trouve dans tous les projets et dans lequel vous avez défini l'icône de l'application.

Key	Value
▼ Information Property List	(12 items)
Localization native development region	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone environmen	☑
Main nib file base name	MainWindow

Figure 7.10 : Exemple de liste de propriétés

Ce fichier est au format *XML*. Il contient un dictionnaire composé d'un ensemble de couples (clé, valeur) :

```
<?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>CFBundleDevelopmentRegion</key>
   <string>English</string>
   <key>CFBundleDisplayName</key>
   <string>${PRODUCT_NAME}</string>
   <key>CFBundleExecutable</key>
   <string>${EXECUTABLE_NAME}</string>
   <key>CFBundleIconFile</key>
   <string></string>
   <key>CFBundleIdentifier</key>
   <string>com.yourcompany.
            ${PRODUCT_NAME:rfc1034identifier}</string>
   <key>CFBundleInfoDictionaryVersion</key>
   <string>6.0</string>
   <key>CFBundleName</key>
   <string>${PRODUCT_NAME}</string>
   <key>CFBundlePackageType</key>
```

```
    <string>APPL</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>1.0</string>
    <key>LSRequiresIPhoneOS</key>
    <true/>
    <key>NSMainNibFile</key>
    <string>MainWindow</string>
</dict>
</plist>
```

Utilisation des listes de propriétés

Accès à une liste de propriétés

Une liste de propriétés est un dictionnaire, la classe `NSDictionary` dispose donc des méthodes pour y accéder :

- `+ (id)dictionaryWithContentsOfFile:(NSString *)path` permet de créer un dictionnaire à partir de la liste de propriétés dont le chemin d'accès est passé en paramètre.
- `– (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag` permet d'enregistrer le récepteur dans un fichier dont le chemin d'accès est passé en paramètre. Le paramètre `atomically` permet de garantir l'intégrité du fichier. En cas d'erreur lors de l'écriture, le fichier n'est pas modifié si ce paramètre vaut `YES`.

Types de données

Une liste de propriétés est donc un dictionnaire soumis à quelques limitations :

- Les clés sont obligatoirement des chaînes de caractères.
- Les valeurs doivent être de l'un des types prédéfinis ci-après.

Le tableau indique, pour chaque type autorisé, l'étiquette utilisée dans le fichier *XML* et la classe d'objet correspondante.

Tableau 7.5 : Types de données autorisés dans une liste de propriétés

Type	Élément XML	Classe Objective-C
Tableau	`<array>`	`NSArray`
Dictionnaire	`<dict>`	`NSDictionary`
Chaîne de caractères	`<string>`	`NSString`
Data	`<data>`	`NSData`
Date	`<date>`	`NSDate`

Tableau 7.5 : Types de données autorisés dans une liste de propriétés

Type	Élément XML	Classe Objective-C
Nombre entier	`<integer>`	`NSNumber` (`intValue`)
Nombre réel	`<real>`	`NSNumber` (`floatValue`)
Booléen	`<true/>` ou `<false/>`	`NSNumber` (`boolValue`)

Mise en pratique

Nous allons reprendre notre application *Convertisseur2* pour la doter de la persistance des données. Le principe sera le suivant :

- Des méthodes pour lire et écrire un fichier sont ajoutées à la classe `Convertisseur`. C'est elle qui détient les données que nous souhaitons persistantes.
- Le fichier de données est lu au démarrage de l'application puis écrit lorsque l'application va quitter.

Modifier la classe Convertisseur

1 Ajoutez la déclaration des méthodes dans le fichier *Convertisseur.h* :

```
- (BOOL)readFromFile:(NSString *)path;
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag;
```

Ces méthodes prennent les mêmes paramètres que les méthodes de `NSDictionary` pour la lecture et l'écriture d'une liste de propriétés. Leur travail consistera essentiellement à constituer un dictionnaire intermédiaire.

2 Ajoutez-les dans le fichier *Convertisseur.m*.

```
- (BOOL)readFromFile:(NSString *)path{
   if ([self init]) {
      NSDictionary * dict = [NSDictionary
                     dictionaryWithContentsOfFile:path];
      if (dict) {
         dollar =
             [[dict objectForKey:@"dollar"] floatValue];
         euro = [[dict objectForKey:@"euro"] floatValue];
         dollarsPourUnEuro =
    [[dict objectForKey:@"dollarsPourUnEuro"] floatValue];
         }
      }
   return self;
}
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag{
```

```
    NSDictionary *dict = [NSDictionary dictionaryWithObjects:
                                [NSArray arrayWithObjects:
                          [NSNumber numberWithFloat:self.dollar],
                            [NSNumber numberWithFloat:self.euro],
     [NSNumber numberWithFloat:self.dollarsPourUnEuro],nil]
      forKeys:[NSArray arrayWithObjects:
               @"dollar",@"euro",@"dollarsPourUnEuro",nil]];
    return [dict writeToFile:path atomically:flag];
}
```

Classe NSNumber

La classe `NSNumber` utilisée précédemment sert à empaqueter un nombre, entier ou flottant, dans un objet. On ne peut pas insérer directement une valeur de type `int` ou `float` dans un conteneur `NSArray` ou `NSDictionary`. On utilise donc la classe `NSNumber`. On emploie également cette classe avec le motif *KVC*.

Tableau 7.6 : Principales méthodes de la classe NSNumber

Thème	Signature de la méthode	Remarque
Créer un nombre	`+ (NSNumber *) number Wih<Type>: (type) value`	`<Type>` doit être remplacé par l'un des types scalaires ci-après.
Obtenir la valeur d'un nombre	`– (type) <type>Value`	

Les types scalaires utilisables sont : `Bool`, `Char`, `Double`, `Float`, `Int`, `Integer`, `Long`, `LongLong`, `Short`, `UnsignedChar`, `UnsignedInt`, `UnsignedInteger`, `UnsignedLong`, `UnsignedLongLong` et `UnsignedShort`.

Le nom du type utilisé prend une majuscule dans les méthodes `+number`, et une minuscule dans les méthodes `Value`. Par exemple :

```
+ (NSNumber *) numberWihLongLong: (long long) value
- (long long) longLongValue
```

Lecture au démarrage de l'application

Jusqu'à présent, nous avons réalisé les initialisations dans le délégué d'application. Dans l'application *Convertisseur2*, ce délégué n'a pas de propriété convertisseur contrairement au contrôleur de la vue principale. Nous aurons donc moins de code à modifier si l'initialisation est réalisée dans ce dernier.

Identification du fichier

Nous allons nous inspirer de la technique utilisée pour identifier et gérer l'unité de stockage Core Data.

1. Déclarez une propriété `NSString * storeFile` dans l'interface de la classe `MainViewController` puis ajoutez le synthétiseur des accesseurs pour cette propriété dans le fichier *MainViewController.m*
2. Ajoutez la définition de l'accesseur :

```
- (NSString *)storeFile {
    if (!storeFile){
        NSString *directory =
    [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
            NSUserDomainMask, YES) lastObject];
        storeFile = [directory stringByAppendingPathComponent:
                                    @"Convertisseur.plist"];
        [storeFile retain];
    }
    return storeFile;
}
```

Nous utiliserons un fichier nommé *Convertisseur.plist* situé dans le dossier des documents.

Initialisation

Les initialisations de l'interface utilisateur doivent être réalisées dans la méthode `–viewDidLoad` du contrôleur de vue, c'est le bon endroit pour lire la liste de propriétés et initialiser les champs de texte :

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self.convertisseur readFromFile:self.storeFile];
    euroField.text =
                stringWithCurrency(self.convertisseur.euro);
    dollarField.text =
              stringWithCurrency(self.convertisseur.dollar);
}
```

La propriété `storeFile` est obtenue en passant par son accesseur ; cela garantit qu'elle contiendra le chemin du fichier à utiliser.

Écriture lorsque l'application quitte

Lorsque l'application se termine, nous savons que le message `–applicationWillTerminate` est transmis au délégué d'application. Ce serait l'endroit idéal pour sauvegarder l'objet convertisseur dans la liste de propriétés *Convertisseur.plist*. Mais la lecture du fichier est réalisée dans le contrôleur de vue. C'est ce dernier qui possède les propriétés `convertisseur` et `storeFile`. Il est donc logique que l'écriture du fichier soit également réalisée dans le contrôleur de vue.

Il faut que le contrôleur de vue soit prévenu lorsque l'application va se terminer, de la même façon que le délégué d'application.

Notifications

Le framework Cocoa Touch propose le mécanisme des **notifications**. Chaque fois qu'un événement important se produit, le *Centre de notification* est informé. Les objets qui souhaitent être informés de ces événements doivent s'*abonner* au centre de notification.

Comment connaître la liste des événements disponibles ? Dites-vous que toutes les classes possédant un délégué sont susceptibles d'émettre des notifications. À titre d'exemple, le tableau ci-après indique quelques notifications émises par les classes que nous connaissons déjà.

Tableau 7.7 : Exemples de notifications émises

Classe	Notification
`UITextField`	`UITextFieldTextDidBeginEditingNotification`
	`UITextFieldTextDidChangeNotification`
	`UITextFieldTextDidEndEditingNotification`
`UITableView`	`UITableViewSelectionDidChangeNotification`
`UIApplication`	`UIApplicationDidBecomeActiveNotification`
	`UIApplicationDidFinishLaunchingNotification`
	`UIApplicationDidReceiveMemoryWarningNotification`
	`UIApplicationSignificantTimeChangeNotification`
	`UIApplicationWillResignActiveNotification`
	`UIApplicationWillTerminateNotification`

Le délégué est informé des événements importants mais on voit également que le mécanisme des *notifications* le permet aussi à tout objet d'être informé de ces événements.

Nous allons abonner le contrôleur de vue principal de *Convertisseur2* à la notification `UIApplicationWillTerminateNotification`.

Programmer l'abonnement

Modifiez la méthode `–initWithNibName:` dans le fichier *MainViewController.m*.

```
- (id)initWithNibName:(NSString *)nibNameOrNil
                        bundle:(NSBundle *)nibBundleOrNil {
   if (self = [super initWithNibName:nibNameOrNil
                                  bundle:nibBundleOrNil]) {
      [[NSNotificationCenter defaultCenter]
         addObserver:self
            selector:@selector(applicationWillTerminate:)
```

```
                name:UIApplicationWillTerminateNotification
              object:nil];
    }
    return self;
}
```

Nous indiquons au centre de notification par défaut que nous souhaitons que le message `applicationWillTerminate:` soit transmis au contrôleur de vue principal (`addObserver:self`), lorsque l'application est sur le point de se terminer, quel que soit l'objet émettant cette notification (`object:nil`).

Le contrôleur de vue principal devient un **observateur**. Il ne faut pas oublier de supprimer l'observateur lorsqu'il est appelé à disparaître.

```
- (void)dealloc {
    [self viewDidUnload];
    [[NSNotificationCenter defaultCenter]
                                        removeObserver:self];
    [super dealloc];
}
```

Réception de la notification

Il suffit maintenant d'ajouter la déclaration de la méthode `applicationWillTerminate:` dans le fichier *MainViewController.h* :

```
- (void)applicationWillTerminate:
                              (NSNotification *)notification;
```

Définissez ensuite cette méthode dans le fichier *MainViewController.m*.

```
- (void)applicationWillTerminate:
                              (NSNotification *)notification{
    [self.convertisseur writeToFile:self.storeFile
                         atomically:YES];
}
```

L'instance de la classe `NSNotification` qui est reçue par cette méthode est un conteneur dont les propriétés décrivent la notification :

- `name` est un `NSString` contenant le nom de la notification.
- `object` est l'objet qui a émis la notification.
- `userInfo` est un dictionnaire contenant des informations complémentaires optionnelles.

7.3. Checklist

Nous avons exploré deux techniques importantes mises en œuvre pour la persistance des données :

- *Core Data* qui permet de réaliser de petites bases de données :
  - les entités, attributs et relations ;
  - les règles d'intégrité ;
  - la pile Core Data, composée des unités de stockage, du coordonnateur, du modèle de données et du contexte.
- les *listes de propriétés* qui permettent la persistance de dictionnaires de données :
  - le format *XML* utilisé pour représenter un dictionnaire ;
  - les types de données autorisés.

Nous avons également avancé dans notre compréhension des vues en table et de la navigation par barre de navigation :

- le mode Édition ;
- la classe `NSFetchedResultsController` qui facilite l'utilisation de Core Data et son délégué.

Nous avons vu les classes `NSError`, `NSNumber` et `NSNotification` et le motif *notification/observation*.

DESSINS ET ANIMATIONS

Nous en avons terminé avec les interfaces utilisateur un peu tristes contenant des champs de texte et des boutons. Nous allons maintenant tirer parti des possibilités graphiques de l'iPhone, en particulier de ses capacités à gérer les animations. Nous apprendrons également à agrémenter nos applications avec des effets sonores.

8.1. Animer les images

Il existe deux techniques pour agrémenter son interface utilisateur avec des éléments graphiques :

- disposer d'images préparées, positionnées voire animées par le programme ;
- coder les instructions pour que le programme dessine.

Le plus simple et le plus efficace est de disposer d'images déjà prêtes, au format *PNG* ou *JPEG*. Nous avons déjà appris à positionner une image statique avec Interface Builder dans le chapitre consacré à la prise en main du SDK. Nous nous attacherons ici aux deux techniques d'animation utilisées avec les images :

- animation du contenu de l'image : les images animées ;
- déplacement d'une image sur l'écran.

Images animées

La technique d'*animation* d'une image est celle utilisée dans les dessins animés : nous affichons une succession d'images à un rythme rapide, par exemple 30 images par seconde, pour produire la sensation d'animation. Nous supposerons donc que nous disposons d'un ensemble d'images. Il nous reste à voir comment utiliser cet ensemble dans une application iPhone.

Application Terre

L'objet de l'application *Terre* est de voir tourner le globe terrestre. La rotation complète du globe est décomposée en 44 images au format PNG disponibles dans les exemples complémentaires à cet ouvrage. Vous pouvez aussi choisir votre propre séquence d'images pour réaliser cette application (voir Figure 8.1).

L'interface utilisateur doit contenir une vue de type `UIImageView` dans laquelle sera effectuée l'animation.

Figure 8.1 : Application Terre

Contrôleur de vue

1. Sous XCode, créez une application de type *View Based Application* et nommez-la *Terre*. Ajoutez au projet les fichiers d'images pour composer l'animation.
2. Modifiez l'interface de la classe `TerreViewController` pour y déclarer l'outlet `terre` de type `UIImageView*`.

```
@interface TerreViewController : UIViewController {
   IBOutlet UIImageView *terre;
}
@property(nonatomic,retain) UIImageView *terre;
@end
```

Nous allons maintenant écrire le code pour charger les 44 images dans la vue `terre` puis déclencher l'animation.

3. Modifiez la méthode `–viewDidLoad` dans le fichier *TerreViewController.m*. N'oubliez pas d'enlever la mise en commentaire de cette méthode :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   NSMutableArray *images = [[NSMutableArray alloc] init];
   for (int i=1;i<=44;i++) {
      UIImage *oneImage=[UIImage imageNamed:[NSString
                         stringWithFormat:@"%d.png",i]];
      [images addObject:oneImage];
   }
```

```
    terre.animationImages = [NSArray arrayWithArray:images];
    [images release];
    [terre startAnimating];
}
```

Les fichiers images sont nommés *1.png*, *2.png*, etc. jusqu'à *44.png*. La méthode contient donc une boucle pour composer le nom de chaque fichier et l'ajouter dans le tableau modifiable `images`. Ce tableau est ensuite converti en tableau immuable, pour améliorer les performances, avant d'être utilisé pour initialiser la propriété `animationImages` de la vue `terre`. Enfin, l'animation de cette vue est déclenchée par l'émission du message `startAnimating`.

Interface utilisateur

1. Ouvrez le fichier *TerreViewController.xib* pour composer l'interface utilisateur. Ajoutez une **Vue Image** (*Image View*) et liez-la avec l'outlet `terre` du contrôleur de vue.
2. Utilisez l'inspecteur d'attributs pour définir le mode de dessin de l'image comme souhaité. Vous pouvez faire glisser l'une des images à partir de la bibliothèque *media* dans la vue image pour réaliser des essais.

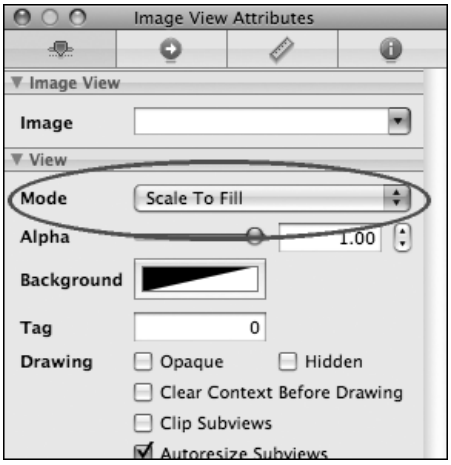


Figure 8.2 : Mode de dessin

Les modes les plus appropriés sont généralement les suivants :

- *Scale To Fill* redimensionne l'image qui sera insérée pour qu'elle remplisse toute la vue.
- *Aspect Fit* redimensionne l'image pour qu'elle prenne la place maximale sans que son aspect soit modifié. Les zones éventuellement non occupées sont transparentes.

- *Aspect Fill* redimensionne l'image pour qu'elle occupe toute la vue sans que son aspect soit modifié. Certaines parties de l'image peuvent être coupées.
- *Center* centre l'image dans la vue sans la redimensionner.

Vous pouvez tester votre application.

Classe UIImage

Dans la méthode `–viewDidLoad`, nous utilisons la méthode `–imageNamed:` de la classe `UIImage` pour créer une image. Le paramètre de cette méthode est le nom d'un fichier, y compris l'extension mais sans chemin d'accès. Le fichier est recherché dans les ressources de l'application.

Cette classe dispose aussi de la méthode `–imageWithContentsOfFile:` qui joue le même rôle mais prend en paramètre le chemin d'accès complet à un fichier. Ces deux méthodes retournent `nil` si le fichier n'a pu être trouvé.

Challenge

Le code que nous avons écrit fonctionne pour charger 44 images. Il serait plus facile d'en assurer la maintenance s'il ne contenait pas cette information, s'il fonctionnait quel que soit le nombre d'images insérées dans les ressources de l'application.

Modifiez la méthode `–viewDidLoad` du contrôleur de vue pour qu'elle ne dépende plus du nombre d'images à lire.

Classe UIImageView

La création d'une animation était très simple : tout le travail est réalisé par la classe `UIImageView`. Elle mérite qu'on la regarde plus attentivement ; les méthodes et propriétés les plus utilisées sont documentées dans le tableau.

Tableau 8.1 : Principales méthodes et propriétés de la classe UIImageView

Thème	Signature	Objet
Initialisation	`– (id) initWithImage: (UIImage *)image`	Initialise le récepteur avec une image.
Image	`@property(nonatomic, retain) UIImage *image`	Image contenue dans le récepteur.

Tableau 8.1 : Principales méthodes et propriétés de la classe UIImageView

Thème	Signature	Objet
Animation	`@property(nonatomic, copy) NSArray *animationImages`	Tableau d'images pour l'animation. Si cette propriété est initialisée, la propriété `image` n'est pas utilisée.
	`@property(nonatomic) NSTimeInterval animationDuration`	Durée d'un cycle en secondes. Par défaut, le cycle est calculé pour une vitesse d'affichage de 30 images pas seconde.
	`@property(nonatomic) NSInteger animationRepeatCount`	Nombre de cycles à dérouler avant l'arrêt de l'animation. Par défaut 0 ; l'animation ne s'arrête pas seule.
	`– (void)startAnimating`	Démarre l'animation.
	`– (void)stopAnimating`	Stoppe l'animation.
	`– (BOOL)isAnimating`	Retourne `YES` si l'animation est en cours.

Sonoriser une application

Pour donner encore plus de vie à une application, on peut lui adjoindre des *effets sonores*. Nous allons ajouter un bruit d'ambiance sur la rotation du globe terrestre.

1 Ajoutez un fichier au format *MP3* ou au format *WAV* au projet *Terre*, par exemple *ambiance.mp3*. Ajoutez les lignes de code suivantes dans la méthode `–viewDidLoad` du contrôleur de vue :

```
NSError *error;
player = [[AVAudioPlayer alloc] initWithContentsOfURL:
          [NSURL fileURLWithPath:[[NSBundle mainBundle]
                  pathForResource:@"ambiance"
                           ofType:@"mp3"
                      inDirectory:@"/"]] error:&error];
player.numberOfLoops = -1;
[player prepareToPlay];
[player play];
```

2 Ajoutez une propriété `AVAudioPlayer *player` à la classe `TerreViewController`.

Nous avons créé un lecteur audio, une instance de la classe `AVAudioPlayer`. Nous l'avons initialisé avec le fichier *ambiance.mp3*. Nous avons ensuite demandé à ce lecteur de préparer la restitution puis de lire le fichier sonore.

Les méthodes principales de cette classe sont :

- `–initWithContentOfURL:error:` pour initialiser le lecteur avec le contenu d'une URL ;

- `-prepareToPlay` pour préparer la restitution ;
- `-play` pour commencer la lecture ;
- `-pause` pour suspendre la lecture ;
- `-stop` pour arrêter la lecture.

3 Testez l'application afin de vérifier que la terre tourne maintenant dans une belle ambiance sonore.

Format des sons
L'iPhone sait lire plusieurs formats sonores mais les meilleures performances sont atteintes avec le format natif du processeur : PCM 16 bits signé, little endian, 44 100 Hz, encapsulé dans un fichier *WAV* ou *AIF*.

Déplacer une image

Intéressons-nous maintenant à la technique de base permettant de déplacer sur l'écran un objet représenté par une image. Cette dernière pourra ensuite être enrichie pour animer plusieurs objets. Nous aborderons plus tard les techniques permettant de créer des images par programmation : *Quartz2D* puis *OpenGL-ES*.

Nous illustrerons cette technique avec le déplacement d'une boule sur une table de billard.

Débuter l'application

1 Créez une application *Billard* de type *View Based Application*. Ajoutez un outlet `ball` de type `UIImageView*` à la classe `BillardViewController` :

```
@interface BillardViewController : UIViewController {
   IBOutlet UIImageView *ball;
}
@property(nonatomic,retain) UIImageView *ball;
@end
```

2 Ajoutez aux ressources du projet un fichier au format *PNG* ou *JPEG* contenant une image représentant la boule de billard. L'effet graphique sera optimal si le pourtour de la bille est transparent, l'image doit contenir une couche Alpha (voir Figure 8.3).

Couche Alpha

Sur iPhone PS, les couleurs sont définies par quatre composantes dont la valeur est comprise entre `0.` et `1.0`. Les trois premières définissent l'intensité des couleurs primaires, Rouge, Vert et Bleu, la quatrième est la valeur Alpha de la couleur qui en définit le niveau de transparence ; `1.0` pour une couleur opaque et `0.` pour une couleur totalement transparente.

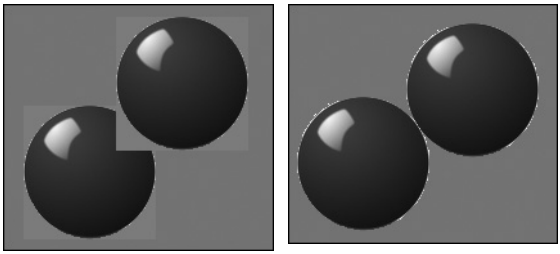


Figure 8.3 : Effet de la transparence sur le pourtour de la bille

3 Ouvrez le fichier *BillardViewController.xib* pour préparer l'interface utilisateur :

- Modifiez la couleur de fond de la vue principale pour obtenir un vert proche du feutre d'une table de billard.
- Faites glisser l'image de la boule de billard depuis la bibliothèque media d'Interface Builder sur la vue principale.
- Si vous souhaitez modifier la taille de l'image, choisissez le mode approprié, par exemple *Aspect Fit*.
- Connectez l'outlet `ball` du propriétaire du fichier (*File's owner*) à l'image de la bille, en fait à la vue image (*Image View*) contenant l'image.

Se repérer dans une vue

Jusqu'à présent, nous avons utilisé Interface Builder pour disposer les différents éléments de l'interface utilisateur. Nous allons maintenant procéder par programmation ; il faut donc comprendre comment est définie la position d'une vue sur l'écran.

Un système de coordonnées par vue

Dans la hiérarchie des vues dont est composée l'interface utilisateur, chaque vue dispose de son propre système de coordonnées. Sur iPhone OS, l'*origine* par défaut se situe en haut à gauche de la vue, l'axe des *abscisses* défile de gauche à droite et l'axe des *ordonnées* de haut en bas.

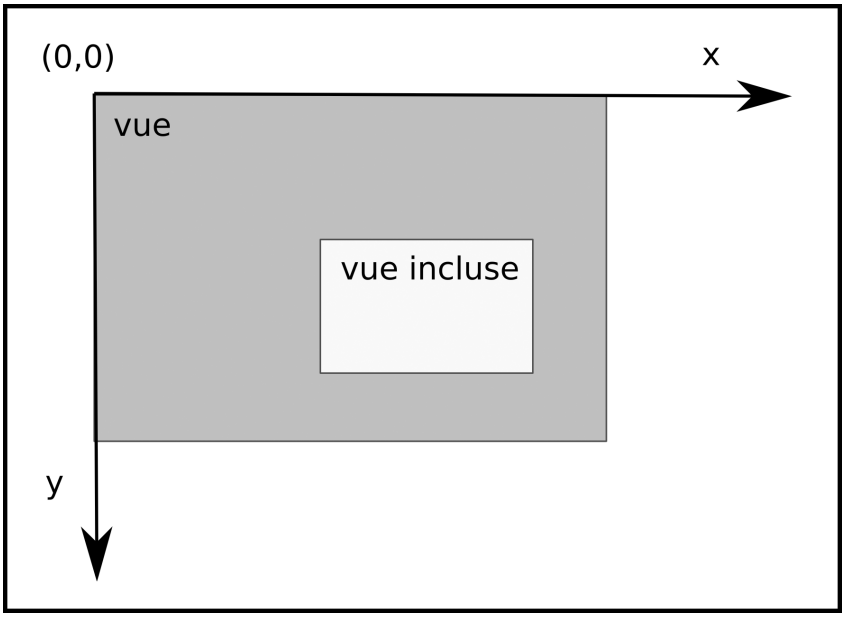


Figure 8.4 : Système de coordonnées d'une vue

ATTENTION

Mac OS X et iPhone OS

Pour les habitués de la programmation sur Mac OS X, sur lequel l'origine des coordonnées se situe en bas à gauche de la vue sous Cocoa, l'axe des ordonnées est inversé par rapport à Cocoa Touch.

Frame et Bounds

La position d'une vue incluse dans sa super-vue s'appelle le **cadre-** (*frame*) de la vue. C'est le rectangle, exprimé dans le système de coordonnées de la super-vue, dans lequel la vue est inscrite.

Un rectangle est défini par :

- un point ; l'*origine* du rectangle ;
- une *taille*, c'est-à-dire une *largeur* et une *hauteur*.

REMARQUE

Taille "négative"

La largeur ou la hauteur d'un rectangle peuvent être négatives. Leur signe permet de déterminer la position de l'origine : par exemple, si elles sont

positives, l'origine du rectangle est son angle en haut à gauche. Si elles sont toutes deux négatives, l'origine est son angle en bas à droite.

Une vue incluse peut elle-même contenir d'autres vues, il faut donc définir son système de coordonnées. Cela est fait indirectement en exprimant le *cadre* dans les coordonnées de la vue : les **limites-** (*bounds*) de la vue. Par défaut, le rectangle défini par les *limites* présente la même taille que le *cadre* et (`0.,0.`) pour origine.

Le cadre et les limites sont un seul et même rectangle, mais le *cadre* est exprimé dans les coordonnées de la *super-vue* alors que les *limites* le sont dans les coordonnées de la *vue incluse*.

Trois propriétés de la classe `UIView` sont interdépendantes, `frame` (le cadre), `bounds` (les limites) et `center` (les coordonnées du centre de la vue incluse, exprimées dans les coordonnées de la super-vue) :

- Lorsque `frame` est modifiée, `center` est recalculée et la taille de `bounds` prend la valeur de la taille de `frame`.
- Lorsque la taille de `bounds` est modifiée, elle est utilisée ainsi que `center` pour recalculer `frame`.
- Lorsque `center` est modifiée, l'origine de `frame` est recalculée.

Types C pour les éléments graphiques

Les propriétés `frame` et `bounds` sont de type `CGRect`, La propriété `center` est de type `CGPoint`, deux structures C.

Tableau 8.2 : Principales structures C utilisées pour les opérations graphiques

Structure	Composition	Utilisation
`CGRect`	`CGPoint origin`	Origine du rectangle
	`CGSize size`	Taille du rectangle
`CGPoint`	`CGFloat x`	Abscisse du point
	`CGFloat y`	Ordonnée du point
`CGSize`	`CGFloat width`	Largeur
	`CGFloat height`	Hauteur

S'agissant de structures C, leurs éléments sont donc accessibles par la notation pointée, par exemple `ball.center.x` pour l'abscisse du centre de la boule et `ball.center.y` pour son ordonnée.

CGFloat

Le type `CGFloat` est un synonyme de `float` défini dans le framework *CoreGraphics*. Nous avons déjà rencontré des types spécifiques à un framework, par exemple `NSInteger`. C'est une pratique qui permet d'améliorer la portabilité du code.

Les structures ne sont pas des classes

Les variables et propriétés graphiques sont généralement des structures, pas des références, contrairement aux objets qui sont toujours des références. En pratique, la déclaration des variables ne contient pas le caractère `*`. Ce sont des structures :

```
UIView * myView ;          // référence sur un objet
CGRect rect ;              // variable de type structure
```

Le framework CoreGraphics fournit plus de 30 fonctions pratiques pour manipuler ces structures géométriques, par exemple :

- des constructeurs :
  - `CGPoint CGPointMake (CGFloat x,CGFloat y) ;`
  - `CGSize CGSizeMake (CGFloat width,CGFloat height) ;`
  - `CGRect CGRectMake (CGFloat x,CGFloat y,CGFloat width,CGFloat height).`
- des comparateurs :
  - `bool CGRectContainsPoint (CGRect rect,CGPoint point) ;`
  - `CGRect CGRectIntersection (CGRect r1,CGRect r2).`
- des calculs d'informations :
  - `CGFloat CGRectGetMinX (CGRect rect) ;`
  - `CGFloat CGRectGetMaxY (CGRect rect).`

Vous trouverez la liste exhaustive de ces fonctions dans la documentation *CGGeometry Reference*.

Animer la boule de billard

Après cette introduction sur les concepts des coordonnées graphiques, revenons à notre application *Billard*.

Nous avons besoin de plusieurs éléments pour animer la boule de billard :

- la valeur d'un déplacement élémentaire ;
- une méthode qui réalise un déplacement élémentaire ;
- la fréquence des déplacements élémentaires ;
- un moyen de séquencer les déplacements élémentaires à la bonne fréquence.

1 Modifiez le fichier *BillardViewController.h* pour y ajouter la méthode `–moveBall` qui réalisera un déplacement élémentaire et les variables d'instances qui contiendront la valeur d'un déplacement élémentaire.

```
@interface BillardViewController : UIViewController {
   IBOutlet UIImageView *ball;
   CGFloat moveX;
   CGFloat moveY;
}
@property(nonatomic,retain) UIImageView *ball;
- (void)moveBall;
@end
```

2 Dans le fichier *BillardViewController.m*, définissez une constante `timerInterval` à la valeur souhaitée pour la fréquence de rafraîchissement, 1/30e de seconde :

```
#import "BillardViewController.h"
const float timerInterval = 1./30.;
@implementation BillardViewController
```

3 Modifiez la méthode `–viewDidLoad` pour initialiser la valeur du déplacement élémentaire et lancer le premier déplacement :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   moveX = 3.;
   moveY = -5.;
   [self moveBall];
}
```

4 Écrivez la méthode `–moveBall` :

```
- (void) moveBall {
   CGPoint center = ball.center;
   center.x += moveX;
   center.y += moveY;
   ball.center = center;
   [self performSelector:@selector(moveBall)
              withObject:nil afterDelay:timerInterval];
}
```

Après avoir déplacé la boule en modifiant sa propriété `center`, nous armons un *temporisateur* qui va réémettre le message `–moveBall` après une attente de `timerInterval` secondes. La méthode `–perform`

`Selector:withObject:afterDelay:` est disponible pour tous les objets, quelle que soit leur classe. Son paramètre `withObject:` est utilisé comme paramètre du message armé si le sélecteur attend un paramètre.

5 Construisez l'application et testez-la sur le simulateur. La boule se déplace correctement mais malheureusement, elle disparaît rapidement. Nous allons maintenant implémenter les rebonds sur les bandes de la table de billard.

Détecter les bandes

Dans les nombreux jeux que vous programmerez, il vous faudra surveiller les interactions entre différents objets. Le framework *CoreGraphics* propose la fonction `CGRectIntersectsRect` qui prend deux rectangles en paramètres et retourne `YES` si ces rectangles se recouvrent au moins en partie, et `NO` s'ils sont disjoints. Cette fonction est intéressante pour détecter si deux objets sont en contacts mais ne convient pas pour détecter si la boule de billard est sur le point de "sortir" de la table.

1 Modifiez la méthode `–moveBall` pour détecter si la boule sort de la table et éventuellement changer le déplacement élémentaire :

```
- (void) moveBall {
    // tableRect doit contenir les limites de l'écran
    CGRect tableRect = self.view.bounds;
    // ballRect doit contenir le cadre de la boule
    CGRect ballRect = self.ball.frame;
    if (CGRectGetMinX(ballRect)<CGRectGetMinX(tableRect)||
        CGRectGetMaxX(ballRect)>CGRectGetMaxX(tableRect)){
       moveX = -moveX;
    }
    if (CGRectGetMinY(ballRect)<CGRectGetMinY(tableRect)||
        CGRectGetMaxY(ballRect)>CGRectGetMaxY(tableRect)){
       moveY = -moveY;
    }
    CGPoint center = ball.center;
    center.x += moveX;
    center.y += moveY;
    ball.center = center;
    [self performSelector:@selector(moveBall)
                withObject:nil afterDelay:timerInterval];
}
```

Nous n'avons pas utilisé la fonction `CGRectContainsRect` qui teste si le second rectangle passé en paramètre est contenu en totalité dans le

premier car nous avons besoin de savoir si le débordement est dans le sens horizontal ou vertical pour modifier le déplacement élémentaire.

2 Reconstruisez et testez l'application ; la boule rebondit maintenant sur les bandes. Essayez d'augmenter le déplacement élémentaire pour accélérer le mouvement :

```
moveX = 15.;
moveY = -12.;
```

L'animation reste fluide. Le framework *CoreGraphics* est optimisé pour que le déplacement d'une vue ne nécessite pas de redessiner cette vue ou celle située en dessous : chaque vue possède son propre *calque* (*layer*).

Challenges

Challenge 1

Pour améliorer simplement le rendu du mouvement de la boule de billard, il faut modéliser le frottement sur la table et la perte d'énergie due aux chocs. Apportez cette amélioration dans la méthode `–moveBall`. Vous pouvez adopter une perte de vitesse de 10 % à chaque choc et de 0,5 % à chaque déplacement élémentaire.

Challenge 2

Pour atteindre un rendu très réaliste, ajoutez l'émission d'un son à chaque rebond de la boule sur une bande, comme nous l'avons fait précédemment dans ce chapitre.

Seul le premier son est émis lorsque deux rebonds sont trop rapprochés. Une solution pour corriger ce problème consiste à utiliser alternativement 2 lecteurs.

Où trouver des sons

Outre les ressources, sons et images, qui vous sont proposées avec le code source accompagnant cet ouvrage, vous trouverez de nombreux effets sonores de bonne qualité sur le site **http://www.soundsnap.com**.

Challenge 3

Plus difficile, restructurez l'application pour pouvoir positionner plusieurs boules sur la table. Il faut bien sûr détecter et traiter les collisions entre les boules.

Pour calculer les vitesses des deux boules après l'impact, vous inspirez-vous du code C que vous trouverez sur le site http://fr.wikipedia.org/wiki/Choc_élastique.

8.2. Dessiner avec Quartz2D

Cette section est consacrée aux moyens de composer et d'optimiser le tracé d'un dessin pour conserver une bonne fluidité des animations.

Afin d'illustrer ces techniques, nous visualiserons la trajectoire de la boule, pendant son déplacement, dans notre application *Billard*.

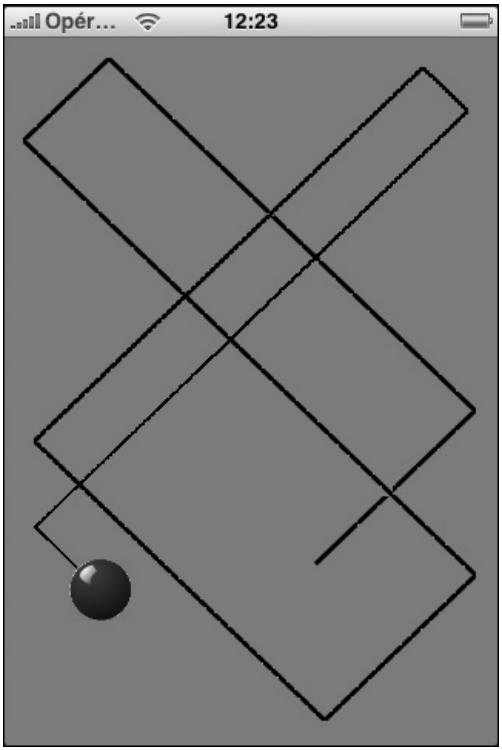


Figure 8.5 : Tracé de la trajectoire

Principe de fonctionnement

Précisions sur la classe UIView

Tous les objets qui s'affichent à l'écran, les *vues*, dérivent directement ou indirectement de la classe `UIView`. Elle a la responsabilité de gérer :

- la hiérarchie des vues ;
- l'affichage ;
- l'animation des vues.

La méthode `-drawRect:` de la classe `UIView` dessine le contenu de la vue. Cette méthode est appelée par le framework lorsque la vue est affichée la première fois, puis lorsque son contenu évolue. Si vous voulez personnaliser l'aspect graphique de vos applications, il vous faudra donc définir votre propre classe dérivée de `UIView`, et coder le comportement graphique souhaité dans la méthode `-drawRect:`.

Le paramètre passé à cette méthode est le rectangle dans lequel le dessin doit être exécuté. Par défaut, le contenu de ce rectangle est effacé avant l'appel de `-drawRect:`, son contenu doit alors être complètement redessiné par la méthode.

La vue que vous définissez devra vraisemblablement avoir un affichage variable, par exemple en fonction des valeurs des propriétés. Lorsque l'affichage doit évoluer, il faut appeler l'une des deux méthodes :

- `-setNeedsDisplay` pour redessiner toute la vue.
- `-setNeedsDisplayInRect:`. La partie à redessiner est l'intérieur du rectangle passé en paramètre qui sera retransmis à `-drawRect:`.

Ne pas appeler drawRect

Vous ne devez pas appeler la méthode `-drawRect:` directement. Vous devez appeler l'une des méthodes `-setNeedsDisplay` ou `-setNeedsDisplayInRect:` pour informer le framework qu'il doit appeler `-drawRect:`.

Contexte graphique

Les fonctions permettant de dessiner sont regroupées dans le framework *CoreGraphics*. Elles prennent pratiquement toutes pour premier paramètre un pointeur sur le contexte graphique, leur permettant de savoir "où" dessiner. Lorsque la méthode `-drawRect:` est appelée, un *contexte graphique* adéquat est initialisé par défaut, avec le système de coordonnées défini par les *limites* de la vue.

Généralement, le code de la méthode `-drawRect:` commence par l'obtention du contexte graphique courant.

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

Contexte graphique

Le *contexte graphique* permet de faire le lien entre les fonctions graphiques et la destination de l'image. Que le dessin soit tracé sur l'écran ou dans un fichier PDF, le développeur utilise les mêmes fonctions graphiques. Il n'a pas à se préoccuper de la destination de l'image qu'il compose, c'est le contexte graphique qui prend en charge les opérations détaillées.

Les fonctions graphiques sont des fonctions C

Pour des raisons de performances, les fonctions graphiques n'ont pas été développées sous forme de classes Objective-C mais regroupées dans une bibliothèque de fonctions C.

Mise en pratique

Avant d'aller plus loin, mettons en pratique ce que nous venons d'apprendre : `-drawRect:` et le contexte graphique.

Nous allons réaliser une classe dérivée de `UIView` pour effectuer le tracé de la trajectoire de la boule de billard. Cette classe `SnookerView` comportera deux propriétés qui permettront au contrôleur de vue de lui transmettre les mouvements de la boule :

- `lastPoint`, position finale de la boule, mise à jour à chaque mouvement élémentaire ;
- `drawing`, booléen indiquant si le tracé doit être réalisé, indispensable pour commencer le dessin uniquement après avoir indiqué la position initiale de la boule.

Un trait doit être dessiné à chaque mouvement élémentaire entre la dernière position de la boule et sa nouvelle position.

Créer la classe SnookerView

1. Sous XCode, créez les fichiers sources *.m* et *.h* pour la classe `SnookerView`, en indiquant que cette classe hérite de `UIView` (voir Figure 8.6).
2. Modifiez le fichier *SnookerView.h* pour y déclarer les propriétés de la nouvelle classe :

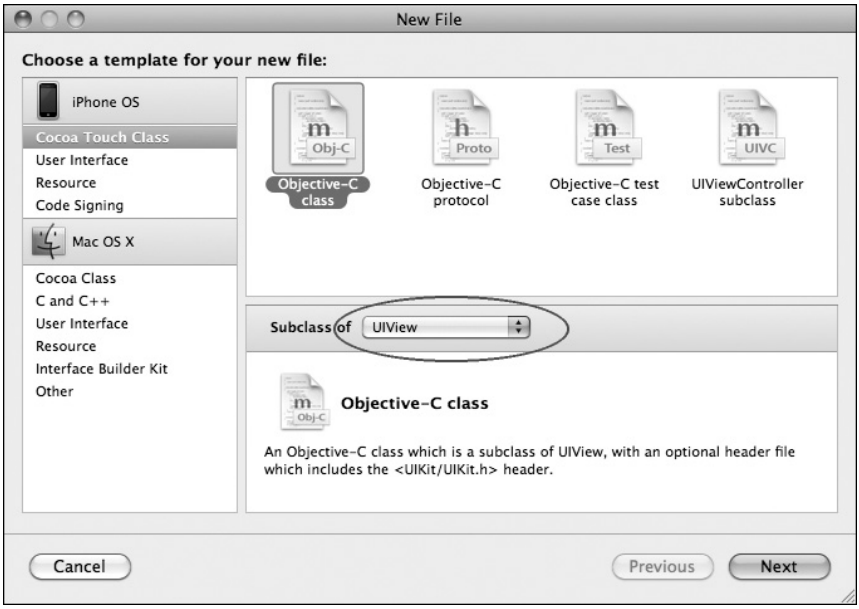


Figure 8.6 : Création d'une classe dérivée de UIView

```
@interface SnookerView : UIView {
   BOOL drawing;
   CGPoint precedingLastPoint;
   CGPoint lastPoint;
}
@property(nonatomic,getter=isDrawing) BOOL drawing;
@property(nonatomic) CGPoint lastPoint;
@end
```

Nous déclarons :

- une variable d'instance `precedingLastPoint` qui n'est pas définie comme une propriété ; nous aurons besoin de retenir la position précédente de la boule pour effectuer le tracé mais les utilisateurs de la classe `SnookerView` n'ont pas besoin d'y accéder ;
- un accesseur dénommé `isDrawing` plutôt que `drawing` ; c'est une pratique courante pour les propriétés de type `BOOL`.

3 Ajoutez la définition de la méthode `-setLastPoint:` dans le fichier *SnookerView.m* :

```
-(void)setLastPoint:(CGPoint)aPoint{
   precedingLastPoint = lastPoint;
   lastPoint = aPoint;
   if (drawing) {
```

```
            [self setNeedsDisplayInRect:
          CGRectMake(precedingLastPoint.x, precedingLastPoint.y,
                     lastPoint.x-precedingLastPoint.x,
                     lastPoint.y-precedingLastPoint.y)];
        }
    }
```

Nous modifions le manipulateur par défaut de la propriété `lastPoint` car nous avons deux choses importantes à faire lors de chaque modification de cette propriété :

- enregistrer la position précédente de la boule ;
- informer le framework que cette vue doit être redessinée dans le rectangle dont une diagonale est définie par le dernier point et le point précédent.

Dessiner le tracé

Écrivez le code de la méthode `–drawRect:` :

```
- (void)drawRect:(CGRect)rect {
   CGContextRef context = UIGraphicsGetCurrentContext();
   CGContextSetRGBStrokeColor(context, 1., 0.5, 0., 1.0);
   CGContextSetLineWidth(context, 3.);
   CGContextSetLineCap(context, kCGLineCapSquare);
   CGPoint segment[2] = {precedingLastPoint,self.lastPoint};
   CGContextStrokeLineSegments(context, segment, 2);
}
```

Nous reconnaissons la première instruction qui permet d'obtenir une référence au contexte graphique courant, vers lequel toutes les commandes graphiques seront transmises.

Nous voulons que l'appel de cette méthode provoque le tracé d'un trait entre le dernier point (propriété `lastPoint`) et le point précédent (variable d'instance `precedingLastPoint`). Cela est accompli par l'appel de la fonction `CGContextStrokeLineSegments` qui prend en paramètres un tableau de points et le nombre de points contenus dans le tableau. Cette fonction est utilisable pour tracer une succession de segments.

Trois autres fonctions sont employées pour définir les attributs graphiques du trait à tracer :

- `CGContextSetRGBStrokeColor` permet de définir les composants Rouge, Vert, Bleu et Alpha de la couleur du trait.
- `CGContextSetLineWidth` permet de définir la largeur du trait.
- `CGContextSetLineCap` permet de définir le tracé de l'extrémité des segments. Le paramètre de cette fonction est une constante.

Tableau 8.3 : Paramètres de la fonction CGContextSetLineCap		
Forme de terminaison	**Paramètre à utiliser**	**Commentaire**
	`kCGLineCapButt`	Le trait est arrêté à l'extrémité du segment.
	`kCGLineCapRound`	Le trait est arrondi autour de l'extrémité du segment.
	`kCGLineCapSquare`	Le trait est carré autour de l'extrémité du segment.

Il existe d'autres fonctions graphiques mais nous vous demandons un peu de patience. Il nous reste à utiliser notre nouvelle classe dans le contrôleur de vue pour terminer l'application.

Utiliser la nouvelle classe

Modifier le fichier NIB

Afin d'utiliser les propriétés et les méthodes que nous venons de définir, il faut modifier la classe de la vue dans le fichier *NIB*.

1. Ouvrez le fichier *BillardViewController.xib* dans Interface Builder.
2. Sélectionnez la vue principale et indiquez qu'elle doit être de la classe `SnookerView` dans l'inspecteur d'identité (⌘+4).

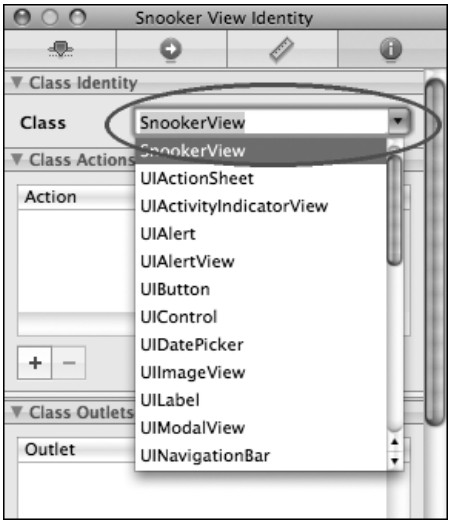


Figure 8.7 : La vue principale doit appartenir à la classe SnookerView

Modifier le contrôleur de vue

1 Modifiez la méthode `–viewDidLoad` dans le fichier *BillardViewController.m* pour initialiser le tracé du déplacement :

```
- (void)viewDidLoad {
    [super viewDidLoad];
    moveX = 15.;
    moveY = -12.;
    [(SnookerView *)self.view setLastPoint: ball.center];
    [(SnookerView *)self.view setDrawing:YES];
    [self moveBall];
}
```

cast `(SnookerView *)`
La propriété `view` est déclarée comme une `UIView` dans la classe `UIViewController`, et ne dispose pas des propriétés `lastPoint` et `drawing`. Nous employons donc l'instruction de changement de type (*cast*) `(SnookerView *)` pour éviter un message d'avertissement à la compilation.

2 Modifiez la méthode `–moveBall` pour réaliser le tracé de la trajectoire à chaque déplacement élémentaire :

```
- (void) moveBall {
    CGRect tableRect = self.view.bounds;
    CGRect ballRect = self.ball.frame;
    if (CGRectGetMinX(ballRect)<CGRectGetMinX(tableRect)||
        CGRectGetMaxX(ballRect)>CGRectGetMaxX(tableRect)){
       moveX = -moveX;
    }
    if (CGRectGetMinY(ballRect)<CGRectGetMinY(tableRect)||
        CGRectGetMaxY(ballRect)>CGRectGetMaxY(tableRect)){
       moveY = -moveY;
    }
    CGPoint center = ball.center;
    center.x += moveX;
    center.y += moveY;
    ball.center = center;
    [(SnookerView*)self.view setLastPoint: center];
    [self performSelector:@selector(moveBall)
                withObject:nil afterDelay:timerInterval];
}
```

3 Reconstruisez l'application et vérifiez que la trajectoire est dessinée correctement.

Cette application démontre deux caractéristiques importantes du framework *CoreGraphics* pour l'optimisation du dessin et des animations :

- Seul le contenu du rectangle passé en paramètre à la méthode `-drawRect:` doit être redessiné.
- Chaque vue étant dessinée dans son propre calque, la boule se superpose au tracé de la trajectoire. Elle apparaît bien au-dessus de la table et il n'est pas nécessaire de redessiner ce qui était sous la boule lorsque celle-ci se déplace.

Primitives graphiques

Les principales primitives graphiques sont résumées dans le tableau ci-après.

Tableau 8.4 : Principales primitives graphiques

Thème	Signature	Objet
Contexte Graphique	`CGContextRef UIGraphicsGet CurrentContext (void)`	Retourne le contexte graphique par défaut.
Attributs graphiques	`void CGContextSet LineCap ( CGContextRef c, CGLineCap cap)`	Définit le type de terminaison du tracé des segments de droite.
	`void CGContextSet LineWidth ( CGContextRef c, CGFloat width)`	Définit la largeur du tracé des segments de droite.
	`void CGContextSet RGBFillColor ( CGContextRef c, CGFloat red, CGFloat green, CGFloat blue, CGFloat alpha)`	Définit les composantes de la couleur de remplissage pour les formes géométriques.
	`void CGContextSet RGBStrokeColor ( CGContextRef c, CGFloat red, CGFloat green, CGFloat blue, CGFloat alpha)`	Définit les composantes de la couleur de tracé.

Tableau 8.4 : Principales primitives graphiques

Thème	Signature	Objet
Fonctions de dessin	`void CGContext FillRect ( CGContextRef c, CGRect rect)`	Peint le contenu du rectangle passé en paramètre avec la couleur de remplissage préalablement définie.
	`void CGContext FillEllipseInRect ( CGContextRefcontext, CGRect rect)`	Peint le contenu de l'ellipse définie par le rectangle passé en paramètre avec la couleur de remplissage préalablement définie.
	`void CGContext StrokeRect ( CGContextRef c, CGRect rect)`	Trace le contour du rectangle passé en paramètre avec la couleur de tracé préalablement définie.
	`void CGContext StrokeEllipseInRect ( CGContextRefcontext, CGRect rect)`	Trace le contour de l'ellipse définie par le rectangle passé en paramètre avec la couleur de tracé préalablement définie.
	`void CGContext StrokeLineSegments ( CGContextRef c, const CGPointpoints[], size_t count)`	Trace la suite de segments dont les points sont dans le tableau passé en paramètres.

N'hésitez pas à consulter la documentation Apple et à essayer les nombreuses fonctions graphiques. Le framework *Core Graphics* est très riche et permet notamment de :

- tracer des arcs, des courbes de Bézier, des motifs et des lignes discontinues ;
- définir des dégradés de couleurs, des ombres ;
- réaliser des rotations ou d'autres transformations ;
- dessiner du texte, etc.

8.3. Débuter la 3D avec OpenGL ES

Le graphisme en trois dimensions permet de représenter des scènes très réalistes : ombres portées, textures, sources de lumière, brillance, etc. Malheureusement, ce résultat est obtenu au prix d'une grande complexité ; le livre *OpenGL superbible* édité par Addison Wesley compte 1 200 pages. Nous allons limiter notre ambition dans cette section qui est destinée à ceux d'entre vous qui connaissent déjà OpenGL ES et souhaitent savoir comment le mettre en œuvre sur iPhone OS :

- présenter *OpenGL ES* utilisé sur iPhone OS pour le graphisme en trois dimensions ;
- expliquer comment cette bibliothèque standard est exploitée dans une application *Cocoa Touch*.

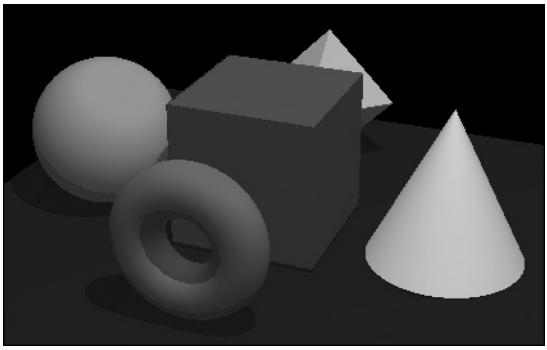


Figure 8.8 : Exemple de graphisme 3D

Présentation d'OpenGLES

OpenGL est une bibliothèque graphique (*Graphics Library*) standard accessible en langage C, donc en Objective-C. Elle est largement utilisée dans nombre d'applications professionnelles ou ludiques. *OpenGL ES* est une version allégée d'OpenGL conçue pour les appareils mobiles (*Embedded Systems*).

OpenGL ES existe en deux versions, prises en charge sur iPhone OS. Le développeur choisira celle qu'il souhaite utiliser :

- la version 1.1 est une bibliothèque classique de primitives graphiques.
- la version 2.0 permet de programmer des fonctions qui s'exécutent directement sur le processeur graphique.

La documentation de référence officielle, en anglais, se trouve sur les sites suivants :

- http://www.khronos.org/opengles/sdk/1.1/docs/man/ pour la version 1.1 ;
- http://www.khronos.org/opengles/sdk/docs/man/ pour la version 2.0.

Intégration dans Cocoa Touch

L'utilisation d'*OpenGL ES* nécessite de la part du développeur un effort supplémentaire. Non seulement il faut connaître la bibliothèque et les principes du graphisme 3D, mais il faut aussi comprendre comment OpenGL ES et *Core Animation* travaillent ensemble.

Il n'y a pas dans *Cocoa Touch*, contrairement à *Cocoa* sur Mac OS X, d'objet de type Vue prêt à l'emploi pour utiliser *OpenGL ES*. Il nous faudra dériver une classe d'`UIView`, appelons-la `EAGLView`, pour nous conformer au modèle d'application proposé par XCode.

Frameworks

Pour utiliser OpenGLES dans un projet sous XCode, il faut y ajouter les frameworks *QuartzCore* et *OpenGLES*. Sélectionnez la cible (*target*) du projet puis activez la commande **Existing Framework** ... du sous-menu **Add** du menu contextuel.

Modèle OpenGL ES
Les frameworks nécessaires sont inclus dans le modèle de projet **OpenGL ES Application** sous XCode.

Les déclarations à importer sont les suivantes :

```
#import <QuartzCore/QuartzCore.h>
#import <OpenGLES/ES1/gl.h>            // pour OpenGL ES 1.1
#import <OpenGLES/ES1/glext.h>         // pour OpenGL ES 1.1
#import <OpenGLES/ES2/gl.h>            // pour OpenGL ES 2.0
#import <OpenGLES/ES2/glext.h>         // pour OpenGL ES 2.0
#import <OpenGLES/EAGL.h>
#import <OpenGLES/EAGLDrawable.h>
```

Calque OpenGL ES

Nous savons déjà que chaque instance de la classe `UIView`, ou d'une classe dérivée, possède son propre *calque*. Il est accessible via la propriété `layer` de type `CALayer*`. La vue et le calque associé étant intimement liés, la classe `UIView` définit une méthode `+layerClass` qui retourne la classe à utiliser comme calque. Cette classe doit dériver de `CALayer` qui est le défaut.

Pour utiliser la bibliothèque OpenGL ES, il faut utiliser un calque de la classe `CAEAGLLayer`. Notre classe `EAGLView` devra donc modifier la méthode `+layerClass` de la classe `UIView` :

```
+ (Class) layerClass {
   return [CAEAGLLayer class];
}
```

Contexte graphique

Les primitives graphiques d'*OpenGL ES*, comme celles de *Core Graphics*, sont dirigées vers un contexte graphique. Dans le cas

d'OpenGL ES, le contexte graphique doit être une instance de la classe `EAGLContext` créée à l'initialisation de la vue :

```
context = [[EAGLContext alloc]
                  initWithAPI:kEAGLRenderingAPIOpenGLES1];
```

Le contexte est créé soit avec la constante `kEAGLRenderingAPIOpenGLES1`, soit avec la constante `kEAGLRenderingAPIOpenGLES2` suivant que l'on veut travailler en version 1.1 ou en version 2.0.

Le contexte auquel toutes les commandes graphiques doivent être transmises est ensuite spécifié par l'instruction :

```
[EAGLContext setCurrentContext:context];
```

Zones tampons

Lorsque l'application compose une image, elle ne travaille pas directement sur l'écran. Elle utilise des **zones tampons** (*buffers*) spécifiques, suivant le type d'informations (couleur, profondeur, pochoir) qui servent à composer l'image, liées entre elles dans un **cadre tampon** (*framebuffer*). Une fois l'image composée, elle est transmise en une fois sur l'écran.

Chaque tampon est repéré par un identificateur qui est un nombre entier. Les tampons sont généralement créés à l'initialisation de la vue.

Créer le cadre tampon

Sous OpenGL ES v1.1, le *cadre tampon* est créé puis lié au contexte graphique par la suite d'instructions :

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

La fonction `glGenFrameBuffersOES` est employée ici pour créer un seul cadre tampon. Elle prend deux paramètres :

- le nombre de cadres tampons à créer ;
- un tableau d'entiers dans lequel les identifiants des cadres créés seront rangés.

La fonction `glBindFramebufferOES` est utilisée pour lier le cadre tampon nouvellement créé au contexte courant, afin d'en faire la destination des commandes graphiques à venir.

Extension OES

Les fonctions de gestion des *zones tampons* et du *cadre tampon* sont définies dans la version 2.0 d'OpenGL ES. Sous iPhone OS, on utilise le même

système de fonctions avec la version 1.1. Les noms de ces fonctions se terminent dans ce cas par `OES`, nom de l'extension Apple à OpenGL ES 1.1.

À partir de cette section, les exemples seront donnés en version 1.1. Il sera facile d'en déduire l'utilisation en version 2.0 en enlevant les caractères `OES` à la fin des fonctions et constantes.

Le cadre tampon est susceptible de regrouper :

- Une zone tampon pour les couleurs (*color buffer*) dans laquelle seront calculées les couleurs de l'image à dessiner. Cette zone tampon est obligatoire.
- Une zone tampon pour la profondeur (*depth buffer*) permettant de déterminer les parties cachées de l'image. Cette zone tampon n'est pas utilisée pour les images en deux dimensions.
- Optionnellement, une zone tampon pour les pochoirs (*stencil buffer*) ou une zone tampon pour les textures (*texture buffer*).

Créer la zone tampon des couleurs

Sous OpenGL ES v1.1, la *zone tampon des couleurs* est créée puis liée au contexte graphique par la suite d'instructions :

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES,
                                        colorRenderbuffer);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES,
            GL_COLOR_ATTACHMENT0_OES,GL_RENDERBUFFER_OES,
                                        colorRenderbuffer);
```

La fonction `glFramebufferRenderbufferOES` permet d'attacher une zone tampon à un cadre tampon. Le deuxième paramètre permet de spécifier le type d'attachement :

- `GL_COLOR_ATTACHMENT0_OES` pour la zone tampon des couleurs ;
- `GL_DEPTH_ATTACHMENT_OES` pour la zone tampon de profondeur ;
- `GL_STENCIL_ATTACHMENT_OES` pour la zone tampon des pochoirs.

Les autres zones tampons éventuellement nécessaires sont créées et attachées de la même façon.

Définir le port OpenGL

Le **port** OpenGL (*view port*) est la zone sur l'écran dans laquelle doivent être effectués les tracés graphiques. Il doit donc être lié à la vue `EAGLView` et plus précisément à son calque. Mais auparavant, il

faut le dimensionner, ce qui ne peut être fait qu'après que la vue soit concrètement disposée sur l'écran. C'est pourquoi les instructions suivantes sont généralement placées dans la méthode `–layoutSubviews` de la classe `EAGLView` :

```
GLint viewWidth, viewHeight;
[context renderbufferStorage:GL_RENDERBUFFER_OES
                                fromDrawable:self.layer];
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES,
                    GL_RENDERBUFFER_WIDTH_OES, &viewWidth);
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES,
                  GL_RENDERBUFFER_HEIGHT_OES, &viewHeight);
glViewport(0, 0, viewWidth, viewHeight);
```

Le contexte graphique est d'abord attaché au calque de la vue ; il en prend donc les dimensions. Puis ces dimensions sont récupérées pour définir le système de coordonnées du port OpenGL.

Utiliser le contexte graphique

Lorsque toutes les zones tampons ont été créées et attachées au cadre tampon, on peut tester la bonne configuration de ce dernier, par exemple de la façon suivante :

```
if (glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) !=
                              GL_FRAMEBUFFER_COMPLETE_OES) {
   NSLog(@"Echec lors de la création du cadre tampon %x",
         glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES));
   return;
}
```

Le contexte graphique est enfin prêt pour recevoir les instructions de dessin. Lorsque l'image est prête, elle peut être affichée à l'écran :

```
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

Lorsque l'image a été transmise à l'écran, le tampon des couleurs est réinitialisé. Dans le cas d'une animation, il faut recomposer complètement l'image après l'affichage de chaque trame.

Exemple d'application

Afin d'illustrer les concepts que nous venons de voir, nous allons analyser un exemple d'application : le modèle *OpenGL ES Application* proposé par XCode.

Sous XCode, créez un projet de type *OpenGL ES Application*. Construisez l'application et testez-la sur le simulateur ; un carré rempli par un dégradé de couleurs se balance doucement. Vous pouvez créer une application pour iPhone ou pour iPad ; elles fonctionnent de la même façon.

Structure des classes

Examinez les fichiers du groupe *Classes*, le modèle proposé par XCode contient :

- une classe pour le délégué d'application ;
- une classe `EAGLView`, qui dérive de `UIView` ;
- deux classes `ES1Renderer` et `ES2Renderer` ;
- un protocole `ESRenderer`.

Ce modèle ne contient pas de contrôleur de vue spécifique, c'est le délégué d'application qui gère la vue directement : il a une propriété `glView` et contrôle le fonctionnement de l'animation. Les applications OpenGL ES ne respectent pas le modèle MVC.

L'aspect graphique est réparti en trois classes :

- `ES1Renderer` contient les instructions spécifiques à *OpenGL ES 1.1*.
- `ES2Renderer` contient les instructions spécifiques à *OpenGL ES 2.0*.
- `EAGLView` contient le code indépendant de la version utilisée et le mécanisme d'aiguillage entre les deux versions d'OpenGL ES.

Le protocole `ESRenderer` permet au code d'`EAGLView` de fonctionner indifféremment avec l'une des classes `ES1Renderer` ou `ES2Renderer`.

C'est dans l'une des classes `ES1Renderer` ou `ES2Renderer`, suivant que vous adoptez la version 1.1 ou la version 2.0, que vous devez mettre votre code *OpenGL ES*, plus particulièrement dans la méthode `-render`. Ouvrez le fichier *ES1Renderer.m* et vérifiez que vous localisez les instructions de paramétrage du contexte graphique et des zones tampons vus dans les sections précédentes.

Nous allons nous concentrer sur la classe `EAGLView` qui offre quelques particularités que nous n'avons pas encore vues.

Classe EAGLView

La classe `EAGLView` prend en charge :

- La configuration du calque (layer). En particulier, elle implémente la méthode `+layerClass` pour indiquer qu'il faut utiliser un calque OpenGL ES.
- Le pilotage de l'animation.
- Le choix de la version d'OpenGL ES, en l'occurrence 1.1 uniquement si la version 2.0 ne fonctionne pas sur l'appareil.

- L'interface avec les deux méthodes du protocole `ESRenderer`.
  - `-resizeFromLayer:` pour finaliser l'initialisation du contexte graphique lorsque la dimension définitive de la vue est connue ;
  - `-render` pour dessiner l'image.

Pilotage de l'animation

Ouvrez le fichier *EAGLView.m* pour étudier la méthode employée pour rythmer l'animation.

```
- (void) startAnimation {
   if (!animating) {
      if (displayLinkSupported) {
         displayLink = [NSClassFromString(@"CADisplayLink")
               displayLinkWithTarget:self
                            selector:@selector(drawView:)];
         [displayLink
                  setFrameInterval:animationFrameInterval];
         [displayLink
                    addToRunLoop:[NSRunLoop currentRunLoop]
                         forMode:NSDefaultRunLoopMode];
      }
      else
         animationTimer =
    [NSTimer scheduledTimerWithTimeInterval:
    (NSTimeInterval)((1.0 / 60.0) * animationFrameInterval)
                    target:self
                  selector:@selector(drawView:)
                  userInfo:nil
                   repeats:TRUE];
      animating = TRUE;
   }
}
```

Nous voyons que l'une des deux classes suivantes est utilisée :

- `CADisplayLink` ;
- ou `NSTimer`, le temporisateur universel.

L'avantage de `CADisplayLink` sur `NSTimer` est que le temps nécessaire pour calculer une nouvelle image est pris en compte afin de fixer le rythme de rafraîchissement ; c'est la classe à favoriser pour synchroniser une animation. Elle n'est malheureusement disponible qu'à partir de la version 3.1 de l'iPhone OS, c'est pourquoi la classe `EAGLView` possède une propriété `displayLinkSupported` qui est évaluée dans la méthode `-initWithCoder:` :

```
displayLinkSupported = FALSE;
NSString *reqSysVer = @"3.1";
NSString *currSysVer =
```

```
                [[UIDevice currentDevice] systemVersion];
if ([currSysVer compare:reqSysVer
          options:NSNumericSearch] != NSOrderedAscending)
   displayLinkSupported = TRUE;
```

NSClassFromString

Dans le code de la méthode `–startAnimation`, Il faut utiliser l'appel de fonction `NSClassFromString(@"CADisplayLink")` plutôt que simplement la classe `CADisplayLink` pour éviter les erreurs à la construction de l'application si l'on emploie un SDK qui ne contient pas cette classe.

Classe UIDevice

Nous avons vu précédemment un exemple de quelques instructions permettant de connaître les caractéristiques de l'appareil sur lequel l'application s'exécute. Le tableau ci-après résume quelques-unes des propriétés des instances de la classe `UIDevice`.

Tableau 8.5 : Principales propriétés de la classe UIDevice

Thème	Signature	Objet
Obtenir l'instance courante	`+ (UIDevice *)currentDevice`	Retourne l'instance représentant l'appareil sur lequel l'application s'exécute.
Identifier l'appareil et le système d'exploitation	`@property (nonatomic, readonly, retain) NSString *uniqueIdentifier`	Identifiant unique de l'appareil (UDID)
	`@property (nonatomic, readonly, retain) NSString *name`	Nom de l'appareil
	`@property (nonatomic, readonly, retain) NSString *systemVersion`	Version du système d'exploitation
	`@property (nonatomic, readonly, retain) NSString *model`	Modèle d'appareil, actuellement retourne `@"iPhone"`, `@"iPod touch"`, `@"iPad"`, `@"iPhone Simulator"` ou `@"iPad Simulator"`.
État de la batterie	`@property (nonatomic, readonly) float batteryLevel`	Niveau de charge de la batterie. Retourne une valeur comprise entre `0` (0 %) et `1` (100 % de charge).

8.4. Checklist

Ce chapitre nous a permis d'explorer quelques-unes des possibilités graphiques et d'animation d'iPhone OS :

- animation d'une image, avec les classes `UIImageView` et `UIImage` ;
- déplacement d'une image sur l'écran ;
- primitives graphiques de *Quartz2D* :
  - le cadre (*frame*), les limites (*bounds*) et les différents systèmes de coordonnées ;
  - les types de données `CGRect`, `CGPoint` et `CGSize` ;
  - la méthode `-drawRect:`.
- intégration de la bibliothèque *OpenGL ES* pour le graphisme en trois dimensions :
  - l'existence des deux versions 1.1 et 2.0 ;
  - le paramétrage du contexte graphique ;
  - les zones tampons et le cadre tampon.

Nous avons agrémenté nos applications avec des effets sonores à l'aide de la classe `AVAudioPlayer` et vu comment les animer avec la méthode `-performSelector:withObject:afterDelay:` ou la classe `CADisplayLink`.

Nous avons également exposé la classe `UIDevice` qui permet de connaître les caractéristiques de l'appareil courant.

TAPES, TOUCHES ET GESTES

Nous avons découvert le mécanisme *cible-action* au chapitre 2. Il nous a permis de réaliser des applications qui réagissent aux actions de l'utilisateur :

- édition d'un champ de texte ;
- appui sur un bouton ;
- changement de valeur d'un sélectionneur.

L'iPhone OS nous permet aussi de proposer à l'utilisateur une interface élaborée avec des gestes complexes, à un ou plusieurs doigts. Leur mise en œuvre dans une application nécessite d'avancer dans notre compréhension des événements (*events*) gérés par Cocoa Touch.

Nous commencerons par explorer les classes et techniques de base mises en jeu puis développerons quelques applications mettant en œuvre les gestes courants sur iPhone OS.

Gestes sous iPhone OS 3.2

La version 3.2 d'iPhone OS, disponible sur iPad, permet une mise en œuvre simplifiée des gestes standard (pincement, déplacement, glissement, etc.) par le mécanisme cible-action. Cette mise en œuvre sera détaillée dans le chapitre consacré aux spécificités de l'iPad.

Ce chapitre concerne donc principalement le développement sur iPhone et iPod Touch. Il est destiné également à ceux qui souhaitent développer leur propre analyseur de geste pour iPad.

9.1. Comprendre les événements

Classe UIResponder

Le mécanisme *cible-action* est mis en œuvre par les objets de la classe `UIControl` qui dérive indirectement de la classe `UIResponder` avec laquelle nous avons fait connaissance au chapitre 4. (voir Figure 9.1)

Lorsqu'un événement survient, l'application – plus précisément l'instance unique de la classe `UIApplication` – recherche le répondeur (une instance de la classe UIResponder) approprié et lui transmet l'événement :

- Si l'événement est une action sur l'écran, le répondeur est la vue située sous le doigt de l'utilisateur.

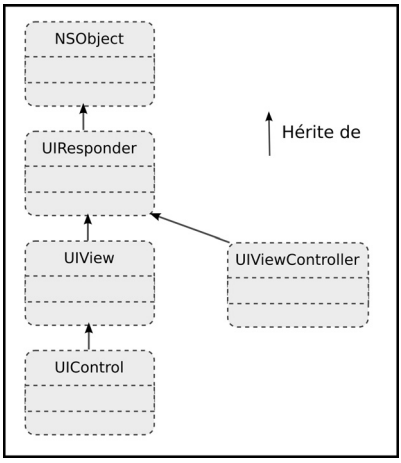


Figure 9.1 : Classes gestionnaires des événements

- Dans le cas contraire, l'événement est transmis au premier répondeur (*First responder*) puis remonte la chaîne des répondeurs jusqu'à ce que l'un d'eux accepte de le traiter.

Si le répondeur est un *contrôle* (une instance de la classe `UIControl`), l'événement est susceptible d'être utilisé pour déclencher le mécanisme *cible-action*. Nous allons nous intéresser ici à la façon dont les événements sont reçus par un répondeur pour définir nos propres comportements dans des vues ou des contrôleurs spécifiques.

Événements élémentaires

Les événements reçus par un répondeur peuvent être de deux sortes représentées par le type énuméré `UIEventType` :

- `UIEventTypeTouches` pour les touches sur l'écran ;
- `UIEventTypeMotion` pour les mouvements de l'appareil.

Le type énuméré `UIEventSubtype` est également défini. Dans la version actuelle, les sous-types concernent uniquement les mouvements de l'appareil :

- `UIEventSubtypeNone`, pas de sous-type particulier ;
- `UIEventSubtypeMotionShake`, mouvement de *secousse* de l'appareil (*shake*).

Les événements concernant les touches sont plus complexes ; chaque touche élémentaire peut être :

- la pose du doigt sur l'écran ;

- le déplacement du doigt sur l'écran ;
- le retrait du doigt de l'écran.

Touche

Une touche est un événement élémentaire concernant seulement un doigt : pose du doigt sur l'écran, déplacement du doigt sur l'écran ou retrait du doigt de l'écran.

En outre, pour décrire la touche élémentaire, il faut également préciser :

- La *position* de la touche. Cocoa Touch fournit un point dans les coordonnées de la vue concernée, bien que la taille d'un doigt normal recouvre plusieurs points lorsqu'il touche l'écran.
- Le *moment* précis auquel l'événement est intervenu. Il est donné par le nombre de secondes écoulées depuis le démarrage de l'appareil.

Un petit point pour un gros doigt

La zone touchée par un doigt sur l'écran est généralement de forme ellipsoïdale, de taille variable en fonction du doigt et de la pression exercée. Le système *Multi-Touch* analyse cette information pour calculer un point unique associé à la touche.

Toutes ces informations sont présentées dans une instance de la classe `UITouch` décrite dans le tableau ci-après.

Tableau 9.1 : Méthodes et propriétés de la classe UITouch

Thème	Signature	Objet
Emplacement des touches	`– (CGPoint)locationInView:(UIView *)view`	Retourne l'emplacement de la touche dans le système de coordonnées de la vue passée en paramètre, ou dans le système de coordonnées de la fenêtre si `nil` est passé en paramètre.
	`– (CGPoint)previousLocationInView:(UIView *)view`	Retourne l'emplacement précédent de la touche dans le système de coordonnées de la vue passée en paramètre, ou dans le système de coordonnées de la fenêtre si `nil` est passé en paramètre.
	`@property(nonatomic, readonly, retain) UIView *view`	La vue dans laquelle la touche a débuté.
	`@property(nonatomic, readonly, retain) UIWindow *window`	La fenêtre dans laquelle la touche a débuté.

Tableau 9.1 : Méthodes et propriétés de la classe UITouch

Thème	Signature	Objet
Attributs de la touche	`@property(nonatomic, readonly) NSUInteger tapCount`	Le nombre de tapes effectuées par l'utilisateur.
	`@property(nonatomic, readonly) NSTime Interval timestamp`	L'horodate de la dernière modification de la touche.
	`@property(nonatomic, readonly) UITouch Phase phase`	La phase dans laquelle se trouve la touche : `UITouchPhaseBegan` lorsque le doigt vient de toucher l'écran ; `UITouchPhaseMoved` lorsque le doigt vient de se déplacer ; `UITouchPhaseStationary` lorsque le doigt n'a pas bougé depuis le dernier événement ; `UITouchPhaseEnded` lorsque le doigt vient de se retirer de l'écran ; `UITouchPhaseCancelled` si l'événement a été interrompu.

Remarquez la propriété `phase` qui permet de déterminer la touche élémentaire représentée : pose, déplacement ou retrait du doigt. Cette propriété peut également indiquer :

- si le doigt est immobile sur l'écran ;
- si l'événement a été interrompu (*cancelled*) ; c'est le cas par exemple si l'iPhone reçoit un appel pendant l'utilisation d'une application.

Écran Multi-Touch

La technologie *Multi-Touch* permet au système de suivre les mouvements simultanés de plusieurs doigts sur l'écran. Chaque mouvement est décomposé en une série de touches élémentaires. Les touches élémentaires simultanées sont regroupées au sein d'un même événement, une instance de la classe `UIEvent`.

Il appartient au répondeur d'interpréter ces suites d'événements pour déterminer les **gestes** effectués par l'utilisateur. Par exemple, un *pincement* (*pinch*) est décomposé de la façon suivante :

- Deux doigts sont posés simultanément sur l'écran.
- La distance entre les deux doigts diminue.
- Les deux doigts sont retirés de l'écran.

Chacun de ces trois événements est composé de deux touches élémentaires : deux posés, deux déplacements et deux retraits.

Les classes dérivées de `UIView` ou de `UIControl` doivent implémenter le code nécessaire pour interpréter les gestes qui leur sont propres : glissement (*swipe*), pichenette (*flick*), pincement (*pinch*), etc. Nous allons examiner les méthodes à utiliser pour implémenter vos propres gestes dans vos classes dérivées.

9.2. Traiter les événements

Recevoir les événements

Classe UIEvent

Une instance de la classe `UIEvent` représente un événement ; c'est sous cette forme qu'il est transmis au répondeur. Elle peut contenir une ou plusieurs touches, sous la forme d'instances de `UITouch`, ou représenter une secousse de l'appareil. Les méthodes et propriétés de la classe `UIEvent` sont décrites dans le tableau ci-après.

Tableau 9.2 : Méthodes et propriétés de la classe UIEvent

Thème	Signature	Objet
Obtenir les touches	`- (NSSet *)allTouches`	Retourne toutes les touches de l'événement.
	`- (NSSet *)touches ForView:(UIView *)view`	Retourne les touches appartenant à une vue.
	`- (NSSet *)touches ForWindow:(UIWindow *)window`	Retourne les touches appartenant à une fenêtre.
Obtenir les attributs de l'événement	`property(nonatomic, readonly) NSTime Interval timestamp`	L'horodate de l'événement en secondes depuis le démarrage du système.
Type d'événement	`@property(readonly) UIEventType type`	Le type est soit `UIEventTypeTouches` pour un ensemble de touches, soit `UIEventTypeMotion` pour un mouvement de l'appareil.
	`@property(readonly) UIEventSubtype subtype`	Le sous-type est `UIEventSubtypeNone` si l'événement n'a pas de sous-type particulier ou `UIEventSubtypeMotionShake` pour une secousse de l'appareil.

Les touches de l'événement sont retournées dans une instance de la classe `NSSet`. Il s'agit d'un conteneur, au même titre que `NSArray` et

`NSDictionary` que nous connaissons déjà, qui représente un *ensemble*. Les éléments d'un ensemble ne sont pas rangés de façon particulière et son contenu est exploré avec l'instruction `for` :

```
NSSet * aSet = [NSSet setWithObjects:@"Jean",@"Marc",
                                          @"Paul",nil];
for ( NSString * name in aSet) {
   // name contiendra successivement Jean, Marc et Paul
}
```

Deux autres méthodes de la classe `NSSet` sont utiles pour traiter les événements :

- `-(id)anyObject` qui retourne un élément quelconque du conteneur ;
- `-(NSArray*)allObjects` qui retourne un tableau contenant tous les objets du conteneur.

Conditions de réception

Événements de touches

Pour des raisons de performance, les événements de touches ne sont transmis par l'application qu'aux vues qui satisfont certains critères :

- La vue doit être affichée à l'écran.

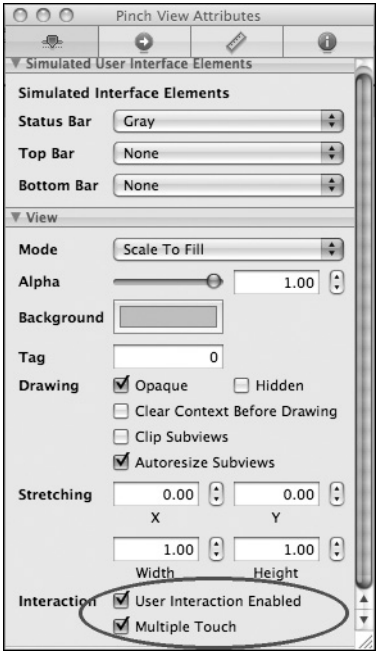


Figure 9.2 : Cases à cocher pour recevoir les événements de touches

- La vue doit contenir le point touché par l'utilisateur, sauf si sa propriété `exclusiveTouch` vaut `YES`, auquel cas la vue recevra tous les événements de la fenêtre.
- La propriété `userInteractionEnabled` de la vue doit valoir `YES` ou la case *User Interaction Enabled* doit être cochée dans l'inspecteur des attributs de la vue sous *Interface Builder*.
- Une vue ne peut recevoir qu'une touche à la fois, sauf si sa propriété `multipleTouchEnabled` vaut `YES` ou si la case *Multiple Touch* est cochée.

Événements de mouvements

Les événements de mouvements sont transmis par l'application au premier répondeur (*First Responder*).

Un répondeur devient le premier répondeur lorsque :

- Il reçoit le message `-becomeFirstResponder`.
- Uniquement si sa méthode `–canBecomeFirstResponder` retourne `YES`.

Par défaut, la méthode `–canBecomeFirstResponder` retourne `NO`. Il faut donc redéfinir cette méthode si l'on veut que nos propres classes dérivées de `UIResponder` puissent devenir des premiers répondeurs :

```
- (BOOL)canBecomeFirstResponder{
   return YES;
}
```

Pour qu'une vue puisse devenir premier répondeur, il faut également qu'elle soit affichée à l'écran. De la même façon, pour qu'un contrôleur de vue puisse devenir premier répondeur, sa vue doit être affichée à l'écran. Il ne faut donc pas appeler la méthode `–becomeFirstResponder` tant que cette condition n'est pas remplie ; la méthode `–viewDidAppear` de `UIViewController` constitue une bonne opportunité pour définir le premier répondeur :

```
- (void)viewDidAppear:(BOOL)animated {
   [self becomeFirstResponder];
}
```

À signaler aussi la possibilité de bloquer temporairement le traitement des événements par l'application :

```
UIApplication * appli = [UIApplication sharedApplication];
[appli beginIgnoringInteractionEvents];
// plus aucun événement n'est traité par l'application
[appli endIgnoringInteractionEvents];
// les événements sont de nouveau traités par l'application
```

Notification d'événements

Nous avons compris ce qu'est un événement, une touche, et quelles sont les conditions pour que les événements arrivent au répondeur. Intéressons-nous maintenant à la façon dont un répondeur reçoit les événements et à la façon dont il doit les traiter.

Mouvements

La secousse de l'appareil est le seul mouvement qui provoque un événement dans la version actuelle d'iPhone OS.

Secouer le Simulateur
Vous pouvez simuler une secousse sur le Simulateur d'iPhone avec la commande **Secousse** du menu **Matériel** (Ctrl+⌘+Z)

Lorsqu'un tel événement se produit, le premier répondeur reçoit les messages suivants :

- `-(void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent *)event` lorsqu'un mouvement débute ;
- `-(void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event` lorsque le mouvement se termine ;
- `-(void)motionCancelled:(UIEventSubtype)motion withEvent:(UIEvent *)event` lorsque le mouvement est interrompu.

En pratique, les paramètres `motion` et `event` ne sont pas utilisés, puisqu'un seul type d'événement peut survenir, et le code qui permet à l'application de réagir à l'action de l'utilisateur est placé dans la méthode `-motionEnded:withEvent:` :

```
- (void)motionBegan:(UIEventSubtype)motion
                        withEvent:(UIEvent *)event {
}
- (void)motionEnded:(UIEventSubtype)motion
                        withEvent:(UIEvent *)event {
   // Insérer ici le code pour traiter l'événement
}
- (void)motionCancelled:(UIEventSubtype)motion
                        withEvent:(UIEvent *)event {
}
```

Définir toutes les méthodes

Si votre répondeur dérive de `UIView` ou `UIViewController`, ce qui est le cas le plus courant, les trois méthodes précédemment décrites doivent être redéfinies même si certaines d'entre elles ne contiennent pas de code.

Challenge

Complétez l'application *Billard* du chapitre précédent : la vitesse de la boule doit être réinitialisée lorsque l'utilisateur secoue l'appareil.

Indications :

- Implémentez les méthodes de traitement des événements dans le contrôleur de vue `BillardViewController`.
- N'oubliez pas de définir ce contrôleur de vue comme premier répondeur.

Touches

Les événements de touches élémentaires sont transmis aux répondeurs par les messages suivants :

- `-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event` pour les touches qui débutent (qui correspondent à une pose de doigt) ;
- `-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event` pour les touches qui correspondent à un déplacement ;
- `-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event` pour les touches qui se terminent (retrait du doigt) ;
- `-(void)touchesCancelled:(NSSet*)touches withEvent:(UIEvent*)event` pour les touches interrompues.

Le paramètre `touches` de chacune de ces méthodes contient les touches élémentaires, instances de la classe `UITouch`, qui sont respectivement dans les états `UITouchPhaseBegan`, `UITouchPhaseMoved`, `UITouchPhaseEnded` et `UITouchPhaseCancelled`. Le paramètre `event` regroupe toutes les touches élémentaires de l'événement, quel que soit leur état.

Nous mettrons en œuvre ces méthodes dans la section suivante.

Libérer les ressources

Les ressources éventuellement allouées pour gérer un geste doivent être libérées lorsque le geste se termine, c'est-à-dire dans la méthode

–touchesEnded:withEvent: s'il s'agit de l'événement qui clôt le geste, et dans la méthode –touchesCancelled:withEvent: lorsque le geste est interompu.

Définir toutes les méthodes

Comme dans le cas des événements de mouvement, si votre répondeur dérive de UIView ou UIViewController, les quatre méthodes précédemment décrites doivent être redéfinies même si certaines d'entre elles ne contiennent pas de code.

Tapes multiples

La classe UITouch définit une propriété tapCount ; un entier contenant le nombre de tapes effectuées au même endroit. Pour savoir si une touche est une tape multiple, il suffit de tester cette propriété dans la méthode –touchesEnded:withEvent:.

Il y a un petit détail auquel il faut faire attention si nous souhaitons obtenir un comportement différent pour chaque tape. Notre répondeur va recevoir une première série d'événements à la première tape, une autre série à la deuxième, etc. Lorsque la première tape est reçue, nous devons attendre une fraction de secondes avant de déclencher l'action attendue afin de déterminer s'il s'agit d'une tape simple, double, etc.

Le plus simple pour arriver à ce résultat est de lancer l'action déclenchée par une tape simple avec la méthode –performSelector:withObject:afterDelay: :

```
- (void)touchesEnded:(NSSet *)touches
                          withEvent:(UIEvent *)event {
   UITouch *theTouch = [touches anyObject];
   if (theTouch.tapCount == 1) {
   [self performSelector:@selector(handleSingleTap)
                                        withObject:nil
                                        afterDelay:0.3];
   } else if (theTouch.tapCount == 2) {
      // Instructions pour traiter une tape double
   }
}
- (void) handleSingleTap {
   // Instructions pour traiter une tape unique
}
```

Ceci nous laissera l'opportunité d'annuler cette action s'il s'avère que la première tape était le début d'une tape double :

```
- (void)touchesBegan:(NSSet *)touches
                                withEvent:(UIEvent *)event {
   UITouch *aTouch = [touches anyObject];
   if (aTouch.tapCount == 2) {
      [NSObject
            cancelPreviousPerformRequestsWithTarget:self];
   }
}
```

On peut utiliser le même mécanisme pour discriminer les tapes triples, quadruples, etc. La multiplicité des tapes n'est pas limitée par *Cocoa Touch*.

9.3. Mettre en œuvre les gestes

Il est temps de mettre en œuvre les éléments que nous venons de voir en réalisant des vues qui réagissent aux gestes de l'utilisateur.

Nous allons enrichir notre application *Billard* en donnant la possibilité à l'utilisateur de propulser la boule par une *chiquenaude*. Nous illustrerons ensuite les touches multiples par la mise en œuvre du *pincement*.

Chiquenaude

Comportement souhaité

Physique de la chiquenaude

La chiquenaude est un déplacement d'un doigt sur l'écran qui doit :

- être *rapide* ; s'il est trop lent, il ne doit pas être pris en compte ;
- *percuter* la boule ; le mouvement doit passer à proximité du centre de la boule, il faut prendre en compte l'imprécision due à la taille du doigt.

Nous allons émettre des hypothèses simplificatrices quant à l'effet de la chiquenaude sur la boule :

- La boule est propulsée à la vitesse de la chiquenaude ; la vitesse de déplacement du doigt sur l'écran.
- La boule est propulsée dans la direction de la chiquenaude ; il n'y a pas d'effet de rotation de la boule due à une percussion qui ne serait pas radiale.

Mathématique de la chiquenaude

Une chiquenaude se traduira par une série d'événements de déplacement d'une touche élémentaire. Chaque déplacement peut être considéré séparément, nous n'avons besoin de conserver que l'horodate (*timestamp*) de la dernière touche élémentaire pour calculer la vitesse de déplacement lors de l'événement suivant.

Nous utiliserons les méthodes `–previousLocationInView:` et `–locationInView:` de `UITouch` pour déterminer les caractéristiques de la chiquenaude :

- Le déplacement rencontre-t-il la boule ?
- Quel est le vecteur vitesse à donner à la boule ?

Répondre à la première question requiert un niveau de mathématique élémentaire. Si ce n'est pas votre cas, nous vous demandons de nous faire confiance. Si le dernier déplacement élémentaire va du *point 1* au *point 2*, nous considérons les deux vecteurs :

- celui qui va du point 1 au point 2 ;
- celui qui va du point 1 à la position de la boule.

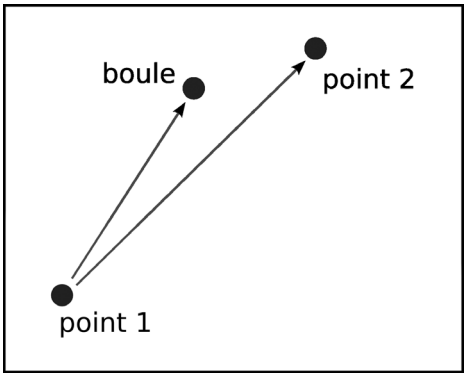


Figure 9.3 : Le déplacement rencontre-t-il la boule ?

On considérera que la boule est percutée si :

- le premier vecteur est plus long que le second, au diamètre de la boule près ;
- l'angle entre les deux vecteurs est suffisamment petit. Cet angle sera indirectement évalué à l'aide du déterminant des deux vecteurs.

Classe SnookerView

Le travail de cette classe est de détecter la chiquenaude et d'en transmettre les paramètres au contrôleur de vue :

- un rectangle défini par les deux points du déplacement ;
- la durée du déplacement du doigt entre ces deux points.

Pour transmettre ces informations, nous allons définir un protocole de délégué spécifique auquel le contrôleur de vue devra se conformer.

1 Ouvrez le projet *Billard* sous XCode et modifiez le fichier *SnookerView.h*, nous en profitons pour définir une variable d'instance `lastTime` qui nous servira à calculer la durée du déplacement :

```
@protocol SnookerViewDelegate;
@interface SnookerView : UIView {
   BOOL drawing;
   CGPoint precedingLastPoint;
   CGPoint lastPoint;
   NSTimeInterval lastTime;
   IBOutlet id <SnookerViewDelegate> delegate;
}
@property(nonatomic,getter=isDrawing) BOOL drawing;
@property(nonatomic) CGPoint lastPoint;
@property(nonatomic,assign)
                        id <SnookerViewDelegate> delegate;
@end

@protocol SnookerViewDelegate <NSObject>
@optional
- (void)swipeMove:(CGRect)move
             withDuration:(NSTimeInterval)swipeDuration;
@end
```

2 Modifiez le début du fichier *SnookerView.m* pour synthétiser les accesseurs de la propriété `delegate` et la vitesse minimale d'une chiquenaude :

```
const float minSpeed = 300.;
@implementation SnookerView
@synthesize drawing,lastPoint,delegate;
```

3 Ajoutez les méthodes permettant de traiter les événements de touches :

```
- (void)touchesBegan:(NSSet*)touches
                                withEvent:(UIEvent*)event{
   lastTime = [[touches anyObject] timestamp];
}
- (void)touchesMoved:(NSSet*)touches
```

```
                                withEvent:(UIEvent*)event{
    UITouch * touch = [touches anyObject];
    NSTimeInterval currentTime = [touch timestamp];
    NSTimeInterval swipeDuration = currentTime-lastTime;
    lastTime = currentTime;
    CGPoint point1 = [touch previousLocationInView:self];
    CGPoint point2 = [touch locationInView:self];
    CGRect move = CGRectMake(point1.x, point1.y,
                    point2.x-point1.x, point2.y-point1.y);
   CGFloat speed = sqrt(move.size.width*move.size.width +
         move.size.height*move.size.height)/swipeDuration;
    if (speed > minSpeed) {
       if ([self.delegate respondsToSelector:
                    @selector(swipeMove:withDuration:)]) {
          [self.delegate swipeMove:move
                              withDuration:swipeDuration];
       }
    }
}
- (void)touchesEnded:(NSSet*)touches
                                withEvent:(UIEvent*)event{
}
- (void)touchesCancelled:(NSSet*)touches
                                withEvent:(UIEvent*)event{
}
```

Classe BillardViewController

Le travail du contrôleur est de vérifier si la chiquenaude doit avoir un effet sur la boule, et le cas échéant de réaliser cet effet. Tout cela sera effectué dans la méthode `–swipeMove:withDuration:` définie dans le protocole `SnookerViewDelegate`.

1 Ajoutez cette méthode dans le fichier *BillardViewController.m* :

```
- (void)swipeMove:(CGRect)move
             withDuration:(NSTimeInterval)swipeDuration {
    // Détermination de la proximité de la boule
       // longueur de la chiquenaude
    CGFloat moveLength =
                   sqrt(move.size.width*move.size.width+
                       move.size.height*move.size.height);
       // Vecteur Origine-Boule et longueur
    CGSize ballVector =
                   CGSizeMake(ball.center.x-move.origin.x,
                              ball.center.y-move.origin.y);
    CGFloat ballLength =
                 sqrt(ballVector.width*ballVector.width+
                    ballVector.height*ballVector.height);
       // calcul du déterminant
    CGFloat det = (move.size.width*ballVector.height-
  move.size.height*ballVector.width)/(moveLength*ballLength);
    // Modification de la vitesse de la boule
```

```
    if ((ballLength<moveLength+distancePrecision)&&
                                (fabs(det)<anglePrecision)){
     moveX = move.size.width*timerInterval/swipeDuration;
     moveY = move.size.height*timerInterval/swipeDuration;
       // effacement de la table
       [(SnookerView*)self.view setDrawing:NO];
       [self.view setNeedsDisplay];
       // premier déplacement
      [(SnookerView*)self.view setLastPoint: ball.center];
       [(SnookerView*)self.view setDrawing:YES];
       [self moveBall];
    }
}
```

2 Déclarez les constantes nécessaires en tête du fichier :

```
const float distancePrecision = 15.;
const float anglePrecision = 0.1;
```

3 N'oubliez pas de déclarer que la classe `BillardViewController` adopte le protocole `SnookerViewDelegate` dans son fichier d'interface puis ouvrez le fichier *BillardViewController.xib* sous Interface Builder pour attacher le délégué de la vue `SnookerView` au propriétaire du fichier.

4 Cochez la case *Clear Context Before Drawing* dans l'inspecteur des attributs pour la vue `SnookerView` dans le fichier *NIB*.

Vous pouvez construire l'application et la tester.

Challenge

Vous aurez certainement remarqué que l'on peut devenir très violent avec la boule, et lui donner une vitesse faramineuse au point qu'elle peut sortir de l'écran. Limitez la vitesse de la boule ou améliorez le code pour que la boule ne sorte jamais de l'écran.

Pincement

Le deuxième exemple de geste que nous allons développer est le *pincement* tel qu'il est utilisé dans Safari pour iPhone et qui a participé à la popularité de l'iPhone.

Un pincement est un geste dans lequel deux doigts sont posés sur l'écran et s'écartent ou se rapprochent. Nous allons créer une vue qui détecte ce geste en vérifiant, lors d'un déplacement d'une touche élémentaire, que deux touches sont en cours d'utilisation.

Si c'est le cas, le ratio *(distance actuelle entre les deux touches)/(distance précédente entre les deux touches)* sera transmis au délégué de la vue.

Classe PinchView

1 Créez un nouveau projet sous XCode de type **View Based Application**. Intitulez-le *Pinch*. Créez une nouvelle classe `PinchView` qui dérive de `UIView`. Ajoutez la définition du protocole de délégué de cette classe dans son fichier d'interface :

```
#import <UIKit/UIKit.h>
@protocol PinchViewDelegate;

@interface PinchView : UIView {
   id <PinchViewDelegate> delegate;
}
@property(nonatomic,retain) id delegate;
@end

@protocol PinchViewDelegate
@required
- (void) pinchPerformed:(float) ratio;
@end
```

2 Ajoutez les méthodes de traitement des événements de touche dans le fichier *PinchView.m* :

```
- (void)touchesBegan:(NSSet *)touches
                            withEvent:(UIEvent *)event {
}
- (void)touchesMoved:(NSSet *)touches
                            withEvent:(UIEvent *)event {
   if ([[event touchesForView:self] count]==2) {
      // 2 doigts sont posés sur l'écran
     NSArray *t=[[event touchesForView:self] allObjects];
      // t1 et t2 sont les deux touches sur l'écran
      UITouch *t1 = [t objectAtIndex:0];
      UITouch *t2 = [t objectAtIndex:1];
      // calcul de la distance précédente
      CGPoint p1 = [t1 previousLocationInView:self];
      CGPoint p2 = [t2 previousLocationInView:self];
      CGFloat previousDistance =
   sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
      // calcul de la distance actuelle
      p1 = [t1 locationInView:self];
      p2 = [t2 locationInView:self];
      CGFloat currentDistance =
   sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
      // transmission du ratio au délégué
      [delegate pinchPerformed:
                        currentDistance/previousDistance];
```

```
    }
}
- (void)touchesEnded:(NSSet *)touches
                              withEvent:(UIEvent *)event {
}
- (void)touchesCancelled:(NSSet *)touches
                              withEvent:(UIEvent *)event {
}
```

On dénombre toutes les touches présentes sur l'écran (`[event touchesForView:self]`) et pas seulement celles qui viennent de se déplacer (`touches`).

Classe PinchViewController

1. Modifiez l'interface du contrôleur de vue pour qu'il adopte le protocole de délégué de PinchView. Nous y ajoutons un champ de texte qui nous permettra de visualiser l'effet des pincements :

```
#import <UIKit/UIKit.h>
#import "PinchView.h"
@interface PinchViewController : UIViewController
                                        <PinchViewDelegate>{
   IBOutlet UILabel *label;
}
@property(nonatomic,retain) UILabel *label;
@end
```

2. Dans le fichier *PinchViewController.m*, synthétisez les accesseurs pour la propriété label et ajoutez la méthode `–pinchPerformed:` :

```
- (void)pinchPerformed:(float)ratio{
   float previousValue = [label.text floatValue];
   label.text = [NSString stringWithFormat:@"%f",
                                        previousValue*ratio];
}
```

Finaliser l'application

Pour finaliser l'application :

1. Ouvrez le fichier *PinchViewController.xib* sous Interface Builder.
2. Définissez la classe de la vue principale ; `PinchView`.
3. Autorisez les touches multiples dans la vue principale.
4. Définissez le propriétaire du fichier NIB comme délégué de la vue principale.
5. Ajoutez un *label* sur la vue principale. Initialisez-le à `100`.
6. Liez ce *label* à l'outlet `label` du propriétaire du fichier.

Vous pouvez maintenant construire et tester l'application.

Pincement sur le simulateur

Le *pincement* est le seul geste à plusieurs doigts réalisable sur le simulateur d'iPhone. Pressez la touche ⌥ sur le clavier en manipulant la souris.

9.4. Checklist

Ce chapitre nous a permis de connaître les différents types d'événements élémentaires de *Cocoa Touch* :

- *secousse* de l'appareil ;
- *touches* élémentaires et leurs différents stades permettant d'interpréter les gestes effectués par l'utilisateur :
  - Un doigt se pose sur l'écran.
  - Un doigt se déplace.
  - Un doigt est retiré de l'écran.

Nous avons ensuite compris comment sont représentés ces événements par des instances des classes `UIEvent` et `UITouch`, puis comment ils sont reçus et traités par les répondeurs de la classe `UIResponder`.

Ces principes ont été illustrés par l'implémentation de deux gestes courant, la *chiquenaude* et le *pincement*.

APPAREIL PHOTO

Nous allons améliorer notre application *Emprunts2* qui deviendra *Emprunts3*. Notre objectif maintenant est de conserver une preuve du prêt ; nous allons créer une fonctionnalité permettant d'ajouter une photo à chaque enregistrement.

Certains exemples fournis dans ce chapitre fonctionnent uniquement sur iPhone. Il n'y a actuellement pas d'appareil photo sur iPod Touch ni sur iPad.

10.1. Sélectionner une photo

Nous allons commencer par découvrir la classe `UIImagePickerController` et son utilisation dans notre application *Emprunts* pour sélectionner une photo parmi les albums de l'application *Photos*.

Figure 10.1 : Interface pour choisir une photo

Codage de l'interface

Sous XCode, créez un projet *Emprunts3* à partir du projet *Emprunts2*.

Interface de la classe

1 Sous XCode modifiez le fichier *LendObjectViewController.h* pour y ajouter :

— un outlet `imageView` de classe `UIImageView` afin de visualiser une miniature de la photo choisie ;

— une variable d'instance `picture` de classe `UIImage` qui contiendra la photo choisie ;

— un outlet `pictureButton` de classe `UIButton` pour régler le comportement du bouton ;

— une action `–takePicture` qui sera activée lorsque l'utilisateur souhaitera choisir une image :

```
@interface LendObjectViewController : UIViewController
                        <UINavigationControllerDelegate,
                        UIImagePickerControllerDelegate>{
    NSManagedObject * lendObject;
    IBOutlet UITextField * objectNameField;
    IBOutlet UITextField * borrowerNameField;
    IBOutlet UIImageView * imageView;
    IBOutlet UIButton * pictureButton;
    IBOutlet UIDatePicker * datePicker;
    UIImage * picture;
}
@property(nonatomic,retain) NSManagedObject * lendObject;
@property(nonatomic,retain) UITextField * objectNameField;
@property(nonatomic,retain) UITextField * borrowerNameField;
@property(nonatomic,retain) UIImageView * imageView;
@property(nonatomic,retain) UIButton * pictureButton;
@property(nonatomic,retain) UIDatePicker * datePicker;
@property(nonatomic,retain) UIImage * picture;
- (IBAction) doneEditing:(id)sender;
- (IBAction) deleteObject;
- (IBAction) takePicture;
@end
```

Nous déclarons aussi que notre classe `LendObjectViewController` adopte les protocoles `UINavigationControllerDelegate` et `UIImagePickerControllerDelegate`.

2 Enregistrez le fichier *LendObjectViewController.h*.

Interface utilisateur

1 Ouvrez le fichier *LendObjectViewController.xib* sous Interface Builder pour y ajouter une vue image et un bouton.

2 Liez la vue image à l'outlet `imageView`, le bouton à l'outlet `pictureButton` et *l'événement Touch Up Inside* du bouton à l'action `takePicture` du propriétaire du fichier.

Codage du contrôleur de vue

Le contrôleur de vue `LendObjectViewController` doit réaliser plusieurs tâches :

- vérifier que des albums photos sont disponibles sur l'appareil ;
- lancer le sélectionneur de photo lorsque l'utilisateur a touché le bouton adéquat ;
- prendre en compte la photo choisie ;
- afficher une miniature de la photo choisie.

Vérifier que les albums photos sont disponibles

Sous XCode, ouvrez le fichier *LendObjectViewController.m* et modifiez la méthode `–viewDidLoad` :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   UIBarButtonItem *cancelButton = [[UIBarButtonItem alloc]
                  initWithTitle:@"Delete"
                          style:UIBarButtonItemStyleDone
                         target:self
                         action:@selector(deleteObject)];
   self.navigationItem.rightBarButtonItem = cancelButton;
   [cancelButton release];
   if ([UIImagePickerController isSourceTypeAvailable:
          UIImagePickerControllerSourceTypePhotoLibrary]){
      [pictureButton setTitle:@"Choisir une photo"
                          forState:UIControlStateNormal];
   } else {
      pictureButton.enabled = NO;
   }
}
```

Le chargement de la vue est le bon endroit pour définir le titre du bouton ou le désactiver si l'album photo n'est pas disponible sur l'appareil.

Nous utilisons la classe `UIImagePickerController` qui permet de gérer toutes les sources de photos de l'appareil. Sa méthode `+isSourceTypeAvailable:` renvoie `YES` si la source dont l'identifiant est passé en paramètre est disponible sur l'appareil. Les types de source existants sont :

- `UIImagePickerControllerSourceTypePhotoLibrary` pour accéder aux albums de la bibliothèque de l'application *Photos* ;
- `UIImagePickerControllerSourceTypeCamera` pour accéder à la caméra vidéo ou à l'appareil photo ;

- `UIImagePickerControllerSourceTypeSavedPhotosAlbum` pour accéder aux vidéos ou photos enregistrées depuis la caméra ou l'appareil photo, ou par défaut aux albums de la bibliothèque de l'application *Photos*.

Lancer le sélectionneur de photos

Ajoutez le code de l'action `–takePicture` :

```
- (void) takePicture {
   UIImagePickerController *picker =
                    [[UIImagePickerController alloc] init];
   picker.sourceType =
            UIImagePickerControllerSourceTypePhotoLibrary;
   picker.delegate = self;
   [self presentModalViewController:picker animated:YES];
   [picker release];
}
```

Nous créons un contrôleur spécifique pour la sélection d'une photo, une instance de la classe `UIImagePickerController` puis nous lui indiquons que nous souhaitons explorer les albums photo. Enfin, nous activons le contrôleur.

Prendre en compte le choix d'image

Nous allons maintenant coder les deux méthodes du protocole `UIImagePickerControllerDelegate`, l'une qui est appelée lorsque l'utilisateur a sélectionné une image, et l'autre lorsqu'il annule l'opération :

```
- (void)imagePickerController:(UIImagePickerController *)
 picker didFinishPickingMediaWithInfo:(NSDictionary *)info{
   self.picture = [info
        objectForKey:UIImagePickerControllerOriginalImage];
   [self dismissModalViewControllerAnimated:YES];
}
- (void)imagePickerControllerDidCancel:
                        (UIImagePickerController *)picker {
   [self dismissModalViewControllerAnimated:YES];
}
```

Le paramètre `info` de la méthode `–imagePickerController:didFinishPickingMediaWithInfo:` est un dictionnaire qui contient les informations relatives au média sélectionné (photo ou vidéo). La clé `UIImagePickerControllerOriginalImage` permet de récupérer la photo sélectionnée.

Comme pour les vues modales standard, il est de la responsabilité du délégué de désactiver la vue modale en appelant la méthode `–dismissModalViewControllerAnimated:`.

Afficher la photo

Pour afficher la photo sélectionnée, il suffit de compléter la méthode `–viewWillAppear:` qui est appelée juste avant que la vue soit affichée :

```
- (void)viewWillAppear:(BOOL)animated{
   if ( [self.lendObject valueForKey: @"lendDate"] ) {
      self.objectNameField.text =
            [self.lendObject valueForKey: @"objectName"] ;
      self.borrowerNameField.text =
           [self.lendObject valueForKey: @"borrowerName"] ;
      self.datePicker.date =
              [self.lendObject valueForKey: @"lendDate"] ;
   }
   self.imageView.image = self.picture;
   [super viewWillAppear:animated];
}
```

Finaliser l'application

1 Synthétisez les accesseurs pour les trois propriétés que nous venons de déclarer : `imageView`, `pictureButton` et `picture` :

```
@synthesize imageView, pictureButton, picture;
```

2 Libérez les propriétés retenues :

```
- (void)viewDidUnload {
   self.objectNameField = nil;
   self.borrowerNameField = nil;
   self.datePicker = nil;
   self.imageView = nil;
   self.pictureButton = nil;
}
- (void)dealloc {
   [self viewDidUnload];
   self.lendObject = nil;
   self.picture = nil;
   [super dealloc];
}
```

Libération des propriétés

Nous libérons les propriétés. Celles qui sont définies dans le fichier *NIB*, les outlets, et celles qui sont allouées dans la méthode `–viewDidLoad` doivent être libérées dans la méthode `–viewDidUnload`. Les autres sont libérées dans la méthode `–dealloc`.

Le sélectionneur d'image étant susceptible d'utiliser beaucoup de ressources, il n'est pas impossible que la méthode `–viewDidUnload` soit appelée sur l'instance de `LendObjectViewController` pendant que

l'utilisateur choisit une photo. Il est donc important de libérer tous les outlets car ils seront tous recréés lorsque ce contrôleur reprendra la main sur l'interface utilisateur. Il est important aussi de ne pas libérer les références vers le modèle, en l'occurrence les propriétés `lendObject` et `picture`, car le contrôleur ne saura pas les recréer seul.

3 Construisez l'application et testez-la. À ce stade, le test peut être effectué sur le simulateur.

C'est bien de pouvoir choisir une photo. Ce serait encore mieux si ce choix pouvait être conservé avec la liste des objets prêtés. Avant de nous occuper de cela, nous allons faire plus ample connaissance avec la classe `UIImagePickerController` et son protocole de délégué.

Classe UIImagePickerController

Le tableau résume les méthodes et propriétés de la classe `UIImagePickerController`.

Tableau 10.1 : Méthodes et propriétés de la classe UIImagePickerController

Thème	Signature	Objet
Sources des images	`+ (NSArray *) availableMediaTypesForSourceType: (UIImagePickerControllerSourceType)sourceType`	Retourne un tableau contenant la liste des types de médias disponible dans le type de source passé en paramètre. En particulier `kUTTypeMovie` si l'appareil est capable d'enregistrer de la vidéo.
	`+ (BOOL) isSourceTypeAvailable: (UIImagePickerControllerSourceType)sourceType`	Retourne `YES` si la source est disponible sur l'appareil.
	`@property(nonatomic) UIImagePickerControllerSourceType sourceType`	Source utilisée pour la sélection d'images. Doit être initialisé avant d'activer le contrôleur.
Configurer le sélectionneur	`@property(nonatomic) BOOL allowsEditing`	Doit être initialisé à `YES` pour autoriser l'édition par l'utilisateur de l'image ou de la vidéo sélectionnée. Vaut `NO` par défaut.
	`@property(nonatomic, assign) id <UINavigationControllerDelegate, UIImagePickerControllerDelegate> delegate`	Délégué.
	`@property(nonatomic,copy) NSArray *mediaTypes`	Tableau contenant les médias dont l'accès est autorisé. `kUTTypeImage` par défaut

Tableau 10.1 : Méthodes et propriétés de la classe UIImagePickerController

Thème	Signature	Objet
Configuration de la prise de vidéo	`@property(nonatomic) UIImagePickerControllerQualityType videoQuality`	Niveau de qualité sélectionnée. `Médium` par défaut.
	`@property(nonatomic) NSTimeInterval videoMaximumDuration`	Durée maximale de la capture vidéo. La valeur par défaut est de `10 minutes`, ce qui est la valeur maximale admissible.
Commandes de la prise de vue	`@property(nonatomic)BOOL showsCameraControls`	`YES` pour que les commandes par défaut soient affichées.
	`@property(nonatomic, retain) UIView *cameraOverlayView`	Vue contenant des commandes personnalisées. `nil` par défaut.
	`@property(nonatomic) CGAffineTransform cameraViewTransform`	Transformation à appliquer sur l'image pendant la prise de vue.
	`– (void) takePicture`	Utilisé dans un contrôle personnalisé pour prendre une photo.

On voit que cette classe peut être utilisée aussi bien pour mettre en œuvre la caméra de l'appareil, prendre une photo ou capturer une vidéo. Nous ajouterons cette fonctionnalité à l'application *Emprunts3*.

Cette classe permet également au développeur d'ajouter ses propres contrôles pendant la prise de vue et d'activer le module élémentaire d'édition d'image.

La propriété `videoQuality` est du type énuméré `UIImagePickerControllerQualityType` qui peut prendre l'une des trois valeurs suivantes :

- `UIImagePickerControllerQualityTypeHigh` pour une qualité haute ;
- `UIImagePickerControllerQualityTypeMedium` pour une qualité moyenne ;
- ou `UIImagePickerControllerQualityTypeLow` pour une qualité médiocre.

Type de média

Seule la caméra de l'iPhone 3GS peut capturer de la vidéo, l'iPod Touch ne dispose pas de caméra et à la date où nous écrivons ces lignes, l'iPad n'en dispose pas non plus.

Utilisez les méthodes `–isSourceTypeAvailable:` et `–availableMediaTypesForSourceType:` pour connaître les caractéristiques de l'appareil sur lequel votre application s'exécute.

Protocole UIImagePickerControllerDelegate

Le tableau résume les deux méthodes définies dans le protocole `UIImagePickerControllerDelegate`, nous les avons déjà utilisé toutes les deux.

Tableau 10.2 : Méthodes du protocole UIImagePickerControllerDelegate

Méthode	Objet
`– (void) imagePickerController: (UIImagePickerController *)picker didFinishPickingMediaWithInfo: (NSDictionary *)info`	Le sélectionneur passé en paramètre vient de sélectionner une image. Les informations sont dans le dictionnaire passé en second paramètre.
`– (void) imagePickerControllerDidCancel:(UIImagePickerController *)picker`	Le sélectionneur passé en paramètre vient d'être annulé par l'utilisateur.

Le dictionnaire info reçu par la méthode `–imagePickerController: didFinishPickingMediaWithInfo:` est susceptible de contenir les informations suivantes :

- le type de média capturé, sous la clé `UIImagePickerControllerMediaType` ; `kUTTypeImage` pour une image et `kUTTypeMovie` pour une vidéo ;
- l'image originale, sous la clé `UIImagePickerControllerOriginalImage`, de type `UIImage` ;
- l'image éventuellement modifiée par l'utilisateur lors de la prise de vue, sous la clé `UIImagePickerControllerEditedImage` ; de type `UIImage` ;
- le rectangle délimitant la partie de l'image sélectionnée par l'utilisateur, sous la clé `UIImagePickerControllerCropRect` ; de type `CGRect` ;
- l'adresse URL de la vidéo capturée, sous la clé `UIImagePickerControllerMediaURL` ; de type `NSURL`.

10.2. Prendre des photos

Ce serait sans doute plus pratique pour l'utilisateur, s'il pouvait prendre une photo directement depuis l'application *Emprunts3*. La classe `UIImagePickerController` le permet, mais comme certains appareils (**iPod Touch** et **iPad**) ne sont pas dotés d'un appareil photo, il faut que l'interface utilisateur s'adapte à la situation.

Adapter l'interface utilisateur

Modifiez la méthode `–viewDidLoad` de la classe `LendObjectViewController` pour y adapter le titre du bouton en fonction des capacités de l'appareil :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   UIBarButtonItem *cancelButton = [[UIBarButtonItem alloc]
                  initWithTitle:@"Delete"
                          style:UIBarButtonItemStyleDone
                         target:self
                         action:@selector(deleteObject)];
   self.navigationItem.rightBarButtonItem = cancelButton;
   [cancelButton release];
   if ([UIImagePickerController isSourceTypeAvailable:
                UIImagePickerControllerSourceTypeCamera]){
      [pictureButton setTitle:@"Prendre une photo"
                         forState:UIControlStateNormal];
   } else if ([UIImagePickerController
          isSourceTypeAvailable:
          UIImagePickerControllerSourceTypePhotoLibrary]){
      [pictureButton setTitle:@"Choisir une photo"
                         forState:UIControlStateNormal];
   } else {
      pictureButton.enabled = NO;
   }
}
```

Adapter le sélectionneur de photos

1. Modifiez la méthode `–takePicture` de la classe `LendObjectViewController` pour indiquer au sélectionneur de photo quelle source utiliser en fonction des capacités de l'appareil :

```
- (void) takePicture {
   UIImagePickerController *picker =
                  [[UIImagePickerController alloc] init];
   if ([UIImagePickerController isSourceTypeAvailable:
                UIImagePickerControllerSourceTypeCamera]){
      picker.sourceType =
                  UIImagePickerControllerSourceTypeCamera;
```

```
    } else {
       picker.sourceType =
            UIImagePickerControllerSourceTypePhotoLibrary;
    }
    picker.delegate = self;
    [self presentModalViewController:picker animated:YES];
    [picker release];
}
```

2 Reconstruisez et testez l'application. Testez-la successivement avec le simulateur et sur un **iPhone** ou sur un **iPod Touch** pour vérifier qu'elle s'adapte aux capacités de l'appareil.

10.3. Enregistrer ses photos

Pour que l'application *Emprunts3* soit utile, il faut que la photo prise par l'utilisateur soit rangée avec les données de l'objet prêté, dans la structure **Core Data**. Nous allons réaliser cette fonction.

Gérer une image sous Core Data

Déclarer une image dans le modèle de données

Éditez le fichier *Emprunts3.xcdatamodel* sous XCode. Ajoutez une propriété `image`, de type `Binary data`, dans l'entité `LendObject`.

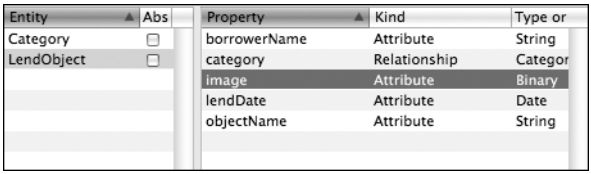


Figure 10.2 : Propriété image de LendObject

Outre les types d'attribut prédéfinis que nous connaissons déjà (nombre, chaîne de caractères, booléen, et date), le type *Binary Data* permet de stocker tout autre type de données dans une structure *Core Data*, en particulier une image de type `UIImage` comme nous le verrons bientôt.

Le type d'attribut *Binary Data* est équivalent à la classe Objective-C `NSData` qui encapsule un tableau d'octets. Et comme toute donnée informatique n'est au final qu'un tableau d'octets, on peut utiliser cette classe et ce type d'attribut pour stocker n'importe quel type de données.

Classe NSData

Le tableau ci-après résume les principales méthodes de la classe `NSData`.

Tableau 10.3 : Principales méthodes de la classe NSData

Thème	Signature	Objet
Création d'instances	`+ (id) dataWithBytes: (const void *)bytes length:(NSUInteger)length`	Crée une instance `NSData` à partir d'un tableau d'octets.
	`+ (id) dataWithContents OfFile:(NSString *)path`	Crée une instance `NSData` à partir du contenu d'un fichier.
	`+ (id) dataWithContents OfURL:(NSURL *)aURL`	Crée une instance `NSData` à partir du contenu d'une URL.
Accéder aux données	`– (const void *) bytes`	Renvoie un pointeur sur le tableau d'octets encapsulé dans le `NSData`.
	`– (NSUInteger) length`	Renvoie la longueur en octets du tableau encapsulé dans le `NSData`.
Enregistrer les données	`– (BOOL) writeToFile: (NSString *)path atomically:(BOOL)flag`	Crée un fichier avec le contenu de l'instance `NSData`. Si `flag` vaut `YES`, le fichier est créé uniquement si son intégrité peut être garantie.
	`– (BOOL) writeToURL: (NSURL *) aURL atomically:(BOOL) atomically`	Crée une URL avec le contenu de l'instance `NSData`. Si `flag` vaut `YES`, l'URL est créée uniquement si son intégrité peut être garantie.

Ces méthodes permettent la conversion d'une instance `NSData` de et vers :

- un tableau d'octets en mémoire ;
- le contenu d'un fichier ;
- un contenu adressé par une *URL*.

Le framework *Cocoa Touch* contient également plusieurs fonctions utilitaires qui permettent de convertir des données particulières en instance `NSData`. Notamment les images, ce que nous allons voir à la section suivante.

Transformer l'image en data

La fonction `UIImagePNGRepresentation` prend une instance `UIImage` en paramètre et retourne une instance `NSData` contenant l'image au format *PNG*. La méthode `+imageWithData:` de la classe `UIImage` réalise l'opération inverse.

1 Modifiez les méthodes `-viewWillAppear:` et `-imagePickerController:didFinishPickingMediaWithInfo:` de la classe `LendObjectViewController` pour enregistrer la photo sélectionnée dans la structure *Core Data* et la récupérer :

```
- (void)viewWillAppear:(BOOL)animated{
   if ( [self.lendObject valueForKey: @"lendDate"] ) {
      self.objectNameField.text =
           [self.lendObject valueForKey: @"objectName"] ;
      self.borrowerNameField.text =
          [self.lendObject valueForKey: @"borrowerName"] ;
      self.datePicker.date =
               [self.lendObject valueForKey: @"lendDate"]
      self.imageView.image = [UIImage
 imageWithData:[self.lendObject valueForKey: @"image"]] ;
   }
   [super viewWillAppear:animated];
}
- (void)imagePickerController:(UIImagePickerController *)
 picker didFinishPickingMediaWithInfo:(NSDictionary *)info{
   UIImage *picture = [info
        objectForKey:UIImagePickerControllerOriginalImage];
   [self.lendObject setValue:
    UIImagePNGRepresentation(picture) forKey: @"image"] ;
   [self dismissModalViewControllerAnimated:YES];
}
```

Dans cette classe, vous pouvez également supprimer la propriété `picture` qui est désormais inutile.

2 Reconstruisez et testez l'application pour vérifier que les images sont conservées avec les données des objets prêtés.

Le modèle Core Data a été modifié

Pour tester cette nouvelle version de l'application, il faudra d'abord supprimer la version précédente sur le simulateur ou sur l'appareil. Par défaut, *Core Data* nécessite que le fichier de stockage utilisé soit lu et produit avec le même modèle.

Challenge

Si vous disposez d'un iPhone 3GS, vous pouvez modifier l'application pour conserver une capture vidéo plutôt qu'une photo.

Enregistrer dans l'album

Le framework *Cocoa Touch* propose des fonctions permettant d'enregistrer des images, photos ou des vidéos dans les albums par défaut de l'appareil.

Enregistrement d'une image

Pour enregistrer une image ou une photo dans l'album, utilisez la fonction qui prend pour paramètres :

- une référence à l'image à enregistrer, de type `UIImage *` ;
- une référence à l'objet, de type `id`, devant recevoir la notification de l'enregistrement, ou `nil` si vous ne souhaitez pas que l'application soit informée de la fin de l'enregistrement ;
- le sélecteur de la méthode à appeler pour la notification de l'enregistrement, ou `nil` ;
- une référence de type `void *` sur des informations, qui sera passée à la méthode de notification, ou `nil` s'il n'y a pas d'informations complémentaires à transmettre.

Le sélecteur de la méthode de notification doit prendre trois paramètres :

- la référence à l'image qui vient d'être enregistrée, de type `UIImage *` ;
- une référence vers une instance `NSError` contenant la description de l'erreur éventuelle ;
- la référence vers les informations complémentaires.

Enregistrement d'une vidéo

Pour enregistrer une capture vidéo dans l'album, utilisez la fonction `UISaveVideoAtPathToSavedPhotosAlbum` qui prend pour paramètres :

- une chaîne de caractères contenant le chemin d'accès vers la vidéo, de type `NSString *` ;
- une référence à l'objet, de type `id`, devant recevoir la notification de l'enregistrement, ou `nil` si vous ne souhaitez pas que l'application soit informée de la fin de l'enregistrement ;
- le sélecteur de la méthode à appeler pour la notification de l'enregistrement, ou `nil` ;
- une référence de type `void *` sur des informations, qui sera passée à la méthode de notification, ou `nil` s'il n'y a pas d'informations complémentaires à transmettre.

Le sélecteur de la méthode de notification doit prendre trois paramètres :

- le chemin d'accès à la vidéo qui vient d'être enregistrée, de type `NSString *` ;
- une référence vers une instance `NSError` contenant la description de l'erreur éventuelle ;
- la référence vers les informations complémentaires.

10.4. Éditer les photos

Le sélectionneur de photos est doté d'un éditeur élémentaire qui permet à l'utilisateur de *recadrer* et de *zoomer* l'image avant de la sélectionner. Pour utiliser cet éditeur, il faut :

- l'activer avant d'afficher le sélectionneur, ce qui est réalisé avec la propriété booléenne `allowsEditing` ;
- choisir l'image éditée plutôt que l'image originale.

Pour utiliser l'éditeur de photos dans l'application *Emprunts3*, modifiez les méthodes `–takePicture` et `–imagePickerController:didFinishPickingMediaWithInfo:` :

```
- (void) takePicture {
   UIImagePickerController *picker =
                    [[UIImagePickerController alloc] init];
   if ([UIImagePickerController isSourceTypeAvailable:
                UIImagePickerControllerSourceTypeCamera]){
      picker.sourceType =
                  UIImagePickerControllerSourceTypeCamera;
   } else {
      picker.sourceType =
            UIImagePickerControllerSourceTypePhotoLibrary;
   }
   picker.allowsEditing = YES;
   picker.delegate = self;
   [self presentModalViewController:picker animated:YES];
   [picker release];
}
- (void)imagePickerController:
        (UIImagePickerController *)picker
       didFinishPickingMediaWithInfo:(NSDictionary *)info{
   UIImage *picture =
   [info objectForKey:UIImagePickerControllerEditedImage];
   [self.lendObject setValue:
     UIImagePNGRepresentation(picture) forKey: @"image"] ;
   [self dismissModalViewControllerAnimated:YES];
}
```

Vous pouvez reconstruire l'application pour la tester.

10.5. Envoyer ses photos

Le framework *MessageUI* permet d'envoyer des courriels depuis une application. La classe `MFMailComposeViewController` qui y est définie est un contrôleur de composition de courriel. Il présente une interface standard à l'utilisateur, lui permettant de composer un message

et de le transmettre. Les différents champs du courriel peuvent être préremplis par l'application : destinataire, objet, pièces jointes, etc.

Classe MFMailComposeViewController

La classe `MFMailComposeViewController` s'utilise comme la plupart des contrôleurs de vue utilitaires, `UIImagePickerController` par exemple :

- création d'une instance du contrôleur de vue ;
- initialisation des propriétés de cette instance, en particulier son délégué ;
- activation du contrôleur par `–presentModalViewController:animated:`.

Le délégué du contrôleur est informé lorsque l'utilisateur souhaite fermer l'interface de composition, soit pour envoyer le courriel, soit pour l'enregistrer dans les brouillons, soit pour annuler.

Les méthodes et propriétés de la classe `MFMailComposeViewController` sont répertoriées dans le tableau.

Tableau 10.4 : Méthodes et propriétés de la classe MFMailComposeViewController

Thème	Signature	Objet
Capacité d'envoyer des courriels	`+ (BOOL)canSendMail`	Retourne `YES` si l'appareil est configuré pour envoyer des courriels.
Préremplissage des champs	`– (void) setSubject: (NSString*)subject`	Préremplit le champ *Objet*.
	`– (void) setToRecipients: (NSArray*)toRecipients`	Préremplit le champ *Destinataires*.
	`– (void) setCcRecipients: (NSArray*)ccRecipients`	Préremplit le champ *Copies*.
	`– (void) setBccRecipients: (NSArray*)bccRecipients`	Préremplit le champ *Copies cachées*.
	`– (void) setMessageBody: (NSString*)body isHTML:(BOOL)isHTML`	Préremplit le champ *Texte*.
	`– (void) addAttachmentData: (NSData*)attachment mimeType:(NSString*)mimeType fileName:(NSString*)filename`	Attache un document en précisant son contenu, son type MIME et son nom.
Délégué	`@property(nonatomic,assign) id<MFMailComposeView ControllerDelegate> mailComposeDelegate`	Délégué du contrôleur de composition de courriel

types MIME

Le type *MIME* permet au destinataire du message d'exploiter le fichier attaché en précisant son format. Par exemple, un fichier de texte pur est de type `text/plain`, une image au format *PNG* de type `image/png`, etc.

La liste des types existants est disponible à l'adresse **http://www.iana.org/assignments/media-types/**.

Protocole MFMailComposeViewControllerDelegate

Le délégué du contrôleur de composition de courriel est informé lorsque l'utilisateur souhaite fermer la vue de composition. Le protocole `MFMailComposeViewControllerDelegate` ne déclare qu'une méthode.

Tableau 10.5 : Méthodes du protocole MFMailComposeViewControllerDelegate

Méthode	Objet
`– (void) mailComposeController: (MFMailComposeViewController*) controller didFinishWith Result:(MFMailComposeResult) result error:(NSError*)error`	Méthode appelée lorsque l'utilisateur veut refermer la fenêtre de composition de courriel.

C'est dans cette méthode que le développeur doit appeler `–dismissModalViewControllerAnimated:` pour désactiver le contrôleur de composition de courriel.

Les valeurs de retours définies dans le type énuméré `MFMailComposeResult` sont :

- `MFMailComposeResultCancelled` si l'utilisateur a annulé la composition ;
- `MFMailComposeResultSaved` si l'utilisateur a enregistré le courriel dans les brouillons ;
- `MFMailComposeResultSent` si l'utilisateur a envoyé le courriel ;
- `MFMailComposeResultFailed` en cas d'erreur.

l'envoi est différé

Un résultat `MFMailComposeResultSent` ne signifie pas que le courriel est effectivement parti. Il a été placé dans la boîte d'envoi et sera envoyé à la

première occasion, par exemple lorsque l'appareil accrochera un réseau Wi-Fi.

Challenge

Complétez l'application *Emprunts3* en ajoutant la capacité d'envoyer la photo prise lors du prêt. Cela vous permettra de vous rappeler au bon souvenir des amis indélicats…

10.6. Checklist

Nous avons appris dans ce chapitre à utiliser l'album photo de l'appareil, et l'appareil de prise de vues ou de capture vidéo pour ceux qui en sont doté. Pour cela, nous avons détaillé le fonctionnement :

- de la classe `UIImagePickerController` ;
- du protocole `UIImagePickerControllerDelegate`.

Nous avons vu comment utiliser la classe `NSData` pour enregistrer et récupérer des images dans une structure *Core Data*, grâce aux attributs de type binaire.

Nous savons maintenant doter nos applications de la capacité de préparer et envoyer des courriels avec le framework *MessageUI* :

- classe `MFMailComposeViewController` ;
- protocole `MFMailComposeViewControllerDelegate`.

GÉO-LOCALISATION

Deux frameworks sont au programme de ce chapitre :

- *CoreLocation*, qui permet d'utiliser les capacités de géo-localisation de l'iPhone ;
- *MapKit*, qui permet d'insérer des cartes géographiques dans une application et qui s'utilise en conjonction avec *CoreLocation*, par exemple, pour y visualiser des marqueurs.

11.1. Déterminer sa position

Technologies de géo-localisation

L'iPhone met en œuvre simultanément plusieurs technologies pour localiser sa position géographique :

- la localisation des réseaux Wi-Fi publics, qui est une technologie relativement précise (quelques dizaines de mètres), mais n'est pas disponible partout ;
- la triangulation des antennes relais de téléphonie mobile, partout disponible mais avec une précision très variable (en montagne, par exemple, le nombre d'antennes relais accessibles est souvent insuffisant pour obtenir une bonne précision) ;
- le positionnement par satellites GPS, très précis en plein air et généralement indisponible à l'intérieur des bâtiments.

La précision de la géo-localisation va de plusieurs kilomètres à quelques mètres. Il faut être conscient qu'une bonne précision nécessite des calculs, donc du temps (souvent plusieurs secondes), et consomme également de l'énergie. Il est recommandé de limiter la précision demandée au strict nécessaire en fonction de l'application.

La mise en œuvre des différentes technologies de géo-localisation est transparente pour le développeur. Le *gestionnaire de géo-localisation* en masque la complexité pour se concentrer sur l'essentiel : fournir une localisation avec le niveau de précision requis.

Classe CLLocationManager

Mise en œuvre du gestionnaire de géo-localisation

La mise en œuvre de la géo-localisation utilise un motif analogue à celui du sélectionneur d'images vu au chapitre précédent et à celui des accéléromètres que nous verrons au chapitre suivant :

- Création d'une instance de la classe `CLLocationManager` (le gestionnaire de géo-localisation) qui est programmée pour définir les critères de notifications relatifs à la position géographique de l'appareil.

- Activation du gestionnaire ; il commence à délivrer des *notifications*.
- Les notifications sont délivrées au *délégué* du gestionnaire qui doit répondre au protocole `CLLocationManagerDelegate`.
- Le gestionnaire est désactivé lorsque les notifications ne sont plus nécessaires.

Les deux premières étapes sont réalisées typiquement par les instructions suivantes :

```
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
locationManager.desiredAccuracy =
                              kCLLocationAccuracyKilometer;
locationManager.distanceFilter = 500.;
[locationManager startUpdatingLocation];
```

Dans ce cas, le gestionnaire est programmé pour délivrer des notifications précise au kilomètre près et à chaque fois que l'appareil s'est déplacé de 500 mètres ou plus.

Les notifications sont reçues par le délégué sur sa méthode `–locationManager:didUpdateToLocation:fromLocation:` détaillée dans la section relative au protocole `CLLocationManagerDelegate`.

La dernière étape est réalisée par l'instruction suivante :

```
[locationManager stopUpdatingLocation];
```

Détails de la classe CLLocationManager

La classe `CLLocationManager` permet de gérer les notifications relatives à la position géographique de l'appareil et celles relatives à son orientation par rapport au Nord, dont il sera question plus loin dans ce chapitre. Les méthodes et propriétés de la classe sont détaillées dans le tableau ci-après.

Tableau 11.1 : Méthodes et propriétés de la classe CLLocationManager

Thème	Signature	Objet
Configuration des mises à jour de la position	`@property(assign, NS_ NONATOMIC_ IPHONEONLY) id<CLLocationManager Delegate> delegate`	Délégué du gestionnaire de géo-localisation
	`@property(assign, NS_ NONATOMIC_IPHONEONLY) CLLocation Distance distanceFilter`	Distance de déplacement minimale entre deux événements de mise à jour de la position
	`@property(assign, NS_NONATOMIC_IPHONEONLY) CLLocation Accuracy desiredAccuracy`	Précision demandée (non garantie)

Tableau 11.1 : Méthodes et propriétés de la classe CLLocationManager

Thème	Signature	Objet
Configuration des mises à jour des directions	`@property(assign, nonatomic) CLLocation Degrees headingFilter`	Distance de déplacement minimale entre deux événements de mise à jour de la direction du Nord
	`@property(readonly, nonatomic) BOOL headingAvailable`	Retourne `YES` si l'appareil dispose de la capacité de déterminer la direction du Nord (compas magnétique).
Démarrer et arrêter les mises à jour	`– (void) startUpdating Location`	Démarre les mises à jour de géo-localisation.
	`– (void) stopUpdating Location`	Stoppe les mises à jour de géo-localisation.
	`– (void) startUpdating Heading`	Démarre les mises à jour de la direction du Nord.
	`– (void) stopUpdating Heading`	Stoppe les mises à jour de la direction du Nord.
	`– (void) dismissHeading CalibrationDisplay`	Referme le panneau de calibration magnétique.
Disponibilité des services de géo-localisation	`@property(readonly, NS_ NONATOMIC_IPHONEONLY) BOOL locationServices Enabled`	`YES` si l'utilisateur de l'appareil a autorisé la géo-localisation dans les préférences système
	`@property(copy, nonatomic) NSString *purpose`	Chaîne de caractères à afficher en même temps que le message demandant à l'utilisateur l'autorisation d'utiliser la géo-localisation
Obtenir la position de l'appareil	`@property(readonly, NS_ NONATOMIC_IPHONEONLY) CLLocation *location`	Dernière position mise à jour

La plupart des propriétés de la classe `CLLocationManager` sont déclarées avec une clause `NS_NONATOMIC_IPHONEONLY`. En effet, le framework *CoreLocation* étant commun à *Cocoa* sur Mac OS X et à *Cocoa Touch* sur iPhone OS, cette déclaration permet de définir des propriétés atomiques sur Mac OS X et non-atomiques sur iPhone OS. Rappelons qu'une propriété est dite atomique si ses accesseurs permettent d'en garantir la validité, même dans un environnement multithreading où le même objet peut être manipulé simultanément par plusieurs *threads*. La priorité est mise sur l'intégrité des données sur Mac OS X (on peut utiliser le même gestionnaire simultanément dans plusieurs threads), alors qu'elle est mise sur les performances sur iPhone OS.

Propriété purpose
La propriété `purpose` est disponible uniquement à partir de la version 3.2.

Types scalaires

Outre la classe `CLLocationManager`, le framework *Core Location* définit également la classe `CLLocation` et le protocole `CLLocationManagerDelegate`, que nous détaillerons plus loin, ainsi que les types scalaires décrits ci-après.

Tableau 11.2 : Types scalaires CLLocation

Type CLLocation	Type C	Utilisation
`CLLocationAccuracy`	`double`	Précision de distance en mètres
`CLLocationCoordinate2D`	`struct { CLLocationDegrees latitude; CLLocationDegrees longitude;}`	Le champ *latitude* prend une valeur comprise entre -90 et +90 (une valeur positive indique une latitude dans l'hémisphère Nord et négative dans l'hémisphère Sud) Le champ *longitude* prend une valeur comprise entre -180 et +180 (une valeur positive indique une longitude à l'Est du méridien de Greenwich, et négative à l'Ouest)
`CLLocationDegrees`	`double`	Angle en degrés
`CLLocationDirection`	`double`	Direction du Nord en degrés
`CLLocationDistance`	`double`	Distance en mètres
`CLLocationSpeed`	`double`	Vitesse en mètres par seconde

Quelques constantes sont définies dans le framework *Core Location* pour en faciliter l'usage :

- `kCLDistanceFilterNone`, de type `CLLocationDistance`, est utilisé avec la propriété `distanceFilter` du gestionnaire de géo-localisation pour lui indiquer de ne pas filtrer les notifications de position.
- Plusieurs constantes sont définies pour le type `CLLocationAccuracy` et la propriété `desiredAccuracy` du gestionnaire de géo-localisation :
  - `kCLLocationAccuracyBest` pour obtenir la meilleure précision possible ;

— `kCLLocationAccuracyNearestTenMeters` pour obtenir une position à 10 mètres près ;

— `kCLLocationAccuracyHundredMeters` pour obtenir une position à 100 mètres près ;

— `kCLLocationAccuracyKilometer` pour obtenir une position à 1 kilomètre près ;

— `kCLLocationAccuracyThreeKilometers` pour obtenir une position à 3 kilomètres près.

Protocole CLLocationManagerDelegate

Le tableau résume les méthodes déclarées dans le protocole `CLLocationManagerDelegate`. Toutes ces méthodes sont optionnelles.

Tableau 11.3 : Méthodes du protocole CLLocationManagerDelegate

Thème	Signature	Objet
Événements de mise à jour de la géo-localisation	`- (void) locationManager:(CLLocationManager *)manager didUpdateToLocation:(CLLocation *)newLocation fromLocation:(CLLocation *)oldLocation`	Mise à jour de la géo-localisation. La méthode reçoit la nouvelle position et l'ancienne.
	`- (void) locationManager:(CLLocationManager *)manager didFailWithError:(NSError *)error`	Une erreur s'est produite pendant la géo-localisation.
Événements de mise à jour de la direction du Nord	`- (void) locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading *)newHeading`	Mise à jour de la direction du Nord. La méthode reçoit la nouvelle direction.
	`- (BOOL) locationManagerShouldDisplayHeadingCalibration:(CLLocationManager *)manager`	Doit retourner `YES` si le délégué autorise l'affichage du panneau de calibration magnétique.

Le délégué du gestionnaire de géo-localisation reçoit les notifications :

- des mises à jour de la géo-localisation de l'appareil, pour savoir où se situe l'appareil sur le globe terrestre ;
- des mises à jour de la direction du Nord, pour savoir commet est orienté l'appareil par rapport au Nord.

Le protocole `CLLocationManagerDelegate` permet également au délégué d'être informé d'une erreur de géo-localisation ou de la néces-

sité d'une *calibration magnétique* (voir plus loin la section *Déterminer l'orientation géographique*).

Classe CLLocation

Les notifications de géo-localisation délivrent la position de l'appareil sous forme d'instances de la classe `CLLocation` décrite dans le tableau ci-après.

Tableau 11.4 : Méthodes et propriétés de la classe CLLocation

Thème	Signature	Objet
Initialisation	`– (id) initWithLatitude: (CLLocationDegrees) latitude longitude: (CLLocationDegrees) longitude`	Crée une instance avec les coordonnées passées en paramètre. La précision horizontale prend une valeur nulle, la précision verticale prend la valeur –1. L'horodate est celle de l'initialisation de l'objet.
	`– (id) initWith Coordinate:(CLLocation Coordinate2D)coordinate altitude:(CLLocation Distance)altitude horizontalAccuracy: (CLLocationAccuracy) hAccuracyvertical Accuracy:(CLLocation Accuracy)vAccuracy timestamp:(NSDate *) timestamp`	Crée une instance avec les données passées en paramètre.
Attributs de géo-localisation	`@property(readonly, NS_NONATOMIC_IPHONEONLY) CLLocationCoordinate2D coordinate`	Coordonnées de la position
	`@property(readonly, NS_NONATOMIC_IPHONEONLY) CLLocationDistance altitude`	Altitude
	`@property(readonly, NS_NONATOMIC_IPHONEONLY) CLLocationAccuracy horizontalAccuracy`	Précision horizontale
	`@property(readonly, NS_NONATOMIC_IPHONEONLY) CLLocationAccuracy verticalAccuracy`	Précision verticale
	`@property(readonly, NS_NONATOMIC_IPHONEONLY) NSDate *timestamp`	Horodate à laquelle la position a été déterminée
	`– (NSString *)description`	Retourne une chaîne de caractères contenant la description de la position.

Tableau 11.4 : Méthodes et propriétés de la classe CLLocation

Thème	Signature	Objet
Mesure de distance	`- (CLLocationDistance) getDistanceFrom:(const CLLocation *)location`	Retourne la distance entre la position du récepteur et celle de l'instance passée en paramètre.
Détermination du mouvement	`@property(readonly, NS_NONATOMIC_IPHONEONLY) CLLocationSpeed speed`	Vitesse instantanée de l'appareil. Une valeur négative indique que cette propriété est indisponible.
	`@property(readonly, NS_NONATOMIC_IPHONEONLY) CLLocationDirection course`	Direction de déplacement de l'appareil Une valeur négative indique que cette propriété est indisponible.

La classe `CLLocation` permet non seulement de retrouver la position de l'appareil, sous forme de *latitude* et de *longitude*, mais aussi :

- l'altitude de l'appareil ;
- la précision horizontale et verticale ;
- l'horodate précise de la mesure ;
- s'il y a lieu, la vitesse et la direction de déplacement de l'appareil.

Challenge

Réalisez une application qui affiche la position de l'appareil.

Vous pourrez agrémenter cette application de fonctions permettant à l'utilisateur de jouer sur la précision recherchée et sur le filtre en distance pour les notifications de géo-localisation.

11.2. Déterminer l'orientation géographique

L'iPhone 3GS est doté d'un *compas magnétique* permettant soit de mesurer un champ magnétique, soit de déterminer l'inclinaison de l'appareil par rapport à la direction du Nord ; les applications *Boussole* et *Plans* (sur iPhone 3GS) utilisent le compas magnétique.

Le compas magnétique est mis en œuvre dans une application à l'aide du gestionnaire de géo-localisation dont nous venons de traiter. Il peut être étonnant au premier abord de mélanger la géo-localisation et la détection magnétique dans un même framework. L'explication est simple, la détermination de la direction du Nord géographique nécessite la connaissance de la position de l'appareil sur le globe terrestre afin de calculer la déclinaison magnétique ; l'écart entre les directions du Nord géographique et du Nord magnétique.

Mise en œuvre du compas magnétique

La mise en œuvre du compas magnétique utilise le gestionnaire de géo-localisation et son motif standard. Les méthodes et propriétés traitant de la géo-localisation et du compas magnétique y sont bien séparées :

- Une instance de la classe `CLLocationManager` (le gestionnaire de géo-localisation) est créée.
- On vérifie que le compas magnétique est disponible sur l'appareil avec la propriété `headingAvailable`.
- Le gestionnaire de géo-localisation est programmé pour définir les critères de notification relatifs à la l'inclinaison de l'appareil par rapport à la direction du Nord.
- Le gestionnaire est activé ; il commence à délivrer des *notifications*.
- Les notifications sont délivrées au *délégué* du gestionnaire qui doit répondre au protocole `CLLocationManagerDelegate`.
- Le gestionnaire est désactivé lorsque les notifications ne sont plus nécessaires.

Les quatre premières étapes sont réalisées typiquement par les instructions suivantes :

```
locationManager = [[CLLocationManager alloc] init];
if (locationManager.headingAvailable) then {
   locationManager.delegate = self;
   locationManager.headingFilter = 5.;
   [locationManager startUpdatingHeading];
}
```

Dans ce cas, le gestionnaire est programmé pour délivrer des notifications chaque fois que l'inclinaison de l'appareil change de 5 degrés ou plus.

Les notifications sont reçues par le délégué sur sa méthode `–locationManager:didUpdateHeading:` détaillée dans la section relative au protocole `CLLocationManagerDelegate`.

La dernière étape est réalisée par l'instruction suivante :

```
[locationManager stopUpdatingHeading];
```

un seul délégué

Bien que les méthodes et propriétés relatives à la géo-localisation et au compas magnétique soient séparées dans la classe `CLLocationManager` et dans le protocole associé, le délégué du gestionnaire de géo-localisation est unique pour ces deux fonctionnalités.

Calibration magnétique

Un compas magnétique étant très sensible aux perturbations, fréquentes dans notre univers domestique et ses nombreux appareils électroniques, le gestionnaire de géo-localisation peut avoir à afficher un panneau de *calibration*. Ce dernier invite l'utilisateur à faire des 8 avec l'appareil ou à s'éloigner d'une source magnétique trop forte.

Dans ce cas, le délégué du gestionnaire reçoit le message `–locationManagerShouldDisplayHeadingCalibration:`. Il doit répondre `YES` s'il autorise l'affichage du panneau et `NO` dans le cas contraire. Par défaut, s'il n'implémente pas cette méthode, le panneau n'est pas affiché.

Classe CLHeading

L'information d'inclinaison par rapport au Nord est transmise au délégué sous la forme d'une instance de la classe `CLHeading`.

Tableau 11.5 : Méthodes et propriétés de la classe CLHeading

Thème	Signature	Objet
Attributs de Direction	`@property(readonly, nonatomic) CLLocation Direction magneticHeading`	Direction vers laquelle pointe le sommet de l'appareil, par rapport au Nord magnétique (`0` pour le Nord, `90` pour l'Est, etc.) Une valeur négative indique que cette propriété ne peut être calculée.
	`@property(readonly, nonatomic) CLLocation Direction trueHeading`	Direction vers laquelle pointe le sommet de l'appareil, par rapport au Nord géographique (`0` pour le Nord, `90` pour l'Est, etc.) Une valeur négative indique que cette propriété ne peut être calculée. Le calcul de cette propriété nécessite que l'appareil puisse être géo-localisé.
	`@property(readonly, nonatomic) CLLocation Direction headingAccuracy`	Estimation de l'erreur sur la direction du Nord magnétique Une valeur négative indique que cette propriété ne peut être calculée.
	`@property(readonly, nonatomic) NSDate *timestamp`	Horodate de la mesure du compas magnétique
	`– (NSString *) description`	Description de la mesure du compas magnétique sous forme de chaîne de caractères

Tableau 11.5 : Méthodes et propriétés de la classe CLHeading

Thème	Signature	Objet
Mesures brutes	`@property(readonly, nonatomic) CLHeadingComponentValue x`	Mesure du champ magnétique en microTesla selon l'axe des abscisses
	`@property(readonly, nonatomic) CLHeadingComponentValue y`	Mesure du champ magnétique en microTesla selon l'axe des ordonnées
	`@property(readonly, nonatomic) CLHeadingComponentValue z`	Mesure du champ magnétique en microTesla selon l'axe des profondeurs

La propriété `magneticHeading` contient l'inclinaison de l'axe de l'appareil par rapport à la direction du Nord magnétique. La propriété `trueHeading` contient l'inclinaison par rapport au Nord géographique.

le Nord géographique nécessite la géo-localisation

La propriété `trueHeading` est mise à jour uniquement si la notification de la géo-localisation est activée sur le gestionnaire de géo-localisation.

11.3. Framework MapKit

Lorsque l'on traite de géo-localisation, le besoin de visualiser une carte vient naturellement. Nous allons voir dans cette section comment afficher une *carte* et permettre à l'utilisateur d'interagir avec elle ; nous mettrons en œuvre le framework *MapKit*.

Afficher une carte

Créez un nouveau projet sous XCode, de type **View Based Application**. Nommez-le *Carte*.

Ajouter le framework MapKit

Le framework *MapKit* n'est pas intégré par défaut aux projets XCode ; rappelons la démarche à suivre pour ajouter un framework à un projet :

1 Dans la zone **Groups&Files** de la fenêtre principale, ouvrez le groupe **Targets**, sélectionnez la cible *Carte* et cliquez du bouton droit pour afficher le menu contextuel.

2 Sélectionnez la commande **Get Info**. Dans le panneau d'information qui s'affiche, choisissez l'onglet **General** et cliquez sur le bouton **+** de la partie *Linked Libraries*.

3 Choisissez le fichier *MapKit.framework* pour ajouter le framework *MapKit* au projet *Carte*.

Ajouter une vue Carte

1 Sous XCode, double-cliquez sur le fichier *CarteViewController.xib* pour l'ouvrir. Ajoutez une vue cartographique (**Map View**) sur l'interface utilisateur (voir Figure 11.1).

2 Utilisez l'inspecteur de taille (⌘+3) pour définir une vue carrée, par exemple une hauteur et une largeur de 280 pixels.

La vue cartographique que nous venons d'insérer est de la classe `MKMapView`. Dans la suite de ce chapitre, nous examinerons les caractéristiques les plus courantes de la classe `MKMapView` qui est au centre du framework *MapKit*.

Figure 11.1 : Vue cartographique sous Interface Builder

Tester l'application

Construisez l'application sous XCode et testez-la (⌘+R).

Figure 11.2 : Carte affichée par défaut

Par défaut, la carte affichée est celle du pays défini dans les préférences de localisation de l'appareil ou le planisphère complet. Vous pouvez la déplacer ou zoomer. Nous allons voir comment connaître et contrôler le positionnement de la carte par programmation.

Connaître la zone affichée

La zone affichée sur la carte est accessible par la propriété `region` de la classe `MKMapView`. Nous allons en illustrer le fonctionnement en modifiant l'application *Carte* pour que l'utilisateur puisse en visualiser les caractéristiques.

Définir les outlets

Sous Interface Builder, ajoutez quatre labels de texte sur la vue principale afin d'y afficher les coordonnées du centre de la carte (latitude et longitude) et la taille de la zone affichée (hauteur et largeur).

1 Sous XCode, ouvrez le fichier *CarteViewController.h* pour y ajouter les outlets permettant d'accéder aux éléments de la vue principale.

Nous y définissons également le contrôleur comme répondant au protocole `MKMapViewDelegate` pour qu'il soit notifié des changements sur la carte :

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
@interface CarteViewController : UIViewController
                                    <MKMapViewDelegate> {
   IBOutlet MKMapView * carte;
   IBOutlet UILabel * latitudeLabel;
   IBOutlet UILabel * longitudeLabel;
   IBOutlet UILabel * hauteurLabel;
   IBOutlet UILabel * largeurLabel;
}
@property(retain,nonatomic) MKMapView * carte;
@property(retain,nonatomic) UILabel * latitudeLabel;
@property(retain,nonatomic) UILabel * longitudeLabel;
@property(retain,nonatomic) UILabel * hauteurLabel;
@property(retain,nonatomic) UILabel * largeurLabel;
@end
```

inclure MapKit.h

Le framework *MapKit* n'étant pas inclus par défaut dans le projet, il faut ajouter une clause `#import <MapKit/MapKit.h>` dans les fichiers sources qui doivent l'utiliser.

2 Revenez sous Interface Builder pour attacher les outlets du contrôleur aux éléments de la vue et définissez le *délégué* de la vue `MKMapView` comme étant le propriétaire du fichier.

Figure 11.3 : Liens de la vue MKMapView

Code source du contrôleur de vue

1 Sous XCode, ajoutez les accesseurs des propriétés de la classe `CarteViewController` dans son fichier source :

```
@implementation CarteViewController
@synthesize carte, latitudeLabel, longitudeLabel,
                                  hauteurLabel, largeurLabel;
```

2 Ajoutez la définition de la méthode `-mapView:regionDidChangeAnimated:` du protocole `MKMapViewDelegate`. Cette méthode est appelée chaque fois que la propriété `region` est modifiée :

```
- (void)mapView:(MKMapView *)mapView
                regionDidChangeAnimated:(BOOL)animated{
    MKCoordinateRegion region = carte.region;
    latitudeLabel.text = [NSString stringWithFormat:@"%g",
                                    region.center.latitude];
    longitudeLabel.text = [NSString stringWithFormat:@"%g",
                                   region.center.longitude];
    hauteurLabel.text = [NSString stringWithFormat:@"%g",
                                  region.span.latitudeDelta];
    largeurLabel.text = [NSString stringWithFormat:@"%g",
                                 region.span.longitudeDelta];
}
```

3 Construisez l'application pour la tester.

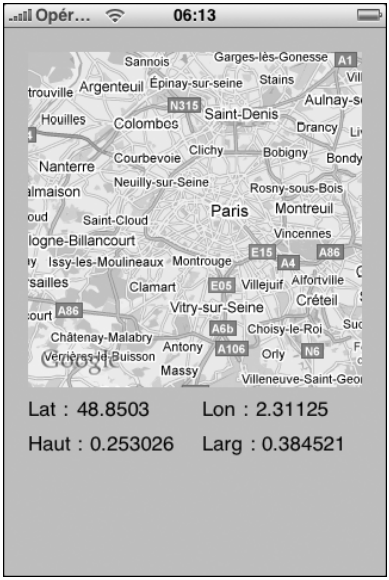


Figure 11.4 : Visualisation des caractéristiques de la région affichée

Propriété region

La propriété `region` de la classe `MKMapView` est une structure de type `MKCoordinateRegion` composée de :

- `center` qui est une structure de type `CLLocationCoordinate2D` elle-même composée de :
  - `latitude` de type `CLLocationDegrees` ;
  - `longitude` de type `CLLocationDegrees`.

- `span` qui est une structure de type `MKCoordinateSpan` composée de :
  - `latitudeDelta` de type `CLLocationDegrees` ;
  - `longitudeDelta` de type `CLLocationDegrees`.

Ainsi une région sur la carte est définie par son centre (latitude et longitude) et par sa taille, elle-même exprimée en écarts de latitude et de longitude. Un écart de latitude de un degré représente une hauteur de 111 km. Un écart de longitude de un degré représente une largeur qui dépend de la latitude : 111 km à l'équateur et 0 aux pôles.

Contrôler la zone affichée

Maintenant que nous savons extraire la zone affichée dans une instance de la classe `MKMapView`, nous allons compléter notre application *Carte* afin qu'elle nous permette de mémoriser une région pour y revenir plus tard.

Définir l'interface

1. Ajoutez deux boutons sur l'interface utilisateur, **Définir Zone** et **Retrouver Zone**. Ajoutez une action pour chacun de ces boutons dans l'interface du contrôleur de vue de l'application, respectivement `defineZone` et `retrieveZone`. Liez les boutons et les actions sous Interface Builder.
2. Toujours dans l'interface du contrôleur de vue, déclarez une nouvelle propriété `zone` de type `MKCoordinateRegion`. Cette propriété n'étant pas une référence, elle doit être déclarée avec la clause `assign` au lieu de `retain`.

Code du contrôleur

1. Dans le fichier *CarteViewController.m*, synthétisez les accesseurs de la nouvelle propriété `zone`, puis définissez les méthodes pour les actions :

```
-(IBAction) defineZone{
   self.zone = carte.region;
}
-(IBAction) retrieveZone{
   [carte setRegion:self.zone animated:YES];
}
```

Nous utilisons ici la méthode `-setRegion:animated:` de la classe `MKMapView` pour définir la région à visualiser sur la carte.

2. Construisez l'application pour la tester.

Appréhender la vue satellite

La classe `MKMapView` dispose des propriétés suivantes pour modifier son comportement :

- `mapType` de type `MKMapType` ;
- `scrollEnabled` de type Booléen ;
- `zoomEnabled` de type Booléen.

`scrollEnabled` et `zoomEnabled` permettent d'autoriser respectivement le déplacement de la carte et le zoom par l'utilisateur. Ces propriétés ont la valeur `YES` par défaut.

`MKMapType` est un type énuméré qui permet de définir le type de visualisation de la carte :

- `MKMapTypeStandard`, pour visualiser le plan ;
- `MKMapTypeSatellite`, pour visualiser la vue satellite ;
- `MKMapTypeHybrid`, pour visualiser la vue satellite augmentée d'information.

Challenge

Modifiez l'application *Carte* pour que l'utilisateur puisse choisir entre les différents types de visualisation.

Figure 11.5 : Différents types de visualisation

Annoter la carte

Nous allons terminer ce parcours du framework *MapKit* par la mise en œuvre du protocole `MKAnnotation` et de la méthode `–addAnnotation:` qui permettent d'ajouter sur la carte des marqueurs en forme d'épingle à tête.

Créer une annotation

Il n'y a pas de classe spécifique pour contenir une *annotation*. N'importe quel objet fait l'affaire pourvu qu'il respecte le protocole `MKAnnotation` qui définit trois propriétés :

- `coordinate`, propriété obligatoire de type `CLLocationCoordinate2D`, pour définir l'emplacement du marqueur sur la carte ;
- `title` et `subTitle`, propriétés optionnelles de type `NSString *` ; ces chaînes de caractères sont affichées lorsque l'utilisateur touche le marqueur.

Procédez ainsi :

1. Sous XCode, créez une nouvelle classe `Annotation` dérivant de `NSOject`. Complétez l'interface de la classe pour y déclarer qu'elle adopte le protocole `MKAnnotation` et ses propriétés :

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@interface Annotation : NSObject <MKAnnotation> {
   CLLocationCoordinate2D coordinate;
   NSString * title;
   NSString * subTitle;
}
@property(assign,nonatomic)
                     CLLocationCoordinate2D coordinate;
@property(retain,nonatomic) NSString * title;
@property(retain,nonatomic) NSString * subTitle;
@end
```

2. Complétez la définition de la classe en définissant les accesseurs pour les propriétés dans le fichier *Annotation.m*. Cette classe est un réceptacle de données, elle ne contient pas d'autres méthodes :

```
#import "Annotation.h"
@implementation Annotation
@synthesize coordinate, title, subTitle;
@end
```

Afficher un marqueur

1. Sous XCode, ouvrez le fichier *CarteViewController.m* pour y déclarer la classe `Annotation` :

```
#import "Annotation.h"
```

2 Modifiez la méthode `-defineZone` pour ajouter un marqueur au centre de la carte :

```
-(IBAction) defineZone{
    self.zone = carte.region;
    Annotation * annotation = [[Annotation alloc] init];
    annotation.coordinate = self.zone.center;
    annotation.title = @"Centre de la zone";
    [carte addAnnotation:annotation];
    [annotation release];
}
```

3 Construisez l'application pour la tester. Si vous touchez le marqueur défini en même temps que la zone, le texte s'affiche.

Figure 11.6 : Carte avec un marqueur

Nous vous laissons explorer le framework *MapKit* et les fonctions permettant d'adapter les vues associées aux annotations.

11.4. Checklist

Nous connaissons maintenant les différentes technologies mises en œuvre pour la géo-localisation de l'appareil, et nous savons mettre en œuvre le framework *Core Location* : le gestionnaire de géo-

localisation, instance de la classe `CLLocationManager`, et son protocole de délégué `CLLocationManagerDelegate`.

Nous avons détaillé les classes décrivant la position de l'appareil :

- `CLLocation` pour la position sur le globe terrestre ;
- `CLHeading` pour l'inclinaison de l'appareil par rapport au Nord, géographique ou magnétique.

Enfin, nous avons appris à utiliser le framework *MapKit*, la vue `MKMapView` et son délégué `MKMapViewDelegate` pour :

- visualiser et manipuler une carte ;
- connaître la zone géographique visualisée ;
- modifier le mode de visualisation de la carte ;
- insérer des annotations sur la carte.

ACCÉLÉROMÈTRES

Tous les appareils, iPhone et iPod Touch, sont dotés de trois *accéléromètres*. Grâce à ces derniers, on peut connaître l'orientation de l'appareil par rapport à la verticale ainsi que ses mouvements ; ce chapitre est consacré à l'étude de leur utilisation.

Nous commencerons par quelques expérimentations avec les classes `UIAccelerometer` et `UIAcceleration`, puis nous nous intéresserons à la détection des mouvements et à la détermination de la position de l'appareil.

Testez sur un appareil

Les exemples développés dans ce chapitre ne fonctionnent pas sur le simulateur qui ne dispose pas d'accéléromètres.

L'annexe décrit le mode opératoire à suivre pour tester vos applications sur un appareil réel.

12.1. Utiliser les accéléromètres

Nous allons commencer par écrire une application nous permettant de visualiser les données fournies par les accéléromètres. Elle sera notre support d'expérimentation qui nous permettra de mieux comprendre le fonctionnement de ces "petites bêtes", et donc de mieux les utiliser par la suite.

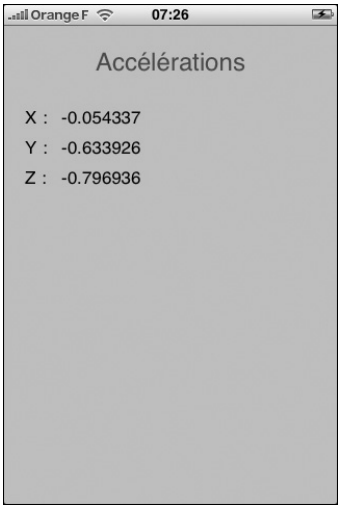


Figure 12.1 : Visualiser les accélérations

Visualiser l'accélération

1 Créez un projet de type **View Based Application** sous XCode et intitulez-le `Accelero`.

Notre première expérimentation contient trois *labels*, qui contiendront les composantes données par les trois accéléromètres. Nous devons définir les *outlets* pour ces labels dans notre contrôleur de vue et il faut également qu'il se conforme au protocole `UIAccelerometerDelegate`.

2 Modifiez le fichier *AcceleroViewController.h* :

```
#import <UIKit/UIKit.h>
@interface AcceleroViewController : UIViewController
                                    <UIAccelerometerDelegate> {
   IBOutlet UILabel * xLabel;
   IBOutlet UILabel * yLabel;
   IBOutlet UILabel * zLabel;
}
@property(nonatomic,retain) UILabel * xLabel;
@property(nonatomic,retain) UILabel * yLabel;
@property(nonatomic,retain) UILabel * zLabel;
@end
```

Le protocole `UIAcceleromerDelegate` définit une seule méthode, `-(void)accelerometer:(UIAccelerometer *)didAccelerate:(UIAcceleration *)` qui fournit :

- une instance de `UIAcceleromer` représentant les trois accéléromètres ;
- une instance de `UIAcceleration`, conteneur d'une mesure d'accélérations.

Tableau 12.1 : Propriétés de la classe UIAcceleration

Propriétés	Objet
`@property(nonatomic, readonly) UIAccelerationValue x`	Accélération en g sur l'axe des abscisses
`@property(nonatomic, readonly) UIAccelerationValue y`	Accélération en g sur l'axe des ordonnées
`@property(nonatomic, readonly) UIAccelerationValue z`	Accélération en g sur l'axe des profondeurs
`@property(nonatomic, readonly) NSTimeInterval timestamp`	L'horodate de la mesure d'accélération, en secondes, depuis le démarrage de l'appareil

Type UIAccelerationValue

Le type `UIAccelerationValue` est équivalent au type `double`.

Orientation des axes

Les axes utilisés par les accéléromètres sont les mêmes que ceux d'OpenGL ES ; lorsque l'appareil est en mode Portrait, le bouton principal vers le bas, les abscisses sont disposées de gauche à droite, les ordonnées du bas vers le haut et les profondeurs du dos vers l'avant de l'appareil

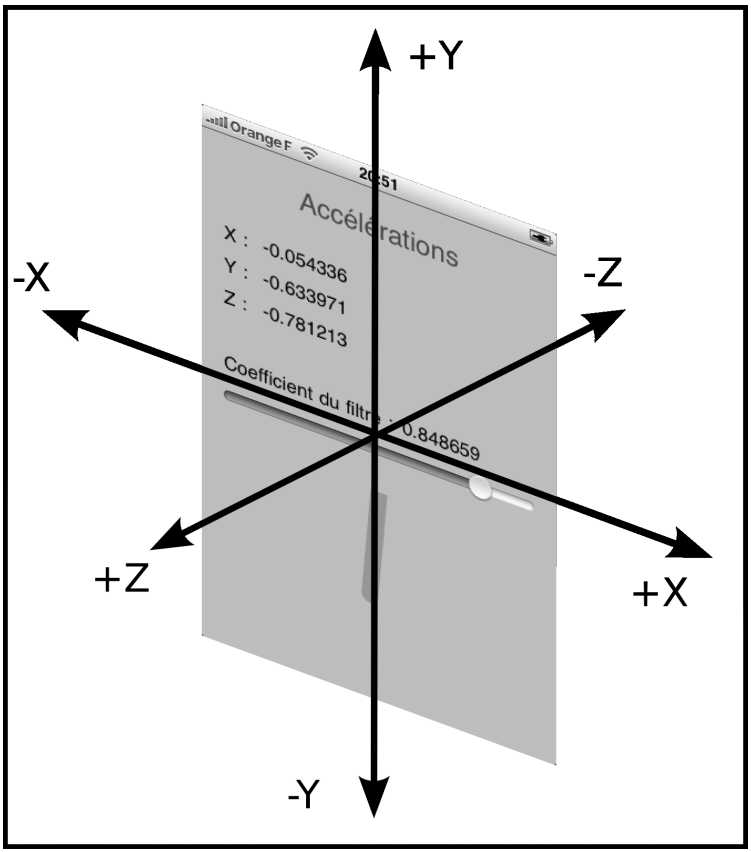


Figure 12.2 : Orientation des composantes d'accélération

3 Écrivez le code de cette méthode de délégué dans le fichier *AcceleroViewController.m*. Nous voulons simplement afficher les valeurs des composantes de l'accélération mesurée :

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
                        didAccelerate:(UIAcceleration *)acc
{
   xLabel.text = [NSString stringWithFormat:@"%f", acc.x];
   yLabel.text = [NSString stringWithFormat:@"%f", acc.y];
   zLabel.text = [NSString stringWithFormat:@"%f", acc.z];
}
```

4 Activez les mesures et modifiez la méthode `viewDidLoad` dans le fichier *AcceleroViewController.m* :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   UIAccelerometer *accelerometer =
                  [UIAccelerometer sharedAccelerometer];
   accelerometer.updateInterval = 0.1;
   accelerometer.delegate = self;
}
```

La variable `accelerometer` y est définie comme l'instance partagée (unique) de la classe `UIAccelerometer`, puis nous définissons la périodicité des mesures ($1/10^e$ de seconde, nous n'arriverons pas à lire plus vite). Enfin, nous définissons le délégué qui recevra les mesures.

5 Synthétisez les accesseurs des propriétés avec `@synthesize xLabel, yLabel, zLabel;` et libérez-les dans la méthode `viewDidUnload` :

```
- (void)viewDidUnload {
   self.xLabel = nil;
   self.yLabel = nil;
   self.zLabel = nil;
}
```

6 Ouvrez le fichier *AcceleroViewController.xib* pour placer trois labels dans la vue principale et les lier aux *outlets* du contrôleur de vue.

7 Testez l'application sur votre appareil réel ; le simulateur ne permet pas d'émuler les accéléromètres. Au besoin, consultez l'annexe A qui détaille le mode opératoire à suivre pour charger et tester une application sur un appareil réel.

Des valeurs non nulles s'affichent, même lorsque l'appareil est immobile. C'est satisfaisant de voir que notre application affiche des éléments mais il faut expliquer pourquoi *notre appareil accélère quand il ne bouge pas*.

Un accéléromètre est en fait un capteur de force, et tous les objets sur la Terre sont soumis à la force de la gravitation. Nous voilà dans la même position qu'Isaac Newton qui eut l'intuition de la loi de la gravitation universelle en observant la chute d'une pomme ; nous découvrons la même chose avec un iPhone, de marque Apple, évidemment.

Lorsque l'appareil est au repos, nous mesurons l'attraction terrestre.

Visualiser la verticale

Améliorons notre application *Accelero* pour visualiser la verticale par un segment de droite représentant la projection de la force d'attraction sur l'écran. Nous aurons besoin de définir une vue spécifique `VerticalView` dont la largeur sera de préférence le double de la hauteur.

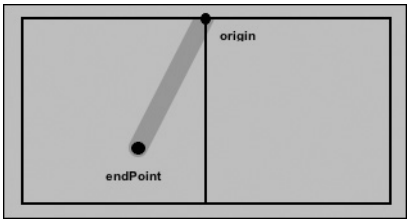


Figure 12.3 : Vue VerticalView

Modifier le contrôleur de vue

1 Modifiez le fichier *AcceleroViewController.h* pour y définir un nouvel outlet de type `VerticalView` :

```
#import <UIKit/UIKit.h>
@class VerticalView;
@interface AcceleroViewController : UIViewController
                              <UIAccelerometerDelegate> {
   IBOutlet UILabel * xLabel;
   IBOutlet UILabel * yLabel;
   IBOutlet UILabel * zLabel;
   IBOutlet VerticalView * vert;
}
@property(nonatomic,retain) UILabel * xLabel;
@property(nonatomic,retain) UILabel * yLabel;
@property(nonatomic,retain) UILabel * zLabel;
@property(nonatomic,retain) VerticalView * vert;
@end
```

Nous doterons notre vue `VerticalView` d'une propriété `vertLine` de type `GCSize` qui contiendra les abscisses et ordonnées de l'accélération.

2 Modifiez le fichier *AcceleroViewController.m* pour utiliser cette propriété :

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
                        didAccelerate:(UIAcceleration *)acc
{
   xLabel.text = [NSString stringWithFormat:@"%f", acc.x];
   yLabel.text = [NSString stringWithFormat:@"%f", acc.y];
   zLabel.text = [NSString stringWithFormat:@"%f", acc.z];
   vert.vertLine = CGSizeMake(acc.x,acc.y);
}
```

3 Dans ce même fichier, importez le fichier *VerticalView.h*, synthétisez les accesseurs de la propriété `vert` et libérez cette propriété.

Le type GCSize et la fonction GCSizeMake sont détaillés au chapitre *Dessins et animations*.

Créer la vue VerticalView

Cette classe met en œuvre les techniques vues au chapitre *Dessins et animations*. Elle doit assurer que :

- L'origine du segment se situe au milieu du bord supérieur de la vue. Nous avons besoin d'une variable d'instance `origin` de type `CGPoint` pour conserver ce point.
- La longueur maximale du segment est la hauteur, ou la demi-largeur, de la vue. Nous définirons une variable d'instance `scale` de type `CGFloat` pour conserver cette longueur maximale.
- Le segment est toujours visible, quelle que soit l'orientation de l'appareil. Une variable d'instance `endPoint` de type `CGPoint` sera évaluée à chaque modification de la propriété `vertLine`.

Procédez ainsi :

1 Sous XCode, créez une nouvelle classe `VerticalView` dérivée de `UIView`. Déclarez son interface :

```
#import <UIKit/UIKit.h>
@interface VerticalView : UIView {
   CGFloat scale;
   CGPoint origin;
   CGSize vertLine;
   CGPoint endPoint;
}
@property(nonatomic,assign) CGSize vertLine;
@end
```

Les variables d'instance `scale` et `origin` doivent être évaluées lorsque la vue est insérée dans la hiérarchie des vues et que sa taille est définie.

2 Ajoutez la méthode `layoutSubviews` dans le fichier *VerticalView.m* :

```
- (void) layoutSubviews
{
   CGFloat height = self.bounds.size.height;
   CGFloat width = self.bounds.size.width;
   if (height<2.*width)
      scale = height;
   else
      scale = width/2.;
   origin = CGPointMake(width/2.,0.);
}
```

Lorsque la propriété `vertLine` est modifiée, la variable d'instance `endPoint` doit être évaluée et la vue redessinée.

3 Définissez la méthode `setVertLine:` dans le fichier *VerticalView.m* :

```
- (void)setVertLine:(CGSize)line{
   vertLine = line;
   if (line.height<0) {
      endPoint.x = line.width*scale+origin.x;
      endPoint.y = -line.height*scale+origin.y;
   } else {
      endPoint.x = -line.width*scale+origin.x;
      endPoint.y = line.height*scale+origin.y;
   }
   [self setNeedsDisplay];
}
```

4 Écrivez la méthode `drawRect:` qui ne présente pas de difficulté particulière :

```
- (void)drawRect:(CGRect)rect {
   CGContextRef context = UIGraphicsGetCurrentContext();
   CGContextSetRGBStrokeColor(context, 1., 0.5, 0., 1.0);
   CGContextSetLineWidth(context, 15.);
   CGContextSetLineCap(context, kCGLineCapRound);
   CGPoint segment[2] = {origin,endPoint};
   CGContextStrokeLineSegments(context, segment, 2);
}
```

Finaliser et tester l'application

N'oubliez pas de synthétiser les accesseurs de la propriété `vertLine`.

1 Ouvrez le fichier *AcceleroViewController.xib* pour ajouter une vue sur l'interface utilisateur. Définissez sa classe : `VerticalView`. Liez cette vue à l'outlet `vert` du propriétaire du fichier.

2 Construisez l'application pour la tester sur un appareil réel. Nous visualisons maintenant la verticale.

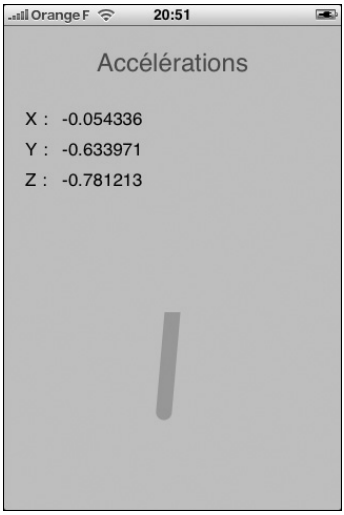


Figure 12.4 : Visualisation de la verticale

Même l'appareil au repos, la verticale gigote sans arrêt. Les accéléromètres sont sensibles et perçoivent les vibrations de quelques milli-g. Si l'on souhaite une verticale plus stable, ou connaître la position de l'appareil en faisant abstraction des vibrations, il faut filtrer les données mesurées.

Filtrer les données

Nous allons ajouter un filtre paramétrable à notre application pour expérimenter un filtre numérique et en illustrer les effets.

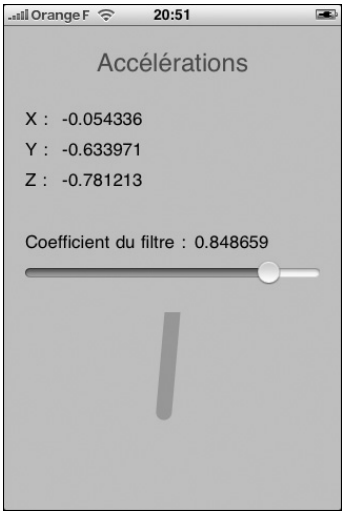


Figure 12.5 : Application avec un filtre numérique

1 Ouvrez le fichier *AcceleroViewController.m* et modifiez la méthode `-accelerometer:didAccelerate:` :

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
              didAccelerate:(UIAcceleration *)acc
{
   static UIAccelerationValue accelX=0.,accelY=0.,accelZ=0.;
   accelX = self.coef*acc.x + (1.-self.coef)*accelX;
   accelY = self.coef*acc.y + (1.-self.coef)*accelY;
   accelZ = self.coef*acc.z + (1.-self.coef)*accelZ;
   xLabel.text = [NSString stringWithFormat:@"%f",accelX];
   yLabel.text = [NSString stringWithFormat:@"%f",accelY];
   zLabel.text = [NSString stringWithFormat:@"%f",accelZ];
   vert.vertLine = CGSizeMake(accelX,accelY);
}
```

La méthode utilise une propriété `coef` et trois variables statiques qui contiennent le résultat du calcul des composantes de l'accélération. Si par exemple la propriété `coef` vaut 10 %, le résultat du calcul est la somme de 10 % de la nouvelle mesure et 90 % du calcul précédent. Ainsi les tremblements seront atténués.

Nous utiliserons un ascenseur pour modifier la valeur du coefficient, ceci facilitera l'expérimentation du filtre et nous aidera à comprendre son effet.

2 Ajoutez une méthode `-changeCoef:` dans le fichier *AcceleroView-Controller.m*. Elle sera l'action déclenchée par l'événement **Change Value** de l'ascenseur.

```
- (void)changeCoef:(id)sender{
   self.coef = [[sender valueForKey:@"value"] floatValue];
   self.coefLabel.text =
             [NSString stringWithFormat:@"%f",self.coef];
}
```

3 Synthétisez les accesseurs pour les nouvelles propriétés et initialisez-les dans la méthode `viewDidLoad` :

```
@synthesize xLabel, yLabel, zLabel, coef, coefLabel, vert;
- (void)viewDidLoad {
   [super viewDidLoad];
   UIAccelerometer *accelerometer =
                  [UIAccelerometer sharedAccelerometer];
   accelerometer.updateInterval = 0.1;
   accelerometer.delegate = self;
   self.coef = 0.5;
   self.coefLabel.text = @"0.5";
}
```

4 Modifiez le fichier *AcceleroViewController.h* pour y déclarer les nouvelles propriétés et la nouvelle méthode de la classe :

```
#import <UIKit/UIKit.h>
@class VerticalView;
@interface AcceleroViewController : UIViewController
                                <UIAccelerometerDelegate> {
   IBOutlet UILabel * xLabel;
   IBOutlet UILabel * yLabel;
   IBOutlet UILabel * zLabel;
   IBOutlet UILabel * coefLabel;
   float coef;
   IBOutlet VerticalView * vert;
}
@property(nonatomic,retain) UILabel * xLabel;
@property(nonatomic,retain) UILabel * yLabel;
@property(nonatomic,retain) UILabel * zLabel;
@property(nonatomic,retain) UILabel * coefLabel;
@property(nonatomic,assign) float coef;
@property(nonatomic,retain) VerticalView * vert;
- (IBAction)changeCoef:(id)sender;
@end
```

5 Modifiez l'interface utilisateur en ouvrant le fichier *AcceleroView-Controller.xib*. Ajoutez un ascenseur et un label. Vérifiez que les valeurs de l'ascenseur sont comprises entre `0` et `1`. Liez ces éléments aux nouveaux *outlets* du propriétaire du fichier et liez l'événement **Change Value** de l'ascenseur à l'action `changeCoef:` du propriétaire du fichier.

Vous pouvez maintenant construire l'application et la tester sur votre appareil. Une faible valeur du coefficient, environ `0,1`, diminue fortement les perturbations ; le repère graphique de la verticale ne tremble plus, et les deux premières décimales des composantes de l'accélération sont stables. Mais cette amélioration présente un inconvénient : lors d'un changement d'orientation de l'appareil, il faut plusieurs secondes pour que le segment représentant la verticale rallie sa nouvelle position.

Apple recommande d'utiliser un coefficient de `0,1`. Si vous trouvez qu'avec cette valeur, l'accélération calculée n'est pas assez réactive, vous pouvez augmenter la fréquence de mesure en modifiant la valeur de la propriété `updateInterval` de l'accéléromètre. Les valeurs recommandées par Apple sont :

- entre `0,05` et `0,1` pour connaître l'orientation de l'appareil ;

- entre `0,015` et `0,03` pour utiliser les mouvements de l'appareil dans des jeux ;
- entre `0,01` et `0,015` pour mesurer des mouvements très rapides, `0,01` est la valeur la plus faible admissible.

12.2. Déterminer les mouvements de l'appareil

Le filtre passe-bas que nous venons d'expérimenter permet de connaître l'orientation de l'appareil. Dans certains cas, on peut souhaiter que notre application réagisse aux *changements* d'orientation. Physiquement, l'application doit rechercher les modifications d'accélération, via un filtre passe-haut :

```
accelX = acc.x - ((acc.x * self.coef) +
                              (accelX* (1.0 - self.coef)));
accelY = acc.y - ((acc.y * self.coef) +
                              (accelY* (1.0 - self.coef)));
accelZ = acc.z - ((acc.z * self.coef) +
                              (accelY* (1.0 - self.coef)));
```

Essayez ce filtre dans l'application *Accelero*.

Souvenez-vous, vous avez également la possibilité de détecter les secousses en utilisant les événements gérés par l'application. Reportez-vous pour cela au chapitre *Tapes, touches et gestes*.

12.3. Connaître l'orientation de l'appareil

Nous savons maintenant utiliser les accéléromètres afin de connaître la position relative de l'appareil par rapport à la verticale, ou pour en déterminer précisément les mouvements. Dans la plupart des cas cependant, l'application n'a pas besoin de ce niveau de détail, elle a simplement besoin de connaître l'orientation de l'écran afin d'afficher les vues dans le "bon" sens.

Figure 12.6 : Orientation Portrait de l'appareil

Figure 12.7 : Orientation Paysage de l'appareil

Retour sur la classe UIDevice

La classe `UIDevice` dispose d'une propriété `orientation` à lecture seule de type `UIDeviceOrientation`. La valeur de cette propriété donne l'orientation de l'appareil :

- `UIDeviceOrientationUnknown`, l'orientation ne peut être déterminée.
- `UIDeviceOrientationPortrait`, orientation Portrait bouton principal en bas.
- `UIDeviceOrientationPortraitUpsideDown`, orientation Portrait bouton principal en haut.
- `UIDeviceOrientationLandscapeLeft`, orientation Paysage bouton principal à droite.
- `UIDeviceOrientationLandscapeRight`, orientation Paysage bouton principal à gauche.
- `UIDeviceOrientationFaceUp`, l'appareil est parallèle au sol, l'écran vers le haut.
- `UIDeviceOrientationFaceDown`, l'appareil est parallèle au sol, l'écran vers le bas.

Nous avons déjà rencontré la classe UIDevice au chapitre *Dessins et animations*.

L'obtention de l'orientation est très simple. Il faut privilégier l'instance unique de la classe `UIDevice`, puis activer l'entretien de sa propriété `orientation` avant d'obtenir sa valeur :

```
UIDevice *device = [UIDevice currentDevice];
[device beginGeneratingDeviceOrientationNotifications];
UIDeviceOrientation *orientation = device.orientation;
```

Afin d'économiser la batterie, il est recommandé de désactiver les accéléromètres lorsque l'application n'a pas besoin de connaître l'orientation de l'appareil :

```
[device endGeneratingDeviceOrientationNotifications];
```

S'abonner aux changements d'orientation

Interroger l'instance unique de la classe `UIDevice` est un moyen de connaître l'orientation de l'appareil. Il est parfois plus pratique d'être informé lors d'un changement d'orientation. Il faut pour cela s'abonner aux notifications `UIDeviceOrientationDidChangeNotification` émises par cette instance.

Dans ce cas aussi, il faut activer l'émission des notifications par le message `beginGeneratingDeviceOrientationNotifications` sur l'instance unique de la classe `UIDevice`.

Reportez-vous au chapitre *Persistance des données* si vous avez oublié comment vous abonner à une notification.

Orienter automatiquement les vues

Une fonctionnalité très attrayante des logiciels sur iPhone est de voir l'interface utilisateur suivre l'orientation de l'appareil. Cette fonctionnalité est facile à obtenir ; nous allons la réaliser avec notre application *Accelero*.

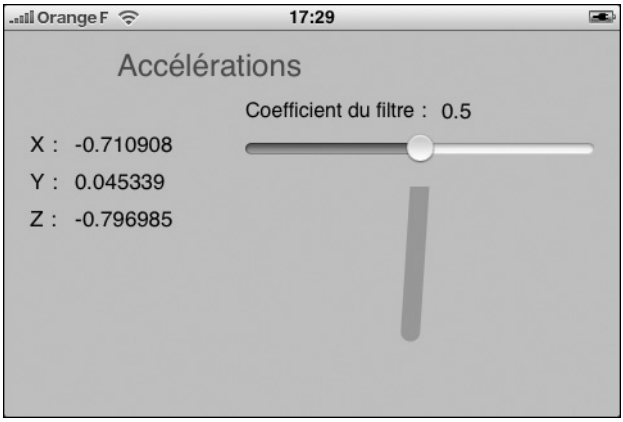


Figure 12.8 : Accelero en mode paysage

Obligatoire sur l'iPad

Sauf cas exceptionnel, vos applications pour iPad devront prendre en charge l'orientation automatique des vues.

La prise en charge de l'orientation automatique nécessite deux étapes :

- Le contrôleur indique les orientations prises en charge.
- La taille et la position de chaque vue doivent être définies pour les orientations Portrait et Paysage.

Autorotations prises en charge

Modifiez la méthode `–shouldAutorotateToInterfaceOrientation:` dans le fichier *AcceleroViewController.m* :

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
            (UIInterfaceOrientation)interfaceOrientation {
   return YES;
}
```

Cette méthode est appelée à chaque changement d'orientation de l'interface. Elle doit retourner `YES` si l'*autorotation* est prise en charge pour l'orientation passée en paramètre.

Par défaut, la méthode définie dans la classe `UIViewController` répond `YES` pour l'orientation Portrait `UIDeviceOrientationPortrait`, et `NO` pour les trois autres orientations de l'interface. Nous redéfinissons cette méthode dans la classe `AcceleroViewController` pour qu'elle réponde `YES` pour toutes les orientations ; l'interface utilisateur sera toujours orientée correctement quelle que soit la position de l'appareil.

Orientation de l'appareil et de l'interface

Seules les quatre valeurs `UIDeviceOrientationPortrait`, `UIDeviceOrientationPortraitUpsideDown`, `UIDeviceOrientationLandscapeLeft` et `UIDeviceOrientationLandscapeRight` peuvent être utilisées pour l'orientation de l'interface utilisateur.

Disposition des vues

1. Ouvrez le fichier *AcceleroViewController.xib* sous Interface Builder. Le bouton en haut à droite de la fenêtre de la vue principale permet de basculer la représentation de l'interface entre le mode Portrait et le mode Paysage.

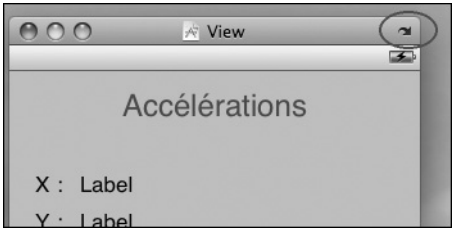


Figure 12.9 : Basculer la vue en mode Paysage

2. Cliquez sur le bouton de basculement pour voir ce que donnerait l'interface en mode Paysage : le résultat n'est pas très concluant. Les vues restent à gauche de l'écran, la partie droite n'est pas utilisée, et les vues du bas sortent de l'écran. Nous aimerions

qu'en mode Paysage, les champs de texte indiquant les valeurs des composantes de l'accélération restent à gauche de l'écran, et que la vue graphique ainsi que le réglage du filtre passent à droite.

3 Revenez en mode Portrait et sélectionnez les vues que vous souhaitez faire glisser sur la droite lorsque la vue principale basculera en mode Paysage.

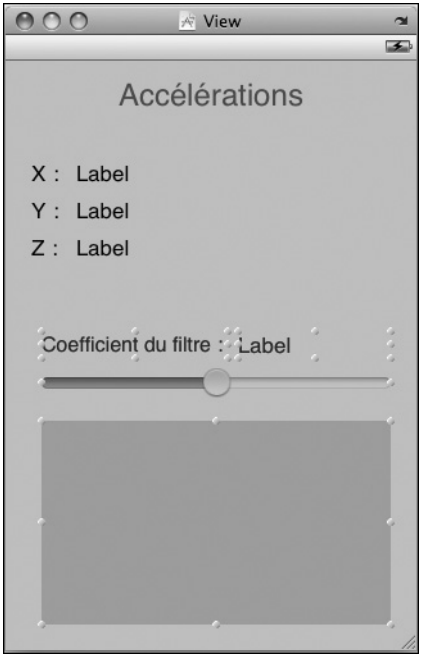


Figure 12.10 : Sélection suite au glisser à droite en mode Paysage

4 Affichez l'inspecteur de taille, commande **Size Inspector** du menu **Tools** (⌘+3), et réglez les paramètres **Autosizing** de sorte que ces vues soient attachées par leurs bords droit et bas plutôt que par leurs bords haut et gauche, ce qui est le défaut.

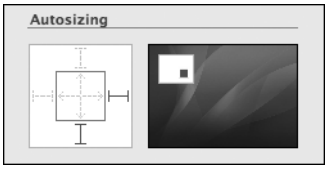


Figure 12.11 : Attache des vues par leurs bords droit et bas

5 Basculez la vue principale en mode Paysage ; les vues du bas glissent vers la droite de l'écran. Au besoin, ajustez la position et la taille de chacune des vues pour que leur disposition soit correcte aussi bien en mode Paysage qu'en mode Portrait.

Les paramètres **Autosizing** fonctionnent de la façon suivante :

- Il existe un attachement pour chacun des quatre bords de la vue, représenté par un trait plein ou pointillé :
  - Un trait plein signifie que la distance est fixe entre le bord de la vue sélectionné et le bord correspondant de sa super-vue.
  - Un trait pointillé signifie qu'il n'y a pas de contrainte de distance entre le bord de la vue sélectionné et le bord correspondant de sa super-vue.
- Il existe deux indicateurs de dimension, pour la hauteur et la largeur de la vue, représentés par un trait plein ou pointillé :
  - Un trait pointillé signifie que la dimension est fixe.
  - Un trait plein signifie que la dimension est variable en fonction de la dimension correspondante de sa super-vue.
- En principe, si les attachements de bords opposés sont tous les deux en traits pleins, il faut que la dimension entre ces bords soit variable, donc également en trait plein.

Pour que ces paramètres soient pris en compte, la case *Autoresize Subviews* doit être cochée dans l'inspecteur des attributs de la super-vue, en l'occurrence la vue principale, ce qui est le défaut.

6 Reconstruisez l'application sous XCode et vérifiez que l'interface utilisateur suit l'orientation de l'appareil.

Challenge

En testant l'application *Accelero*, vous remarquez que l'indicateur graphique de verticale fonctionne uniquement en mode Portrait. Votre challenge sera de modifier l'application *Accelero*, et principalement la classe `VerticalView` pour qu'elle fonctionne pour les quatre orientations de l'interface.

12.4. Checklist

Nous avons appris à mettre en œuvre les accéléromètres pour déterminer la position de l'appareil ou pour en détecter les mouvements, et les classes et types concernés :

- `UIAccelerometer` ;
- `UIAccelerometerDelegate` ;
- `UIAcceleration` ;

- `UIAccelerationValue`.

Nous avons vu comment utiliser la classe `UIDevice`, et son instance unique, pour connaître l'orientation de l'appareil et programmer les notifications des changements d'orientation.

Nous savons doter nos applications de la fonctionnalité d'*autorotation* de l'interface utilisateur.

SPÉCIFICITÉS DE L'IPAD

Vous disposez maintenant de connaissances suffisantes pour développer vos propres applications, qui peuvent s'exécuter de façon identique sur iPhone, iPod Touch et iPad.

Certains ne voient dans l'iPad qu'un gros iPhone simplement moins pratique à glisser dans la poche de son veston. Il est vrai que ces appareils mettent en œuvre exactement les mêmes technologies, avec pour l'iPad un écran plus large et une puissance et une autonomie accrues. L'iPad ouvre un univers d'applications qui reste à défricher ; il lance une révolution de l'informatique mobile comparable à ce que la musique a connu avec le baladeur MP3.

À la fin de ce chapitre, vous connaîtrez les techniques spécifiques à l'iPad qui vous permettront de créer vos applications, et pourquoi pas d'être un acteur de la révolution en marche.

13.1. Un SDK, deux cibles

La version 3.2 du SDK de l'iPhone OS permet de développer des applications pour des versions très anciennes ; par exemple la version 2.0 distribuée au printemps 2008. En principe, une application est produite pour la *dernière version* d'OS disponible, mais il peut être intéressant d'accroître le nombre de ses clients en étendant la compatibilité de l'application avec des versions d'OS antérieures.

La particularité introduite par l'iPad est la coexistence de deux "dernières" versions :

- La *version 3.1.3* est la dernière version pour iPhone (et iPod Touch).
- La *version 3.2* est la dernière version pour iPad (c'est aussi la première).

Tous les chapitres précédents portent sur l'utilisation du SDK 3.2 pour la production d'applications destinées à la version 3.1.3 : les applications iPhone capables de tourner sur iPad. Le présent chapitre porte sur les spécificités de la version 3.2 : sur les applications spécifiquement dédiées à l'iPad.

Choisir sa cible de déploiement

Le plus simple pour choisir la *cible de déploiement* de l'application (iPhone ou iPad) est de le préciser à la création du projet.

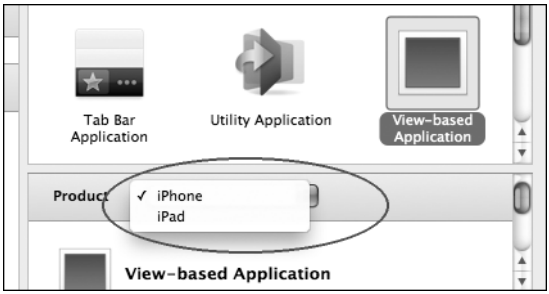


Figure 13.1 : Définition de la cible de déploiement du projet

Si l'on veut redéfinir la cible de déploiement sur un projet préexistant, on peut le faire dans la fenêtre d'information de la cible du projet. Le tableau ci-après résume les valeurs des paramètres de build en fonction de la cible de déploiement.

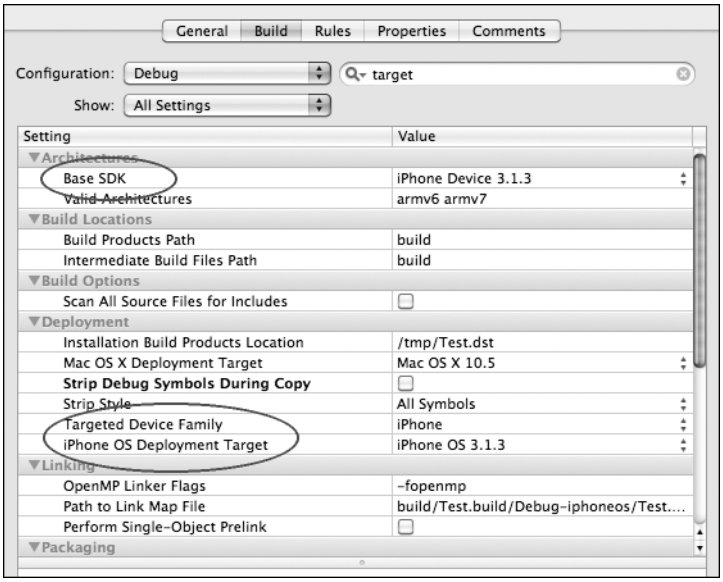


Figure 13.2 : Paramètres de la cible de déploiement

Tableau 13.1 : Valeurs des paramètres de build pour fixer la cible de déploiement

Paramètre	Valeur pour iPhone	Valeur pour iPad	Valeur universelle
iPhone OS deployment target	iPhone OS 3.1.3	iPhone OS 3.2	iPhone OS 3.1.3
Targeted Device Family	iPhone	iPad	iPhone/iPad
Base SDK	iPhone Device 3.1.3	iPhone Device 3.2	iPhone Device 3.2

La dernière colonne du tableau est intitulée *Valeur universelle*. Une application iPhone OS est dite *universelle* lorsqu'elle s'adapte à l'appareil sur lequel elle s'exécute : iPhone ou iPad.

Créer une application universelle

Le plus simple pour créer une application universelle est de partir d'un projet iPhone.

1. Ouvrez votre projet iPhone sous XCode et ouvrez le groupe **Targets** dans la partie gauche de la fenêtre, **Groups & Files**.
2. Cliquez du bouton droit sur la cible désirée pour ouvrir le menu contextuel et sélectionnez la commande **Upgrade Current Target for iPad**.... Une boîte de dialogue s'ouvre pour vous permettre d'indiquer si vous souhaitez créer une application universelle unique (**One Universal application**) ou deux applications spécifiques (**Two device-specific applications**).

Dans les deux cas, XCode va générer des fichiers *NIB* pour iPad en dupliquant les fichiers *NIB* du projet dans un dossier et dans un groupe *Resources-iPad*. Ainsi vous pouvez modifier ces fichiers pour particulariser l'interface utilisateur à chaque type d'appareil.

Dans le cas d'une *application universelle*, il vous faudra écrire du code qui sache s'adapter à l'appareil. Dans le second cas, vous écrirez deux applications dans le même projet.

13.2. Nouveautés de l'interface visuelle

La surface de l'écran plus de quatre fois plus grande sur l'iPad ouvre des possibilités supplémentaires que nous allons examiner dans cette section. Il y sera question notamment des *vues contextuelles* (*popover*) et des *vues scindées* (*splitview*), deux des principales nouveautés de la version 3.2.

Recommandations générales

Autorotation de l'interface

Commençons par une nouveauté qui n'en est pas une. Nous avons vu au chapitre précédent comment inclure l'*autorotation* de l'interface dans une application iPhone.

Reportez-vous au chapitre *Accéléromètres* si vous avez oublié comment gérer l'autorotation.

La "nouveauté" est que cette fonctionnalité doit être systématiquement incluse dans les applications pour iPad afin de respecter les directives d'Apple concernant l'interface utilisateur. Vous pouvez consulter ces directives dans le document *iPad Human Interface Guidelines* d'Apple.

Hiérarchisation des données

La petite taille de l'écran de l'iPhone nécessite un design particulier de l'interface utilisateur, qui impose généralement de structurer les données hiérarchiquement. Cette hiérarchisation est à éviter sur iPad dont l'écran est plus grand que celui d'un iPhone.

Il faudra donc éviter à l'utilisateur d'avoir à naviguer entre plusieurs écrans, lui donner le maximum d'informations et de possibilité sur un écran. L'application de type navigation, très courante sur iPhone, est remplacée par l'application de type *Vues Scindées* (*splitview*) spécifique à l'iPad.

Figure 13.3 : Application de type vues scindées

On peut également atteindre ce résultat en regroupant le contenu de plusieurs écrans d'une application iPhone sur un seul écran dans l'application équivalente sur iPad. Pensez aussi, lors du design de

l'interface utilisateur, à présenter les informations détaillées ou les formulaires de saisie dans une *vue contextuelle* (*popover*).

Figure 13.4 : Utilisation d'une vue contextuelle

Vues modales

Il y a deux différences dans la façon dont sont utilisées les *vues modales* entre l'iPhone et l'iPad :

- Sur iPhone, une vue modale occupe tout l'écran, ce n'est pas forcément le cas sur iPad.
- Il n'y a pas de limitation à l'emploi des vues modales sur iPhone ; les directives d'Apple en limitent l'usage sur iPad.

Présentation à l'écran

La propriété `modalPresentationStyle` de la classe `UIViewController` (dans la version 3.2 du SDK) permet de spécifier la façon dont la vue modale doit être présentée. Elle peut prendre les valeurs suivantes :

- `UIModalPresentationFullScreen`, valeur par défaut, présentation de la vue modale en plein écran (comme sur l'iPhone) ;
- `UIModalPresentationPageSheet`, la vue modale occupe toute la hauteur de l'écran, mais sa largeur est celle de la plus petite dimension de l'écran (768 pixels sur iPad) ;
- `UIModalPresentationFormSheet`, la vue modale est plus petite que l'écran et centrée sur celui-ci ;
- `UIModalPresentationCurrentContext`, la vue modale utilise le même style que sa vue parente.

Lors de l'affichage d'une vue modale, les parties de l'écran non recouvertes par la vue sont grisées et inaccessibles.

Usage des vues modales

Sur iPad, les *vues modales* doivent être utilisées *exclusivement* lorsque la tâche en cours requiert l'intervention de l'utilisateur.

Pour tous les autres usages, il est préférable d'utiliser les *vues contextuelles* (*popover*), nouveauté de la version 3.2 :

- présenter une liste de sélections ou d'actions ;
- présenter des informations détaillées ;
- présenter une boîte à outils ou des options de configuration.

La différence essentielle entre une *vue modale* et une *vue contextuelle* porte sur le comportement lorsque l'utilisateur touche l'extérieur de la vue :

- La première interdit toute action à l'extérieur de la vue, qui est grisée.
- La seconde est refermée.

Vues contextuelles

Une *vue contextuelle* (*popover*) est présentée à l'écran avec une flèche pointant vers un bouton ou une autre vue.

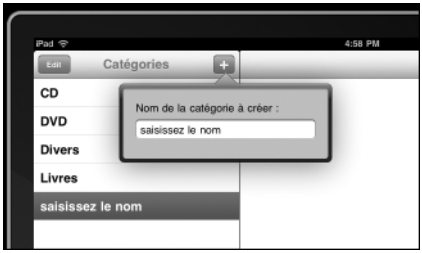


Figure 13.5 : Exemple de vue contextuelle

Préparer une vue contextuelle

Deux contrôleurs de vue sont mis en jeu :

- le contrôleur de vue contextuel, instance de la classe `UIPopoverController`, chargé de gérer le *contenant* de la vue contextuelle ;
- un contrôleur de vue quelconque, dont la classe dérive de `UIViewController`, chargé de gérer le *contenu* de la vue contextuelle.

Le contrôleur du contenu est préparé de la même façon, quelle que soit la manière dont il sera affiché : dans une vue contextuelle, une vue modale, une pile de navigation, etc. Une spécificité toutefois, la taille du contenu de la vue contextuelle doit être spécifiée dans la propriété `contentSizeForViewInPopover` de son contrôleur, par exemple :

```
contentViewController.contentSizeForViewInPopover =
                                    CGSizeMake(320.0, 110.0);
```

Limitation de la taille

La largeur du contenu d'une vue contextuelle doit être comprise entre 320 et 600 pixels. Sa hauteur est libre.

Afficher une vue contextuelle

Le contrôleur du contenu est associé au contrôleur du contenant à la création de ce dernier :

```
UIPopoverController* aPopover =
             [[UIPopoverController alloc]
    initWithContentViewController:contentViewController];
```

Lors de sa présentation à l'écran, il faut préciser l'objet sur lequel doit pointer la flèche de la vue contextuelle ainsi que les directions autorisées. On utilise deux méthodes différentes suivant que la vue est associée à un bouton de barre d'outils (`UIBarButtonItem`) ou plus généralement à un rectangle dans une vue :

- `-presentPopoverFromBarButtonItem:permittedArrowDirections:animated:` pour associer la vue à un bouton ;
- `-presentPopoverFromRect:inView:permittedArrowDirections:animated:` dans le cas général.

Le paramètre `permittedArrowDirections` permet de préciser quelles directions sont autorisées pour la flèche de la vue contextuelle :

- `UIPopoverArrowDirectionUp` autorise une flèche vers le haut.
- `UIPopoverArrowDirectionDown` autorise une flèche vers le bas.
- `UIPopoverArrowDirectionLeft` autorise une flèche vers la gauche.
- `UIPopoverArrowDirectionRight` autorise une flèche vers la droite.

Ces valeurs peuvent être combinées par l'opérateur `|` afin d'autoriser plusieurs directions. La valeur `UIPopoverArrowDirectionAny` peut également être utilisée pour autoriser la flèche dans toutes les directions.

Refermer une vue contextuelle

Lorsque l'utilisateur touche l'extérieur d'une vue contextuelle, cette dernière est automatiquement refermée. Le délégué du contrôleur de la vue contextuelle en est informé ; il peut bloquer la fermeture au besoin.

On peut également refermer la vue contextuelle en transmettant le message `–dismissPopoverAnimated:` à son contrôleur.

Délégué de vue contextuelle

Le délégué du contrôleur de vue contextuelle, non obligatoire, répond au protocole `UIPopoverControllerDelegate`. Il reçoit le message `–popoverControllerShouldDismissPopover:` lorsque l'utilisateur touche l'extérieur de la vue. Il doit retourner `YES` pour autoriser sa fermeture (valeur par défaut).

Lorsque la vue contextuelle est effectivement refermée, son délégué reçoit le message

`–popoverControllerDidDismissPopover:`.

Vues scindées

Les *vues scindées* (*splitview*) concernent un mode d'utilisation de l'écran de l'iPad :

- En mode Paysage, l'écran est scindé en deux, une vue est affichée dans la partie gauche, d'une largeur de 320 pixels, et une seconde vue occupe l'autre partie de l'écran.
- En mode Portrait, seule la partie droite est affichée, la partie gauche peut être visualisée dans une vue contextuelle.

Figure 13.6 : Vue scindée en mode Paysage

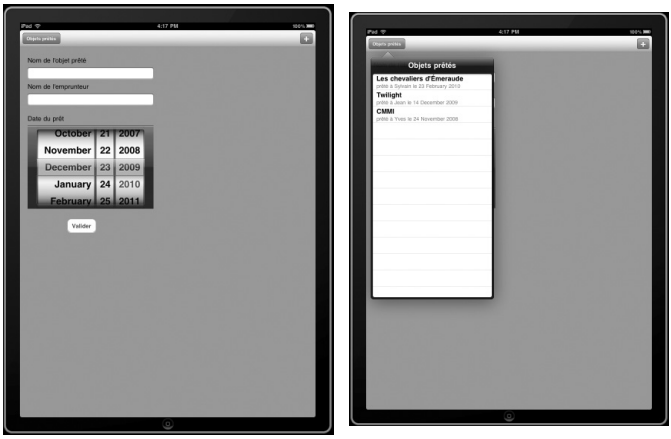


Figure 13.7 : Vue scindée en mode Portrait

Contrôleur de vue scindée

Le contrôleur de vue scindée, instance de la classe `UISplitViewController`, doit être le contrôleur racine de la fenêtre. Il ne peut être inclus dans un autre contrôleur de vue. Il prend en charge le comportement de la vue lors des changements d'orientation de l'interface utilisateur.

Cette classe présente seulement deux propriétés :

- `viewControllers`, tableau contenant les deux contrôleurs de vue associés à la vue scindée, dans l'ordre la vue de gauche puis la vue de droite ;
- `delegate`, le délégué de vue scindée qui est informé des changements d'orientation de l'interface utilisateur.

Délégué du contrôleur de vue scindée

Le délégué du contrôleur de vue scindée répond au protocole `UISplitViewControllerDelegate`. Il reçoit un message `-splitViewController:willHideViewController:withBarButtonItem:forPopoverController:` lorsque l'appareil passe en mode Portrait et que la vue de gauche va être masquée. Ce message contient notamment un bouton de barre d'outils préparé par le contrôleur de vue scindée. Il appartient au délégué d'afficher ce bouton qui permettra à l'utilisateur de voir le contenu de la vue de gauche dans une vue contextuelle. Par exemple, pour afficher ce bouton à gauche d'une barre d'outils tout en lui donnant un titre :

```
- (void)splitViewController:(UISplitViewController*)svc
willHideViewController:(UIViewController *)aViewController
         withBarButtonItem:(UIBarButtonItem*)barButtonItem
          forPopoverController: (UIPopoverController*)pc {
   barButtonItem.title = @"Objets prêtés";
   NSMutableArray *items = [[toolbar items] mutableCopy];
   [items insertObject:barButtonItem atIndex:0];
   [toolbar setItems:items animated:YES];
   [items release];
}
```

À l'inverse, le message `-splitViewController:willShowViewController:invalidatingBarButtonItem:` est reçu par le délégué du contrôleur de vue scindée lorsque l'appareil passe en mode Paysage. Il faut alors retirer le bouton précédent :

```
- (void)splitViewController: (UISplitViewController*)svc
 willShowViewController:(UIViewController *)aViewController
invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem {
   NSMutableArray *items = [[toolbar items] mutableCopy];
   [items removeObjectAtIndex:0];
   [toolbar setItems:items animated:YES];
   [items release];
}
```

Le dernier message susceptible d'être reçu par le délégué du contrôleur de vue scindée est `-splitViewController:popoverController:willPresentViewController:`, en mode Portrait lorsque la vue contextuelle contenant la vue de gauche est sur le point d'être affichée.

Application de type Vue scindée

Le plus simple pour utiliser les vues scindées est de créer sous XCode une application de type *Vue scindée*.

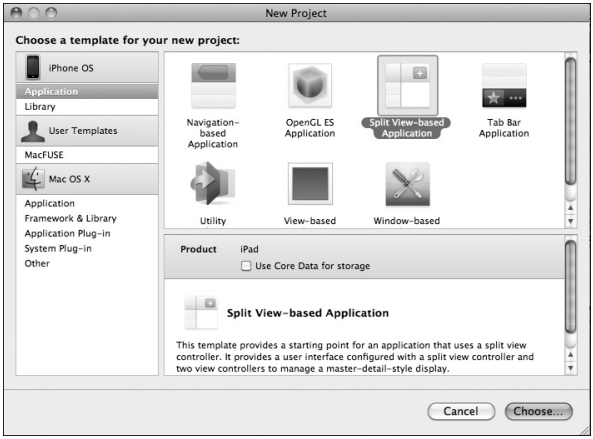


Figure 13.8 : Création d'une application de type Vue scindée

Nous disposons alors d'un fichier *MainWindow.xib* appartenant à l'application et qui contient :

- le délégué d'application, classique ;
- un contrôleur de vue scindé dont :
  - la vue de gauche est un contrôleur de navigation dont la vue principale est un contrôleur de vue en table `RootViewController` ;
  - la vue de droite est décrite dans le fichier *DetailView.xib* et son contrôleur est `DetailViewController`.

La vue détaillée préparée dans le fichier *DetailView.xib* offre une barre d'outils qui contiendra le bouton permettant de voir la vue de gauche en mode Portrait. Le contrôleur de la vue détaillée `DetailView Controller` est le délégué de la vue scindée ; c'est lui qui gère le bouton de visualisation de la vue gauche en mode Portrait.

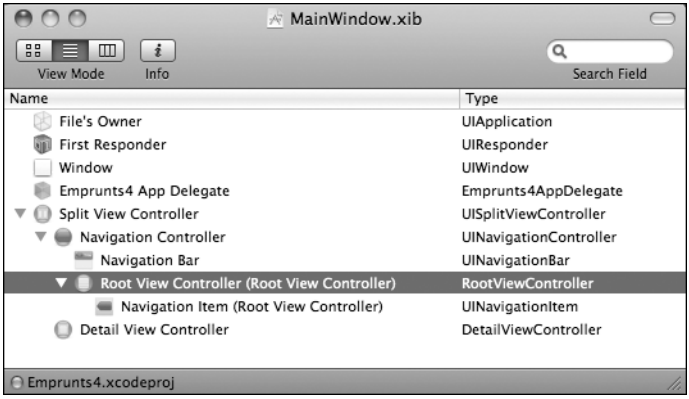


Figure 13.9 : Contenu du fichier MainWindow.xib

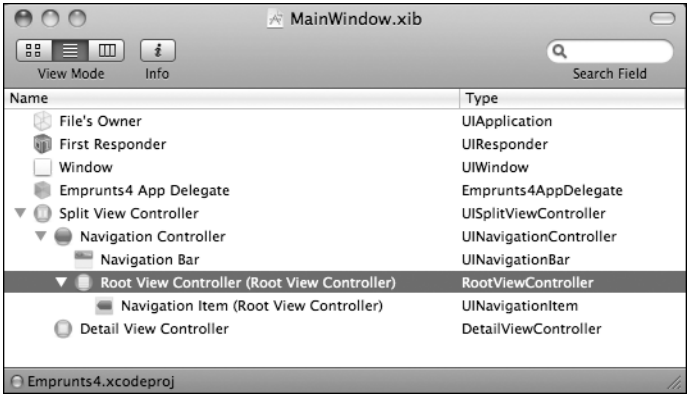


Figure 13.10 : Contenu du fichier DetailView.xib

Challenge

Réécrivez l'application Emprunts sous forme de vue scindée pour iPad. Souvenez-vous qu'il n'est pas utile de hiérarchiser les données pour les manipuler sur iPad.

13.3. Reconnaissance des gestes

Vous savez déjà comment interpréter les événements reçus par une vue pour reconnaître les gestes effectués par l'utilisateur.

Reportez-vous au chapitre *Tapes, touches et gestes* si vous avez besoin de vous rafraîchir la mémoire au sujet de la gestion des événements et la reconnaissance des gestes sur iPhone.

La version 3.2 permet heureusement de simplifier grandement la reconnaissance des gestes. Il suffit maintenant de programmer une *cible-action* sur un ou plusieurs *analyseurs de geste*, puis d'attacher ces dernier à la vue. C'est une bonne nouvelle, vous n'avez plus besoin de dériver la classe `UIView` ni d'écrire le code pour la reconnaissance des gestes, sauf bien sûr si vous voulez développer vos propres analyseurs.

Gestes de base

Un *analyseur de geste* est une instance d'une classe dérivée de `UIGestureRecognizer`. En effet, cette classe est une *classe abstraite* qui implémente les mécanismes fondamentaux pour la reconnaissance des gestes. Chaque geste particulier est reconnu par une *classe concrète* qui dérive de `UIGestureRecognizer`.

Un analyseur est programmé pour reconnaître un geste particulier. Lorsque ce geste est reconnu, l'analyseur délivre une notification par le mécanisme *cible-action* ; plusieurs cibles-actions peuvent être programmées sur chaque analyseur.

Modes de fonctionnement

Certains analyseurs fonctionnent de façon discrète ; une notification vers chaque cible est émise lorsque le geste est reconnu, puis l'analyseur est remis à zéro pour attendre le geste suivant. D'autres fonctionnent de façon continue ; ils délivrent une série de notifications qui débute quand le geste est reconnu et dure tant qu'il est maintenu.

Analyseurs discrets

- Un analyseur de la classe `UITapGestureRecognizer` reconnaît les **tapes**, simples ou multiples avec un ou plusieurs doigts. L'analyseur est programmé pour un nombre de tapes et un nombre de doigts définis.
- Un analyseur de la classe `UISwipeGestureRecognizer` reconnaît les glissements. Il est programmé pour reconnaître un geste dans une ou plusieurs des quatre directions et avec un nombre de doigts prédéterminés.

Analyseurs continus

- Un analyseur de la classe `UIPinchGestureRecognizer` reconnaît les **pincements** ; rapprochement ou éloignement de deux doigts sur l'écran. L'analyseur peut être interrogé sur le taux et la vitesse d'éloignement des deux doigts.
- Un analyseur de la classe `UIPanGestureRecognizer` reconnaît les **déplacements libres** sur l'écran d'un ou plusieurs doigts. L'analyseur est programmé pour un minimum et un maximum de touches simultanées. Il délivre le vecteur et la vitesse de déplacement.
- Un analyseur de la classe `UIRotationGestureRecognizer` reconnaît les **rotations** avec deux doigts. L'analyseur peut être interrogé sur la valeur et la vitesse de rotation.
- Un analyseur de la classe `UILongPressGestureRecognizer` reconnaît les **appuis prolongés**. L'analyseur est programmé pour un nombre de tapes (une par défaut), un nombre de doigts (un par défaut), une durée minimale (0,4 secondes par défaut) et un déplacement maximal (10 pixels).

Classe UIGestureRecognizer

La méthode `–initWithTarget:action:` est définie dans la classe abstraite `UIGestureRecognizer`. Elle doit être appliquée à la création d'une instance de classe concrète pour associer l'analyseur à un couple *cible-action*. Le sélecteur passé en paramètre doit répondre à la signature standard d'une action ; il peut recevoir un paramètre qui est l'émetteur de l'action (*sender*). Des exemples de code seront donnés plus loin.

L'émetteur de l'action peut être utilisé par la cible pour connaître les caractéristiques du geste qui vient d'être reconnu ; en particulier, l'emplacement du geste dans la vue qui est rendu par l'une des deux méthodes :

- `-locationInView:` ;
- `-locationOfTouch:inView:` pour connaître l'emplacement d'une touche particulière.

La propriété `numberOfTouches` permet de connaître le nombre de touches du geste, et la propriété `enabled`, de type `BOOL` et dont l'accesseur est `isEnabled`, permet d'autoriser ou pas le fonctionnement de l'analyseur (il est autorisé par défaut).

Propriétés particulières à chaque geste

Pour chaque classe concrète dérivée de `UIGestureRecognizer`, les propriétés et méthodes permettant de programmer l'analyseur (**P**) ou d'obtenir les caractéristiques du geste reconnu (**C**) sont résumées dans le tableau ci-après

Tableau 13.2 : Propriétés et méthodes des analyseurs de geste

Analyseur concret	Propriété ou méthode	P/C	Usage
`UITapGesture Recognizer`	`NSUInteger number OfTaps Required`	P	Nombre de tapes requis pour reconnaître le geste (1 par défaut)
	`NSUInteger number OfTouches Required`	P	Nombre de doigts requis pour reconnaître le geste (1 par défaut)
`UIPinch Gesture Recognizer`	`CGFloat scale`	C	Facteur de pincement
	`CGFloat velocity`	C	Vélocité en facteur d'échelle par seconde
`UIPanGesture Recognizer`	`NSUInteger maximum Number OfTouches`	P	Nombre maximum de doigts requis pour reconnaître le geste
	`NSUInteger minimum Number OfTouches`	P	Nombre minimum de doigts requis pour reconnaître le geste (1 par défaut)
	`-(CGPoint) translationInView: (UIView *)view`	C	Déplacement du doigt dans les coordonnées de la vue passée en paramètre
	`-(CGPoint )velocity InView: (UIView *)view`	C	Vitesse de déplacement du doigt dans les coordonnées de la vue passée en paramètre
`UISwipe Gesture Recognizer`	`UISwipeGesture RecognizerDirection direction`	P	Direction du déplacement pour reconnaître le geste
	`NSUInteger number OfTouches Required`	P	Nombre de doigts requis pour reconnaître le geste (1 par défaut)

Tableau 13.2 : Propriétés et méthodes des analyseurs de geste

Analyseur concret	Propriété ou méthode	P/C	Usage
`UIRotation Gesture Recognizer`	`CGFloat rotation`	C	Rotation en radians depuis le dernier message
	`CGFloat velocity`	C	Vitesse de rotation en radians/seconde
`UILongPress Gesture Recognizer`	`CFTimeInterval minimumPress Duration`	P	Durée minimale de la touche (défaut 0,4 s)
	`NSInteger number OfTouches Required`	P	Nombre de doigts requis pour reconnaître le geste (1 par défaut)
	`NSUInteger number OfTaps Required`	P	Nombre de tapes requis pour reconnaître le geste (1 par défaut)
	`CGFloat allowable Movement`	P	Déplacement maximum pour reconnaître le geste (défaut 10 pixels)

`UISwipeGestureRecognizerDirection` est un type énuméré qui admet les constantes suivantes :

- `UISwipeGestureRecognizerDirectionRight`, glissement vers la droite ;
- `UISwipeGestureRecognizerDirectionLeft`, glissement vers la gauche ;
- `UISwipeGestureRecognizerDirectionUp`, glissement vers le haut ;
- `UISwipeGestureRecognizerDirectionDown`, glissement vers le bas.

Ces valeurs peuvent être combinées par l'opérateur `|` pour autoriser la reconnaissance du geste selon plusieurs directions.

Utiliser un analyseur de geste

Nous allons illustrer le fonctionnement des analyseurs de geste par la mise en œuvre d'un analyseur de pincement (continu) et d'un analyseur de tapes (discret).

Créer l'interface utilisateur

1 Créez une application pour iPad basée sur une vue (*View-based application*) intitulée *AnalyseurDeGeste*. Ouvrez le fichier *AnalyseurDeGesteViewController.xib* et placez trois labels et trois champs de texte.

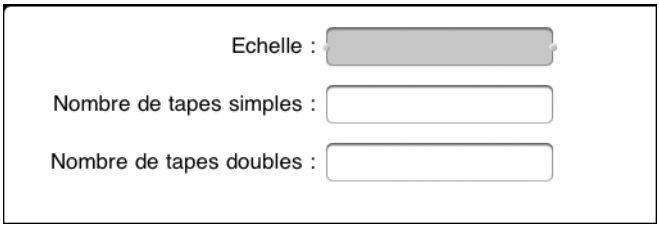


Figure 13.11 : Interface de l'analyseur de geste

2 Créez les outlets pour les champs de texte dans l'interface du contrôleur de vue dans le fichier *AnalyseurDeGesteViewController.h*. Profitez-en pour ajouter une variable d'instance afin de compter les tapes simples et doubles et pour rendre compte du pincement :

```
@interface AnalyseurDeGesteViewController :
                                        UIViewController {
   IBOutlet UITextField *scaleField;
   IBOutlet UITextField *simpleTaps;
   IBOutlet UITextField *doubleTaps;
   CGFloat scale;
   NSUInteger simpleTapCounter;
   NSUInteger doubleTapCounter;
}
@property(retain,nonatomic) UITextField *scaleField;
@property(retain,nonatomic) UITextField *simpleTaps;
@property(retain,nonatomic) UITextField *doubleTaps;
@end
```

3 Effectuez les liens entre les outlets et les champs de texte sous Interface Builder.

Mettre en œuvre les analyseurs

Les analyseurs de geste communiquent par le mécanisme *cible-action* lorsqu'ils reconnaissent un geste. Ajoutez une action dans le contrôleur de vue pour chaque geste que nous souhaitons détecter (simple tape, double tape et pincement) :

```
@interface AnalyseurDeGesteViewController :
                                        UIViewController {
   IBOutlet UITextField *scaleField;
   IBOutlet UITextField *simpleTaps;
   IBOutlet UITextField *doubleTaps;
   CGFloat scale;
   NSUInteger simpleTapCounter;
   NSUInteger doubleTapCounter;
}
@property(retain,nonatomic) UITextField *scaleField;
@property(retain,nonatomic) UITextField *simpleTaps;
```

```
@property(retain,nonatomic) UITextField *doubleTaps;
- (IBAction) scaleChange:(UIGestureRecognizer *)sender;
- (IBAction) simpleTap;
- (IBAction) doubleTap;
@end
```

Nous allons maintenant écrire le code pour mettre en œuvre les analyseurs.

Analyseur de tapes

La méthode `–initWithTarget:action:` définie dans la classe mère de tous les analyseurs de geste (`UIGestureRecognizer`) permet d'identifier le couple *cible-action* à actionner à la reconnaissance du geste. Par exemple, pour l'analyseur de tape simple :

```
UITapGestureRecognizer *simpleTapRecognizer ;
simpleTapRecognizer = [[UITapGestureRecognizer alloc]
         initWithTarget:self action:@selector(simpleTap)];
```

Ensuite, il faut programmer l'analyseur en utilisant ses propriétés. Pour un analyseur de tapes, les propriétés sont :

- `numberOfTapsRequired`, nombre de tapes requises ;
- `numberOfTouchesRequired`, nombre de doigts requis.

Enfin, la méthode `–addGestureRecognizer:` définie dans `UIView` permet d'attacher des analyseurs de geste à une vue. On peut ensuite libérer la référence à l'analyseur car elle est retenue par la vue :

```
[self.view addGestureRecognizer:simpleTapRecognizer];
[simpleTapRecognizer release];
```

Dans notre application, la réception des tape simples et doubles va simplement incrémenter leur compteur respectif. Saisissez le code des actions dans le fichier *AnalyseurDeGesteViewController.m* :

```
- (IBAction) simpleTap{
   simpleTapCounter++;
   simpleTaps.text =
        [NSString stringWithFormat:@"%d",simpleTapCounter];
}
- (IBAction) doubleTap{
   doubleTapCounter++;
   doubleTaps.text =
        [NSString stringWithFormat:@"%d",doubleTapCounter];
}
```

Analyseur de pincement

L'analyseur de pincement est encore plus simple à créer car il ne se programme pas :

```
UIPinchGestureRecognizer *pinchRecognizer ;
pinchRecognizer = [[UIPinchGestureRecognizer alloc]
      initWithTarget:self action:@selector(scaleChange:)];
[self.view addGestureRecognizer:pinchRecognizer];
[pinchRecognizer release];
```

Nous utilisons ici un sélecteur dont le nom finit par deux points pour indiquer qu'il faut lui passer l'émetteur de l'action (*sender*) en paramètre lors de la notification. Ainsi nous pourrons interroger l'analyseur sur les caractéristiques du geste, ce qui était inutile avec les tapes qui n'ont pas de caractéristique particulière.

L'action connectée au pincement affiche le facteur d'échelle à l'écran. Saisissez le code de l'action dans le fichier *AnalyseurDeGesteViewController.m* :

```
- (IBAction) scaleChange:(UIGestureRecognizer *)sender{
   scale = 100.*[(UIPinchGestureRecognizer *)sender scale];
   scaleField.text=[NSString stringWithFormat:@"%f",scale];
}
```

Initialiser l'application

Le code pour initialiser les analyseurs de geste doit être inséré dans la méthode `–viewDidLoad` du contrôleur de vue. Cette méthode contient aussi le code pour initialiser les champs de texte et les variables d'instances. Saisissez ce code dans le fichier *AnalyseurDeGesteViewController.m* :

```
- (void)viewDidLoad {
   [super viewDidLoad];
   // Create and configure the Pinch recognizer
   UIPinchGestureRecognizer *pinchRecognizer ;
   pinchRecognizer = [[UIPinchGestureRecognizer alloc]
       initWithTarget:self action:@selector(scaleChange:)];
   [self.view addGestureRecognizer:pinchRecognizer];
   // Create and configure the Double Tap recognizer
   UITapGestureRecognizer *doubleTapRecognizer ;
   doubleTapRecognizer = [[UITapGestureRecognizer alloc]
          initWithTarget:self action:@selector(doubleTap)];
   doubleTapRecognizer.numberOfTapsRequired = 2;
   doubleTapRecognizer.numberOfTouchesRequired = 1;
   [self.view addGestureRecognizer:doubleTapRecognizer];
   // Create and configure the Simple Tap recognizer
   UITapGestureRecognizer *simpleTapRecognizer ;
   simpleTapRecognizer = [[UITapGestureRecognizer alloc]
         initWithTarget:self action:@selector(simpleTap)];
   simpleTapRecognizer.numberOfTapsRequired = 1;
   simpleTapRecognizer.numberOfTouchesRequired = 1;
   [self.view addGestureRecognizer:simpleTapRecognizer];
   // Release the Gesture Recognizers
   [pinchRecognizer release];
```

```
    [doubleTapRecognizer release];
    [simpleTapRecognizer release];
    // Counters initialization
    scale = 100.;
    simpleTapCounter = 0;
    doubleTapCounter = 0;
    // Text Field initialization
    scaleField.text =
                    [NSString stringWithFormat:@"%f",scale];
    simpleTaps.text =
        [NSString stringWithFormat:@"%d",simpleTapCounter];
    doubleTaps.text =
        [NSString stringWithFormat:@"%d",doubleTapCounter];
}
```

Complétez le code du contrôleur de vue avec la synthèse des accesseurs des propriétés, construisez l'application et testez-la.

Améliorer le comportement

Lorsque vous testez l'application, vous constatez que deux tapes rapprochées sont comptabilisées deux fois :

- La première tape est comptée comme une tape simple.
- La seconde est comptée comme la deuxième tape d'une tape double ; elle n'est pas comptée comme une tape simple.

L'analyseur de tape double ayant reconnu un geste, une notification est émise par le mécanisme cible-action, et l'analyse des gestes est interrompue ; c'est le comportement par défaut.

Dans certains cas, il est souhaitable de retarder la reconnaissance de la tape simple ; attendre l'éventualité d'une deuxième tape pour déterminer s'il s'agit d'une tape simple ou double. La méthode `–requireGestureRecognizerToFail:` permet de demander au récepteur du message de notifier la reconnaissance du geste uniquement si l'analyseur passé en paramètre déclare qu'il n'a pas reconnu son geste.

Pour éviter que la première tape d'une tape double ne soit interprétée comme une tape simple, ajoutez la ligne de code suivante dans la méthode `–viewDidLoad` :

```
[simpleTapRecognizer
       requireGestureRecognizerToFail:doubleTapRecognizer];
```

Challenge

Complétez l'application *AnalyseurDeGeste* pour tester tous les gestes proposés par la version 3.2 du SDK.

Synchroniser les analyseurs

Dans la version 3.2, les événements de touches sont transmis simultanément à la vue et aux éventuels analyseurs de geste qui lui sont attachés.

Les analyseurs de geste sont très faciles à utiliser, le seul point délicat est leur synchronisation lorsque plusieurs sont attachés à la même vue. Un seul geste est reconnu lorsqu'un événement est reçu par la vue, le premier analyseur qui déclenche une action bloque tous les autres, le problème est que l'on ne sait pas lequel est "le premier".

Par exemple si l'on programme simultanément un analyseur de tape pour une tape simple et un second pour une tape double, une tape double peut déclencher, en fonction de l'analyseur qui est "le premier" :

- soit la notification d'une tape simple puis d'une tape double ;
- soit la notification de deux tapes simples.

Le développeur dispose de plusieurs moyens pour synchroniser le fonctionnement des analyseurs de geste :

- La propriété `enabled` permet de bloquer ou d'activer le fonctionnement d'un analyseur.
- La méthode `-requireGestureRecognizerToFail:`, que nous venons d'utiliser, permet de définir la priorité entre deux analyseurs.
- L'un des *délégués* de deux analyseurs peut permettre qu'ils émettent une notification simultanée, au lieu que l'un bloque l'autre comme dans le fonctionnement par défaut.
- Le *délégué* d'un analyseur peut bloquer temporairement sont fonctionnement.

Délégué d'analyseur

Chaque analyseur dispose d'une propriété `delegate` qui référence son éventuel délégué. Ce dernier doit répondre au protocole `UIGesture RecognizerDelegate` qui déclare les trois méthodes suivantes :

- `-gestureRecognizerShouldBegin:` est appelée lorsque l'analyseur est sur le point de commencer l'analyse d'un geste. Elle doit retourner `YES` (valeur par défaut) si l'analyseur est autorisé à débuter l'analyse. Si elle répond `NO`, l'analyse est interrompue.
- `-gestureRecognizer:shouldReceiveTouch:` est appelée avant qu'un événement de touche ne soit transmis à l'analyseur. Elle doit

retourner `YES` (valeur par défaut) pour autoriser la transmission et `NO` pour l'interdire.

- `–gestureRecognizer:shouldRecognizeSimultaneouslyWithGesture Recognizer:` est appelée lorsque deux analyseurs sont sur le point de reconnaître simultanément un geste. La simultanéité est autorisée si l'un des délégués répond `YES`. `NO` est la valeur par défaut.

13.4. Checklist

Nous avons exploré dans ce chapitre les spécificités du développement d'applications pour iPad concernant l'interface utilisateur.

Nous avons commencé par les réglages des paramètres de XCode pour sélectionner la cible des applications : iPhone ou iPad.

Concernant l'aspect visuel, nous savons maintenant :

- que l'utilisation des vues modales est limitée à des cas très particuliers et que l'on peut modifier leur aspect visuel avec la propriété `modalPresentationStyle` ;
- que pour les autres usages, il faut préférer les *Vues contextuelles* (*popover*) avec la mise en œuvre du contrôleur `UIPopoverController` et de son délégué `UIPopoverControllerDelegate` ;
- que les applications de type *Vue scindée* (*splitview*) sont à préférer à la navigation hiérarchique, avec la mise en œuvre du contrôleur `UISplitViewController` et de son délégué `UISplit ViewController Delegate`.

Concernant la gestion des événements de touche, nous avons examiné le fonctionnement et l'utilisation des *analyseurs de geste* (*gesture recognizer*).

ANNEXE

14.1. Épilogue

Nous avons exploré dans cet ouvrage les principaux frameworks permettant de mettre en œuvre les possibilités de l'iPhone, de l'iPod Touch et de l'iPad, en nous concentrant sur celles qui rendent ces appareils si attrayants : écran multi Touch, accéléromètres, capteur magnétique.

Nous vous laissons explorer par vous-même, en fonction de vos besoins, les technologies plus classiques, par exemple celles liées à la connectivité, puis celles plus avancées (sécurité, debugger, tests automatiques, etc.). Certaines de ces technologies sont disponibles sur iPhone et iPad, d'autres sont spécifiques à ce dernier :

- possibilité élémentaire de communication de fichiers ;
- production de documents PDF ;
- gestion améliorée des polices de caractères ;
- possibilité d'adapter la saisie de texte, etc.

La documentation d'Apple est très bien faite et les forums de développeurs francophones vous seront d'une aide précieuse : par exemple http://forum.macbidouille.com ou http://www.pommedev.com.

Nous vous souhaitons un parcours rempli de découvertes et de succès.

14.2. Politique d'Apple

La suite de cette annexe est consacrée à la description de la politique d'Apple concernant les développeurs pour iPhone et aux processus de diffusion des applications.

Apple pouvant à tout moment modifier ces conditions, les informations contenues dans cette annexe sont fournies à titre purement indicatif.

Les différents statuts de développeur

Apple a défini trois niveaux de développeur :

- **développeur enregistré** (*Registered iPhone Developer*) qui permet :
  - de télécharger le SDK ;
  - d'accéder à la documentation Apple ;
  - de tester ses applications avec le simulateur d'iPhone.

- équipe de développement inscrite au **programme iPhone** (*iPhone Developer Program*) qui permet :
  - de tester ses application sur des appareils réels ;
  - de diffuser ses applications de façon limitée ;
  - d'accéder aux versions Bêta d'iPhone OS et du SDK.
- équipe de développement inscrite au **programme iTunes** (*iTunes Connect*) qui permet :
  - de diffuser ses applications sur l'AppStore ;
  - de percevoir des revenus.

Le tableau ci-après résume les conditions associées à ces différents statuts.

Tableau 14.1 : Conditions associées aux différents statuts

Niveau	Coût	Objectif	Condition
Enregistré	0	Développer sur simulateur	-
Programme Standard	99 $ par an (79 €)	Tester ou diffuser sur 100 appareils	Être enregistré
Programme Entreprise	299 $ par an (239 €)	Diffuser en interne dans l'entreprise	Être enregistré et être une société de plus de 500 personnes
Programme iTunes	0	Diffuser sur l'App Store	Avoir adhéré au programme standard

Programme Entreprise

Ce programme ne permet pas de diffuser sur l'AppStore. Il est utile uniquement pour les grandes organisations souhaitant développer des applications spécifiques à usage interne.

Programme Standard

Ce programme est ouvert aux personnes physiques et aux personnes morales (entreprises). Une équipe de plusieurs personnes peut adhérer à ce programme.

Diffusion des applications

Il existe trois façons de diffuser son application :

- diffusion privée sur un maximum de 100 appareils référencés, qui nécessite l'adhésion au programme Standard ;

- diffusion privée sur plus de 500 appareils, qui nécessite l'adhésion au programme Entreprise ;
- diffusion au grand public sur l'AppStore, qui nécessite l'adhésion au programme Standard et à l'iTunes Connect.

Diffusion des applications
La politique d'Apple interdit tout autre mode de diffusion.

Une application peut être mise en diffusion gratuite ou payante sur l'AppStore, au choix du responsable de l'équipe de développement :

- Le prix de l'application est défini par le responsable de l'équipe.
- 70 % du prix des ventes est reversé mensuellement à l'équipe de développement par Apple.
- Cette marge de 30 % prise par Apple intègre tous les services fournis par l'AppStore :
  - frais de paiement par carte de crédit ;
  - hébergement sur le site d'Apple ;
  - marketing.
- Les applications gratuites sont diffusées gratuitement sur l'AppStore.

Signature du code

Le code qui s'exécute sur un appareil réel doit obligatoirement être **signé** par l'adhérent au programme iTunes Connect. Cette signature permet à Apple de contrôler la diffusion des applications, puisqu'il faut passer par le programme Standard pour obtenir les **certificats** permettant de signer le code. Elle permet surtout aux utilisateurs d'être certains de l'origine des applications qu'ils utilisent ; elle est garantie par Apple. Sans cette précaution, un appareil éminemment communiquant tel que l'iPhone, et dans une moindre mesure l'iPod Touch, deviendraient très sensibles aux attaques malveillantes.

Apple a prévu trois types de signature en fonction de la destination du code :

- le **test** des applications sur des appareils réels ; le développeur doit les installer lui-même à partir de XCode ;

- la **diffusion limitée** (*diffusion ad hoc*) ; les applications peuvent être transmises aux possesseurs des appareils référencés ;
- la **diffusion publique**, sur l'AppStore.

La signature est réalisée sous XCode par le développeur (pour le test) ou par le responsable de l'équipe de développement (pour la diffusion). Elle nécessite :

- la possession d'un certificat qui identifie le développeur ou le responsable de l'équipe ; ce certificat est fourni par Apple et permet d'assurer aux utilisateurs que l'application a été réalisée par l'équipe identifiée ;
- l'identification de l'application ;
- pour les tests et les diffusions limitées, l'identification des appareils autorisés.

Certificats

La personne qui s'inscrit à un programme de développeur, Standard ou Entreprise, est considérée par Apple comme le représentant et responsable de l'équipe de développement. Chaque membre de l'équipe doit être un développeur enregistré sur le site des développeurs d'Apple afin de pouvoir télécharger le SDK et accéder à la documentation sur le site.

Le responsable de l'équipe est nommé **Agent**. Une équipe de développement ne peut avoir qu'un seul agent ; il déclare les identifiants d'application et a le droit de les diffuser. Il cumule également les privilèges attachés aux administrateurs.

Un **administrateur** gère les développeurs (il autorise leur certification) et le parc d'appareils. Il peut y avoir plusieurs administrateurs dans une équipe de développement. Ils sont nommés par l'agent.

Un simple **membre** peut demander un certificat personnel. Une fois qu'il l'a obtenu, il peut signer les applications qu'il produit pour les tester sur des appareils réels.

14.3. Processus de diffusion

Apple pouvant à tout moment modifier le site des développeurs, les copies d'écrans et le processus décrits dans cette section sont fournis à titre purement indicatif.

S'enregistrer comme développeur

Chaque personne physique qui souhaite parcourir le site des développeurs doit y être enregistrée : **http://developer.apple.com/iphone**.

Figure 14.1 : Enregistrement sur le site des développeurs sur iPhone

L'identifiant et le mot de passe sont personnels et ne doivent pas être communiqués, sauf aux mineurs de 13 à 17 ans qui peuvent utiliser ceux d'un de leurs représentants légaux.

Un développeur enregistré n'a pas le droit de divulguer les informations confidentielles auxquelles il a accès sur le site des développeurs : les informations relatives aux versions Bêta et les informations payantes sont confidentielles.

Si vous êtes personnellement déjà enregistré sur l'un des sites d'Apple, par exemple sur l'iTunes Store ou sur le service Mobile Me, vous pouvez utiliser le même **identifiant** pour vous enregistrer comme développeur. Il est généralement conseillé d'avoir des identifiants séparés pour l'usage privé (acheter de la musique sur l'iTunes Store, par exemple) et pour l'usage professionnel ou semi professionnel ; vendre des applications sur l'AppStore.

S'inscrire au programme des développeurs

Création d'un compte

Vous pouvez adhérer au programme des développeurs soit à titre individuel, soit comme représentant d'une personne morale et d'une

équipe de développement. Dans ce dernier cas, la personne physique qui s'inscrit au programme des développeurs doit avoir la capacité d'engager son organisation. Elle sera considérée comme responsable de l'équipe de développement par Apple, appelée **Agent** du programme pour l'organisation.

La personne qui adhère au programme des développeurs doit préalablement être enregistrée sur le site des développeurs. Apple propose le programme Standard et le programme Entreprise. Le programme Standard correspond à la majorité des usages ; il donne ensuite accès à la diffusion sur l'AppStore.

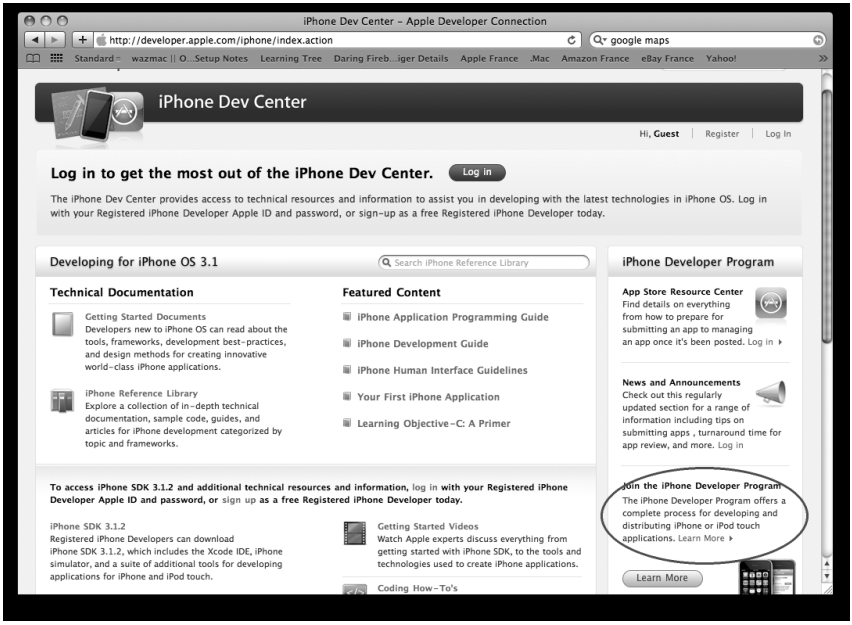


Figure 14.2 : Inscription au programme des développeurs

Administration du compte

Une fois l'inscription au programme Standard réalisée, l'agent et les éventuels administrateurs gèrent le compte à l'aide des portails web prévus à cet effet :

- le **centre des membres** (*Member Center*) pour gérer la composition de l'équipe ;
- le **portail des autorisations** (*iPhone Provisioning Portal*) pour gérer les certificats.

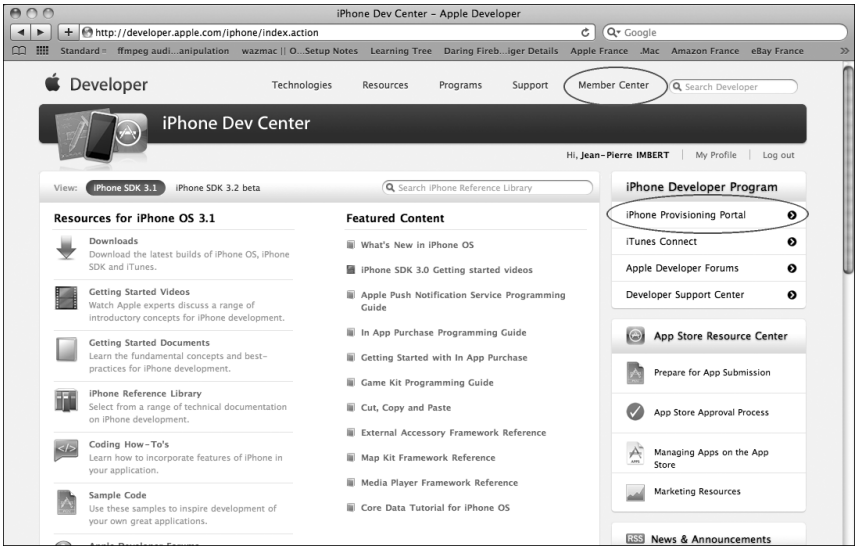


Figure 14.3 : Accès aux portails du programme des développeurs

Centre des membres

La page *People* du centre des membres permet aux administrateurs de gérer les membres de l'équipe.

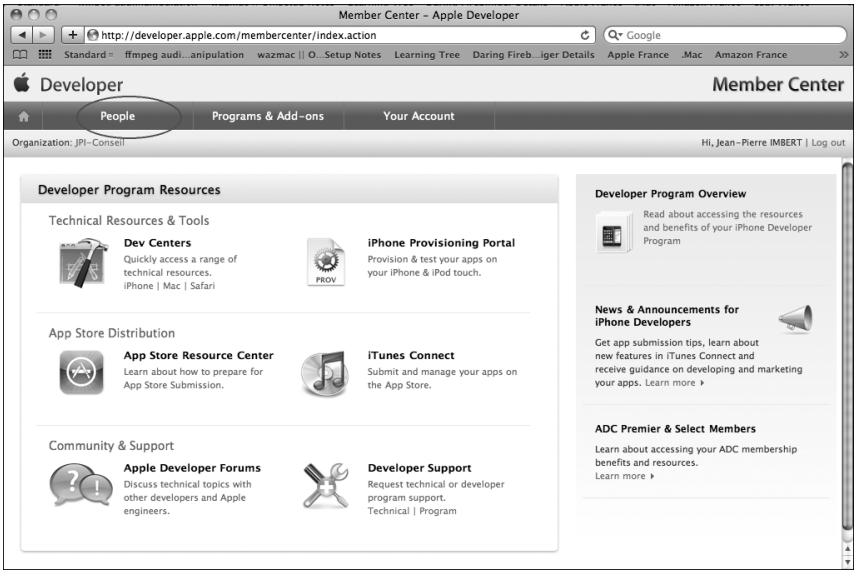


Figure 14.4 : Accès à la gestion des membres de l'équipe

Ils ont la possibilité d'inviter une ou plusieurs personnes à rejoindre l'équipe en fournissant pour chacune :

- son prénom ;
- son nom ;
- son adresse de courriel ;
- son rôle, administrateur (*Admin*) ou membre (*Member*).

Si vous souhaitez inviter un développeur dans votre équipe, saisissez les renseignements demandés et cliquez sur le bouton **Send Invitation** pour envoyer l'invitation.

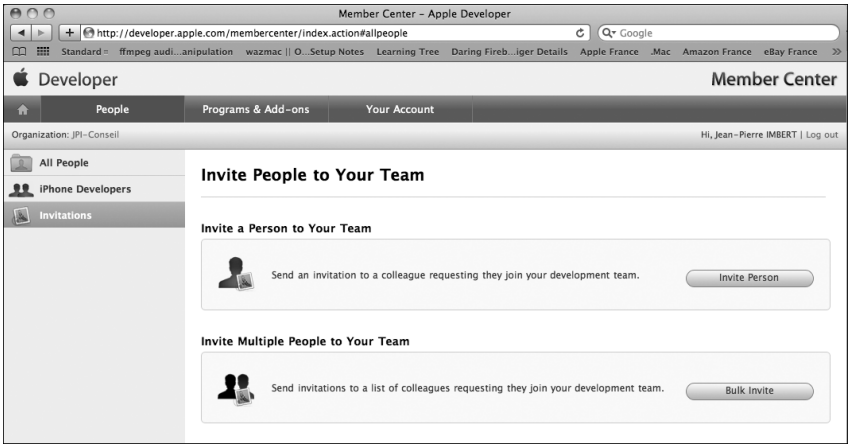


Figure 14.5 : Invitation d'un nouveau membre

Chaque personne invitée reçoit un courriel lui proposant de s'enregistrer comme développeur et de rejoindre l'équipe. Lorsqu'il aura accepté l'invitation, il pourra accéder au portail du programme avec son propre identifiant.

Portail des autorisations

Le bandeau gauche de la page principale du portail permet d'accéder à ses différentes fonctions :

- *Certificates* ; gestion des certificats des membres de l'équipe de développement et des certificats de distribution pour l'agent du compte ;
- *Devices* ; gestion des appareils de test ou pour la diffusion limitée des applications. Le nombre d'appareils est limité à 100 par an au total (test et diffusion, tout type d'appareil confondu) ;
- *App IDs* ; gestion des identifiants pour les applications ;

- *Provisioning* ; gestion des fichiers de Provisioning, ces fichiers permettent la signature et l'exécution du code ;
- *Distribution* ; informations pour la distribution et accès au portail de l'iTunes Connect.

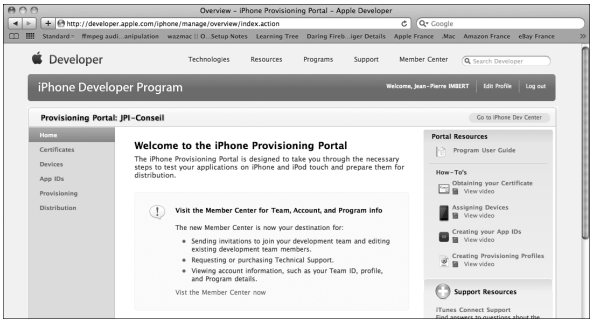


Figure 14.6 : Page principale du portail du programme des développeurs

Certifier un développeur

Afin de pouvoir signer une application pour la tester sur un ou plusieurs appareils, un développeur doit au préalable obtenir un certificat approuvé par un administrateur du programme des développeurs. Ces certificats doivent être renouvelés tous les ans.

Créer une demande de certificat

Le développeur commence par créer une demande de certificat.

1 Ouvrez l'application **Trousseau d'Accès** ; elle est située dans le dossier *Utilitaires* des *Applications*. Il faut vérifier que le paramétrage des certificats est adéquat. Sélectionnez la commande **Préférences** du menu **Trousseau d'accès** et vérifiez le paramétrage des certificats ; les protocoles OCSP et CRL doivent être désactivés.

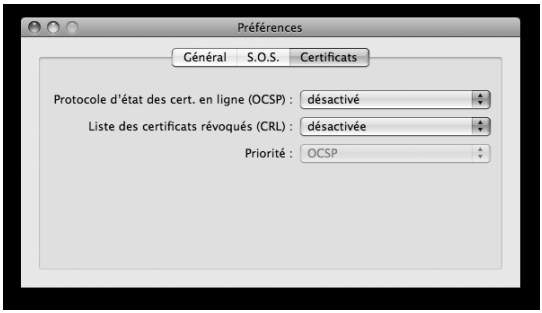


Figure 14.7 : Vérification du paramétrage des certificats

2 Fermez la fenêtre des préférences puis sélectionnez la commande **Demander un certificat à une autorité de certification ...** du sous-menu **Assistant de certification** du menu **Trousseau d'accès**.

3 Saisissez votre nom et votre adresse de courriel de la même façon que lors de votre enregistrement sur le site des développeurs. Sélectionnez les options **Enregistrée sur le disque** et **Me laisser indiquer les données sur la bi-clé** puis cliquez sur le bouton **Continuer**.

Figure 14.8 : Création d'une demande de certificat

4 Indiquez l'emplacement et le nom du fichier que vous souhaitez créer. Vous pouvez laisser le nom par défaut *CertificateSigningRequest.certSigningRequest*. Indiquez les paramètres de la bi-clé : `2048 bits` et `RSA`.

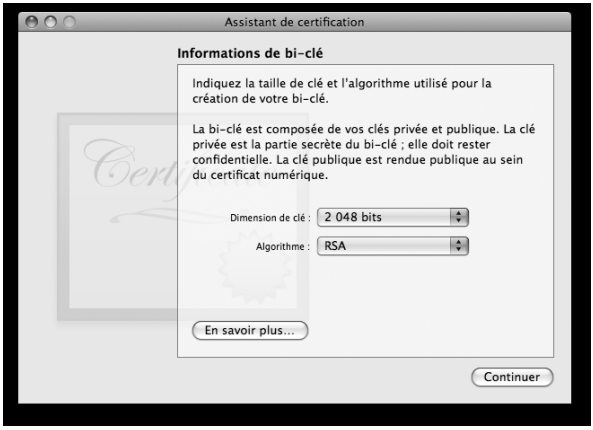


Figure 14.9 : Paramétrage de la bi-clé

Un couple de clé privée-publique est alors créé dans votre trousseau de session et le fichier de demande de certificat est créé. Il faut

maintenant envoyer cette demande sur le portail du programme des développeurs.

Conserver cette demande de certificat

Conservez cette demande de certificat, vous pourrez la réutiliser pour générer un certificat de diffusion.

Obtenir le certificat

1 Sélectionnez la fonction **Certificates** sur le portail du programme des développeurs. L'onglet **Development** vous permet de transférer le fichier contenant la demande de certificat (*CertificateSigningRequest.certSigningRequest*).

Un administrateur doit approuver la demande de certificat. Une fois cette validation effectuée, vous pouvez télécharger le certificat depuis le même onglet du portail.

Auto-approbation

Même lorsque l'équipe de développement est composée d'une seule personne, cette dernière doit approuver sa propre demande de certificat.

Figure 14.10 : Téléchargement des certificats

2 Téléchargez également le certificat intermédiaire WWDR.

Installer les certificats

Vous avez ainsi récupéré :

- votre certificat personnel, dans un fichier *developer_identity.cer* ;
- un certificat intermédiaire *AppleWWDRCA.cer*.

Double-cliquez sur chacun de ces fichiers pour les installer dans le **Trousseau d'accès** de votre session.

Les certificats sont maintenant installés et prêts à être utilisés sous **XCode**.

Tester son application sur un appareil

Identifier les appareils

L'identifiant d'un appareil peut être obtenu, lorsque l'appareil est connecté à un Mac, sous **XCode** ou sous **iTunes**. Sous iTunes, il suffit de cliquer sur le numéro de série de l'appareil, dans la page **Résumé**, pour faire apparaître son **identifiant** (*UDID*) ; c'est une suite de 40 caractères alphanumériques. Lorsqu'il est affiché sous iTunes, on peut copier l'identifiant dans le Presse-papiers (⌘+C).

La gestion des appareils de test sous XCode est réalisée dans la fenêtre *Organizer*.

1. Sélectionnez la commande **Organizer** du menu **Windows** ou tapez le raccourci clavier Ctrl+⌘+O.
2. Sélectionnez l'appareil dans le bandeau gauche et l'onglet **Summary** pour en afficher les paramètres, dont l'identifiant (identifier).

Figure 14.11 : Identifiant de l'appareil sous XCode

Enregistrer les appareils

Les administrateurs du compte du programme peuvent enregistrer des appareils de test sur le portail.

Enregistrez votre appareil sur le portail du programme des développeurs, dans la fonction **Devices**. Procédez de la même façon pour enregistrer tous les appareils sur lesquels vous souhaitez tester vos applications en cours de développement.

Figure 14.12 : Appareils enregistrés sur le portail

Vous pouvez enregistrer jusqu'à 100 appareils par an. Apple vous propose une fois par an d'apurer cette liste des appareils que vous n'utilisez plus.

Identifier les applications

Un **identifiant d'application** est composé d'un préfixe de 10 caractères fournis par Apple (*Bundle Seed ID*) et d'un identifiant de paquetage (*Bundle ID*) transmis par un administrateur. Il est recommandé d'utiliser une notation de domaine inversée pour identifier un paquetage, par exemple `com.jpiconseil.convertpro`.

Pour créer un identifiant d'application, connectez-vous sur le portail du programme et sélectionnez la fonction *App ID*. Cliquez sur le bouton **Create Add ID** et saisissez :

- un nom qui vous permettra de repérer l'identifiant pour en faciliter la gestion ;
- un identifiant de paquetage ; c'est cet identifiant qu'il faudra insérer dans les informations de l'application sous XCode.

Cliquez sur le bouton **Submit**. L'identifiant d'application est généré.

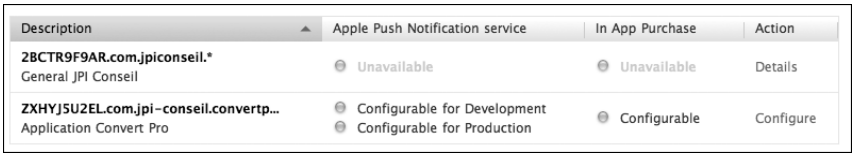


Figure 14.13 : Identifiants d'applications sur le portail

Identifiant du paquetage

L'identifiant du paquetage devra être saisi dans le fichier *Info.plist* de l'application sous XCode.

Partage de l'identifiant d'application

Les applications qui doivent partager les mêmes mots de passe, pour accéder à des sites web par exemple, doivent avoir le même identifiant d'application. Pour ce faire, il faut saisir un identifiant de paquetage avec un caractère joker "`*`", par exemple `com.jpiconseil.*`.

Générer un profil d'autorisation

Un **profil d'autorisation** (*provisioning profile*) est un fichier qui doit être installé sur un appareil afin de pouvoir y exécuter des applications. Il contient :

- un identifiant d'application ;
- un ou plusieurs certificats de développement ;
- un ou plusieurs identifiants d'appareil.

Les profils d'autorisation pour le test sont générés par un administrateur. Ils seront ensuite téléchargés par les développeurs pour être utilisés.

Pour créer un profil d'autorisation, connectez-vous sur le portail du programme des développeurs, sélectionnez la fonction **Provisioning** dans le bandeau gauche puis l'onglet **Development**.

Cliquez sur le bouton **New Profile**, saisissez les informations demandées puis cliquez sur **Submit**.

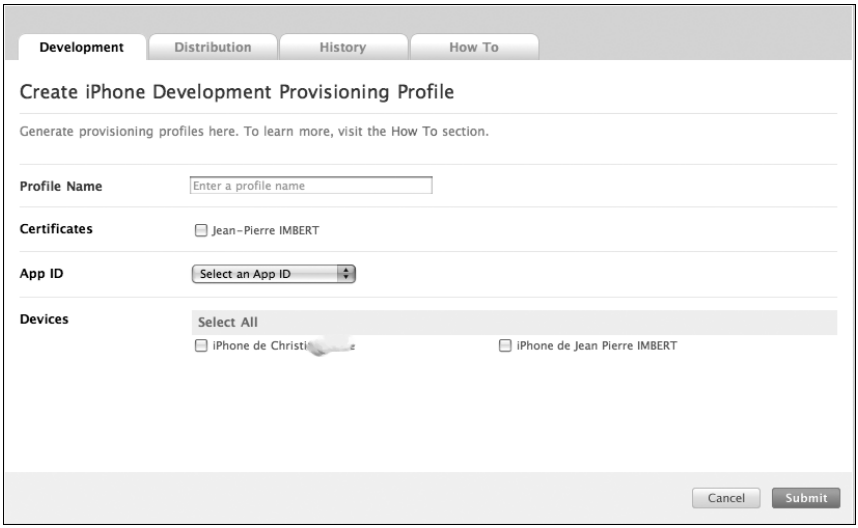


Figure 14.14 : Création d'un nouveau profil d'autorisation

Ces profils sont valables 3 mois. Ils peuvent être édités et renouvelés par un administrateur.

Installer un profil d'autorisation

Pour pouvoir tester une application sur un appareil, ce dernier doit être connecté sur le poste du développeur qui doit comporter :

- le certificat du développeur dans le Trousseau d'Accès ;
- le profil d'autorisation adéquat dans XCode.

Procédez ainsi :

1 Pour télécharger un profil d'autorisation, connectez-vous sur le portail du programme des développeurs, sélectionnez la fonction **Provisioning** dans le bandeau gauche puis l'onglet **Development**.

Figure 14.15 : Liste des profils disponibles

2 Cliquez sur le bouton **Download** du profil que vous souhaitez obtenir. Une fois le téléchargement terminé, connectez l'appareil sur lequel vous souhaitez transférer le profil, ouvrez la fenêtre **Organizer** sous XCode (Ctrl+⌘+O) puis faites glisser le fichier téléchargé (un fichier d'extension .*mobileprovision*) dans la zone *Provisioning* correspondant à l'appareil.

Figure 14.16 : Installation d'un profil d'autorisation sous XCode

Utilisation des profils téléchargés

Bien que tous les membres d'une équipe puissent télécharger un profil d'autorisation, seuls les développeurs dont le certificat est inclus dans le profil pourront utiliser celui-ci.

Construire l'application pour l'appareil

Une fois que vous disposez d'un certificat et d'un profil installé sous XCode, vous pouvez construire votre application pour l'appareil, l'y installer et la tester.

Compiler pour un appareil réel

Ouvrez le projet de votre application sous XCode et dans la barre d'outils de la fenêtre principale, sélectionnez un *iPhone Device* dans le menu déroulant **Active SDK**.

Figure 14.17 : Sélection du SDK pour un appareil réel

Signer l'application

1. Dans la liste gauche de la fenêtre principale, ouvrez le groupe *Targets* et sélectionnez la cible correspondant à votre application. Cliquez du bouton droit et sélectionnez la commande **Get Info** du menu contextuel (voir Figure 14.18).
2. Dans la fenêtre d'informations qui s'ouvre, saisissez `sign` dans le champ de recherche afin de limiter la liste des informations affichées. Ouvrez la rubrique **Code Signing Identity** pour définir la valeur du paramètre **Any iPhone OS Device** : `iPhone Developer` (voir Figure 14.19).

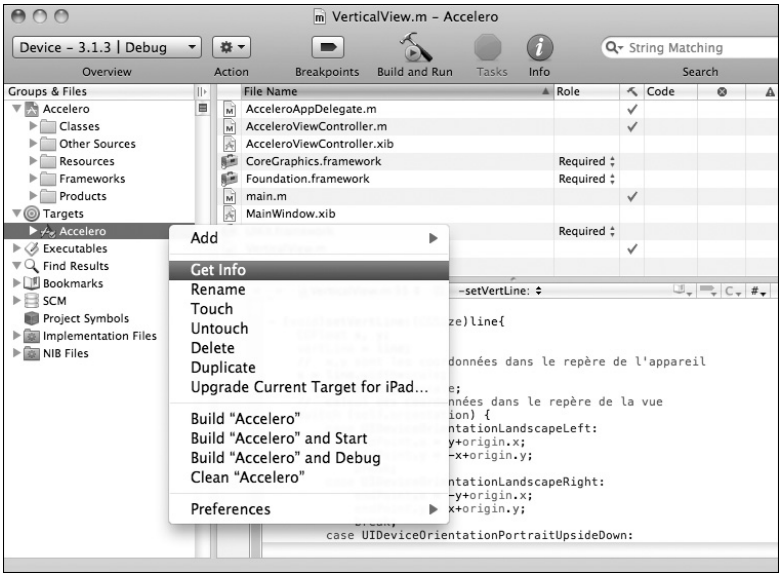


Figure 14.18 : Ouverture de la fenêtre des informations de l'application

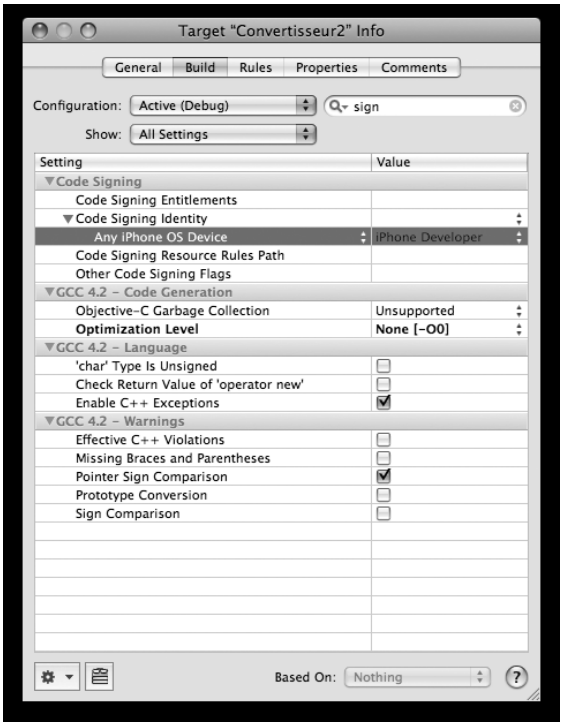


Figure 14.19 : Signature du code pour le test en développement

Identifier l'application

Sous XCode, sélectionnez le groupe *Ressources* puis le fichier *…Info.plist* de l'application. Modifiez le paramètre **Bundle Identifier** pour lui donner la valeur de l'identifiant de paquetage que vous avez défini lors de l'identification de l'application sur le portail du programme des développeurs.

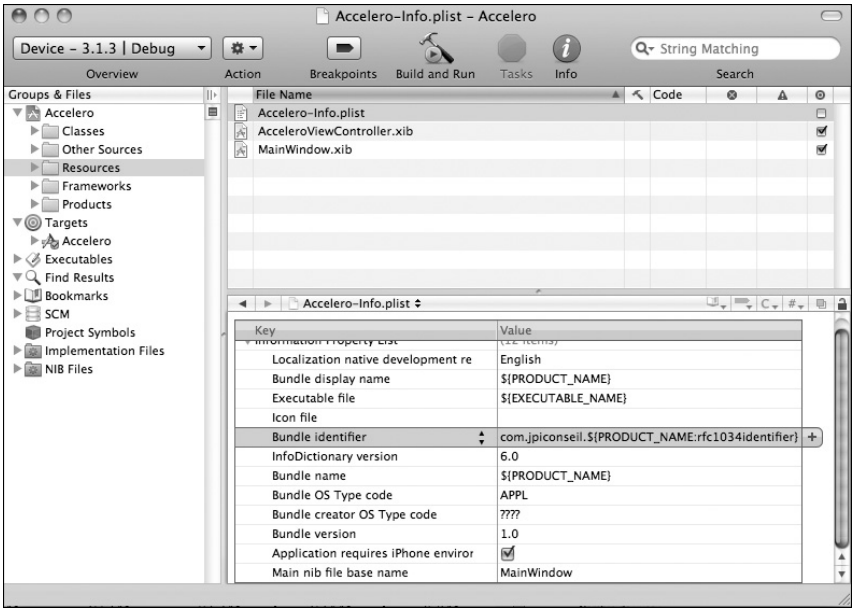


Figure 14.20 : Identification du paquetage sous XCode

Tester l'application sur l'appareil

Vous pouvez maintenant connecter votre appareil et lancer l'exécution depuis XCode (⌘+R). L'application est chargée sur l'appareil et exécutée sur celui-ci.

Diffusion limitée de son application

Une diffusion limitée permet un usage privé de votre application ou un bêta-test avant une diffusion publique.

Créer un certificat de diffusion

Seul l'agent du compte du programme des développeurs peut diffuser une application. Il doit pour cela commencer par créer son certificat de diffusion. Le processus est identique à celui utilisé pour créer un certificat de développeur.

Ce certificat doit être renouvelé tous les ans.

Créer une demande de certificat

La demande de certificat est créée exactement de la même façon que pour un développeur.

Utiliser la même demande de certificat

La demande de certificat que vous avez générée pour le certificat de développeur est utilisable pour obtenir un certificat de diffusion.

Obtenir le certificat

1 Sélectionnez la fonction **Certificates** sur le portail du programme des développeurs. L'onglet **Distribution** vous permet de transférer le fichier contenant la demande de certificat (*CertificateSigningRequest.certSigningRequest*). Cliquez sur le bouton **Request Certificate**.

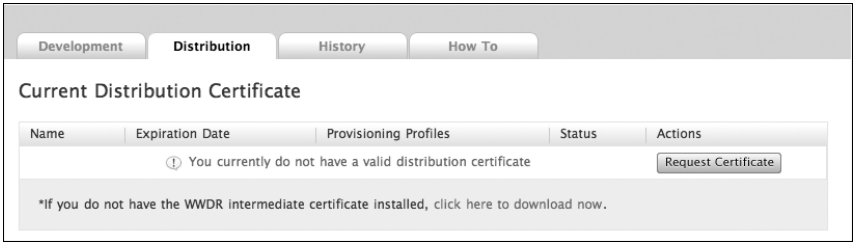


Figure 14.21 : Accès à la demande de certificat

2 Envoyez votre demande de certificat sur le site puis approuvez-la ; votre certificat de diffusion est prêt à être téléchargé.

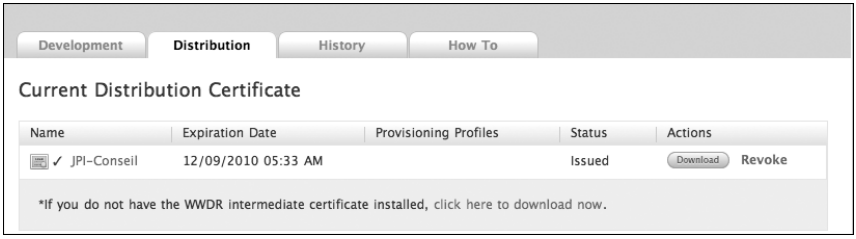


Figure 14.22 : Téléchargement du certificat

Installer le certificat

De la même façon que pour le certificat de développeur, double-cliquez sur le fichier *distribution_identity.cer* que vous venez de télécharger pour l'installer dans le **Trousseau d'accès** de votre session.

Votre certificat de diffusion est installé et prêt à être utilisé sous **XCode**.

Créer et installer un profil de diffusion limitée

Vous avez déjà généré un profil d'autorisation pour le développement. De la même façon, nous allons créer un profil d'autorisation pour la **diffusion limitée** (*ad hoc distribution provisioning profile*). Seul l'agent du compte peut créer et utiliser un profil de diffusion limitée.

1 Connectez-vous sur le portail du programme des développeurs, sélectionnez la fonction **Provisioning** dans le bandeau gauche puis l'onglet **Distribution**. Cliquez sur le bouton **New Profile**, saisissez les informations demandées – en particulier choisissez **ad hoc** pour une diffusion limitée –, puis cliquez sur **Submit**.

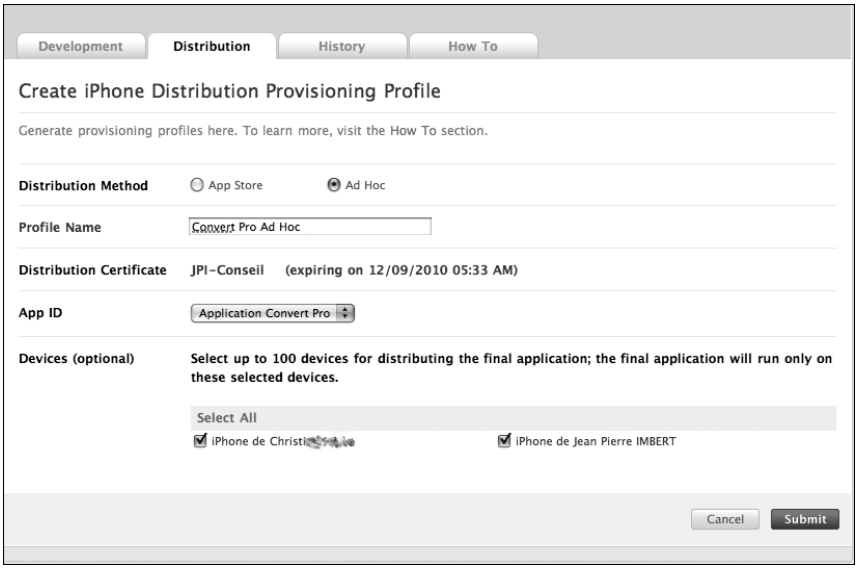


Figure 14.23 : Création d'un profil de diffusion limitée

Ces profils sont valables 1 an. Ils peuvent être édités et renouvelés par l'agent du compte.

2 Revenez à la liste des profils de diffusion et cliquez sur le bouton **Download** du profil que vous souhaitez obtenir.

3 Ouvrez la fenêtre **Organizer** sous XCode (Ctrl+⌘+O) puis faites glisser le fichier téléchargé (*Convert_Pro_Ad_Hoc.mobileprovision*) dans la zone *Provisioning* correspondant à l'appareil connecté.

Figure 14.24 : Installation d'un profil d'autorisation sous XCode

4 Conservez le fichier contenant le profil de diffusion ; les utilisateurs en auront besoin pour installer votre application sur leurs appareils.

Construire l'application pour la diffusion

Interdire le contrôle externe

Les applications en développement peuvent être lancées et examinées depuis l'ordinateur de développement sur lequel l'appareil est connecté en USB. Cette possibilité est interdite pour les applications diffusées, il faut donc configurer l'application à cet effet.

1 Sous XCod, créez un nouveau fichier (⌘+N). Dans la rubrique **Code Signing** de l'iPhone OS, choisissez le type **Entitlements** puis cliquez sur le bouton **Next** (voir Figure 14.25).

2 Donnez un nom à ce fichier, par exemple *Entitlement.plist*, puis cliquez sur le bouton **Finish**.

Figure 14.25 : Création d'un fichier Entitlements

Le fichier doit être à la racine

Vérifiez que le fichier est bien créé à la racine du projet, qu'il n'appartient à aucun groupe.

3 Sélectionnez ce fichier sous XCode et décochez la case *get-task-allow*.

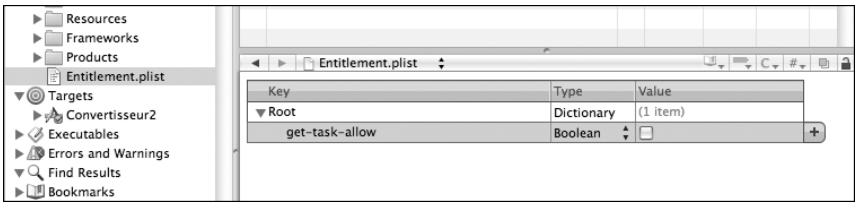


Figure 14.26 : Interdiction du contrôle externe

Construire l'application

La **construction pour la diffusion** nécessite un paramétrage spécifique. Nous allons donc créer une configuration sous XCode.

1 Cliquez du bouton droit sur la cible de l'application et sélectionnez la commande **Get Info** du menu contextuel qui s'affiche.

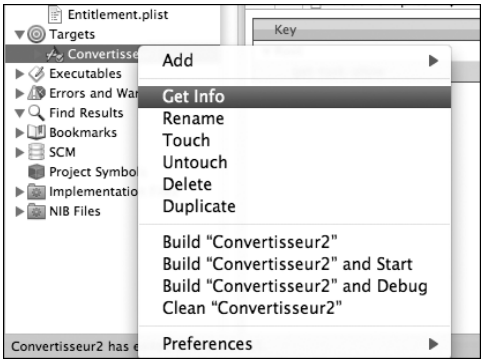


Figure 14.27 : Afficher les informations de la cible

2 Sélectionnez l'onglet **Build** dans la fenêtre d'information et ouvrez le menu **Configuration**. Sélectionnez la commande **Edit Configurations ….**

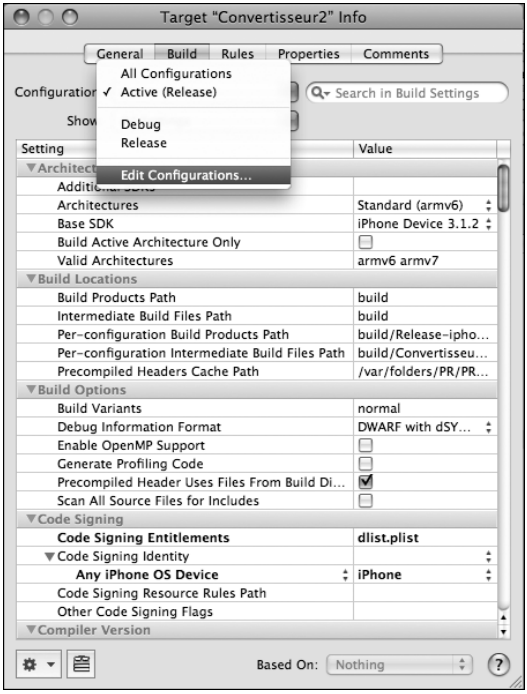


Figure 14.28 : Éditer les configurations

3 Sélectionnez la configuration *Release* dans la liste puis cliquez sur le bouton **Duplicate**. Changez le nom de la configuration créée : `Distribution`.

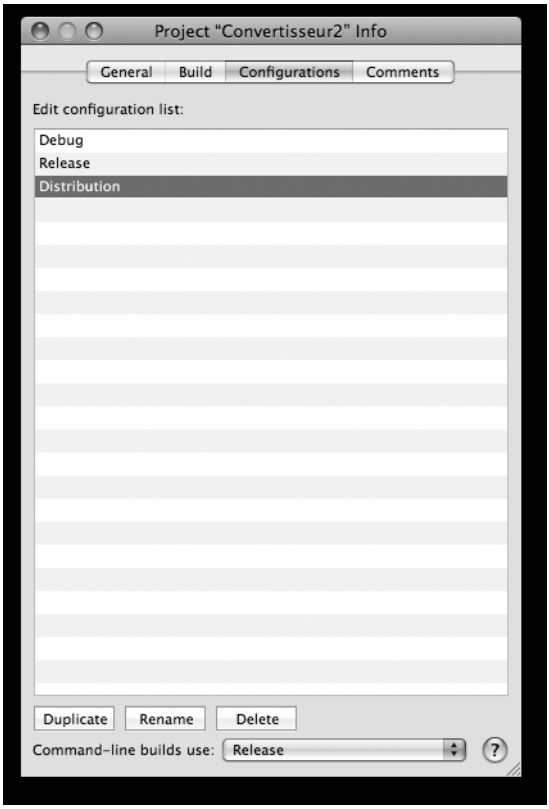


Figure 14.29 : Création d'une configuration Distribution

4 Refermez la fenêtre pour revenir aux informations de la cible. Sélectionnez la configuration *Distribution* puis saisissez `sign` dans le champ de recherche. Nous allons configurer la signature de l'application pour la distribution.

5 Modifiez le paramètre *Code Signing Entitlements*, saisissez le nom du fichier que vous avez créé : *Entitlement.plist*.

6 Modifiez le sous-paramètre *Any iPhone OS Device* du paramètre *Code Signing Identity* pour lui donner la valeur **iPhone Distribution**.

Figure 14.30 : Paramétrage de la configuration Distribution

7 Fermez la fenêtre d'information de la cible. Sous XCode, sélectionnez la configuration *Distribution* à l'aide de la commande **Set Active Build Configuration** du menu **Project**.

8 Effacez les constructions précédentes avec la commande **Clean** (Maj+⌘+K) et construisez l'application (⌘+B).

L'application est prête à être distribuée.

Distribuer son application

L'application terminée se trouve dans le dossier du projet sous XCode, dans le sous-dossier *build*.

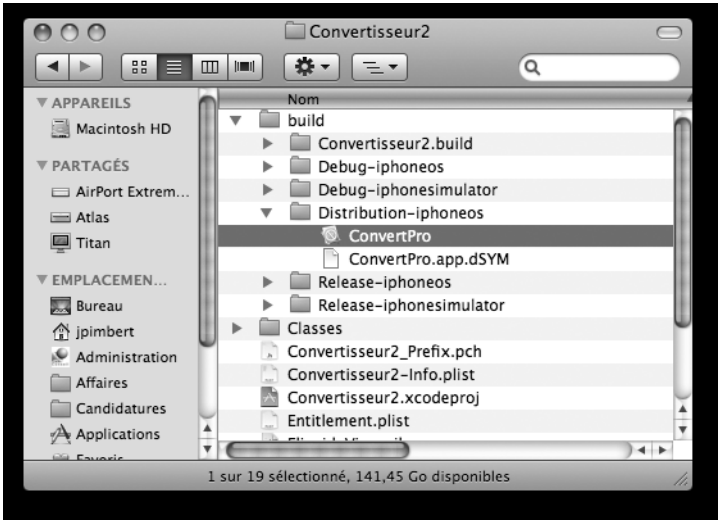


Figure 14.31 : Localisation de l'application terminée

Transmettez l'application ainsi que le profil de diffusion aux utilisateurs. Ils devront faire glisser ces deux fichiers dans l'application iTunes puis synchroniser leur appareil.

Compressez l'application

Il faut compresser l'application (format *ZIP*) pour la transmettre par courriel. Le type de fichier *.app* étant un dossier (*Bundle*), il est déconseillé de le placer directement en pièce jointe.

L'application est alors prête à être utilisée.

Diffuser son application sur l'AppStore

Apple met à la disposition des développeurs le site iTunes Connect (**https://itunesconnect.apple.com**) pour gérer la diffusion des applications sur l'App Store.

L'ouverture d'un compte sur le programme standard des développeurs provoque automatiquement la création d'un compte sur l'iTunes Connect avec le même mot de passe. Ce compte est immédiatement disponible pour la diffusion d'applications gratuites, la diffusion d'applications payantes sera possible après avoir saisi les informations bancaires et fiscales.

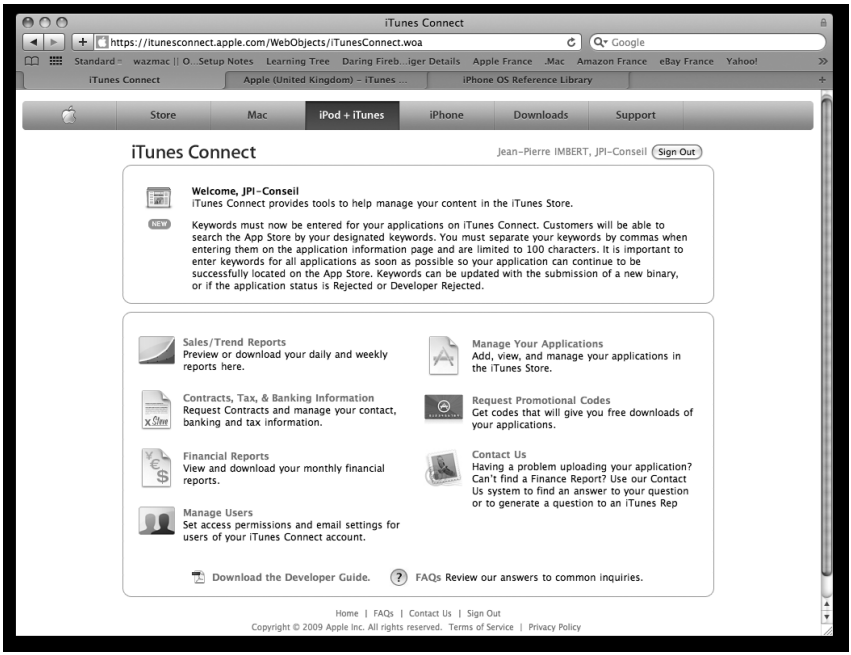


Figure 14.32 : Site web iTunes Connect

Engagement iTunes Connect

Lors de votre première connexion sur le site de l'iTunes Store, il vous est demandé d'approuver les conditions d'utilisation. En acceptant ces conditions, vous vous engagez à :

- fournir des informations juridiques, bancaires et fiscales exactes et précises concernant votre organisation, ou votre situation personnelle, et mettre à jour ces informations ;

- ne pas divulguer votre identifiant et votre mot de passe (vous avez la possibilité d'ajouter des utilisateurs sur le même compte, avec leur propre identifiant) ;
- accepter les conditions de vente sur l'App Store.

En outre, Apple ne garantit pas le bon fonctionnement de l'iTunes Connect.

Profil de diffusion

Le processus pour créer un profil de diffusion sur l'App Store est exactement le même que pour un profil de diffusion limitée. La seule différence est qu'il faut sélectionner *App Store* à la place de *Ad Hoc* pour le paramètre *Distribution Method* du profil.

La construction de l'application est réalisée de la même façon quel que soit le type de diffusion.

Soumission d'une application

Considérations légales

Vous devez donner un nom à votre application afin de la diffuser ; veillez à ne pas utiliser le nom d'une marque dont vous ne détiendriez pas les droits.

Concernant l'utilisation des logos et des marques déposées par Apple, vous pouvez consulter la page http://www.apple.com/legal/trademark/guidelinesfor3rdparties.html. En résumé, vous n'avez le droit d'utiliser aucun nom de marque déposé par Apple ni aucun dérivé, à l'exception notable du terme *Mac* qui peut être utilisé lorsqu'il est combiné (*MacWorld*, *MacProject*, etc.) mais pas seul.

La liste des marques déposées par Apple peut être consultée sur la page http://www.apple.com/legal/trademark/appletmlist.html.

Préparer la soumission

La soumission d'une application sur l'iTunes Connect n'est pas complexe mais assez longue. Il est conseillé de préparer tous les éléments :

- le nom de l'application ;
- une description de l'application (4 000 caractères au maximum) ;
- le type d'appareil sur lequel l'application fonctionne : iPhone, iPod touch, iPad ou iPhone/iPad (pour les applications universelles) ;
- un numéro de référence (SKU) devant identifier sans ambiguïté chaque version majeure de chacune de vos applications ;

- la catégorie principale de l'application (voir le tableau) et éventuellement une catégorie secondaire ;
- le numéro de version ;
- le copyright, par exemple `2010 JPI–Conseil` ;
- l'adresse du site web où l'utilisateur peut trouver des informations complémentaires ;
- une adresse de courriel ; cette adresse sera utilisée par Apple, elle ne sera pas diffusée ;
- éventuellement le texte de la licence d'utilisation de l'application, par défaut la licence standard de l'App Store s'applique ;
- la liste des pays ou zones géographiques sur lesquels on souhaite diffuser l'application ;
- la date de disponibilité de l'application ;
- le prix de l'application ;
- éventuellement, les localisations pour le texte descriptif et la licence d'utilisation. Les localisations admises sont l'anglais, le français, l'allemand, le hollandais, l'italien, l'espagnol et le japonais ;
- l'icône de votre application en grand format ; 512 x 512, 72 dpi, au format *JPEG* ou *TIFF* ;
- la copie de l'écran principal de votre application, au format *JPEG* ou *TIFF* ;
  - pour iPhone ou iPod Touch, 320 x 460 (sans la barre d'état), ou 320 x 480 (pour les applications plein écran), 480 x 300 ou 480 x 320 ;
  - pour iPad 1 024 x 748, 1 024 x 768, 768 x 1 004 ou 768 x 1 024.
- éventuellement des copies d'écrans additionnelles :

Tableau 14.2 : Catégories d'applications sur l'App Store

Anglais	Français	Anglais	Français
Book	Livres numériques	Navigation	Navigation
Business	Économie et entreprise	News	Actualités
Education	Enseignement	Photography	Ptotographie
Entertainment	Divertissement	Productivity	Productivité
Finance	Finance	Reference	Références
Games	Jeux	Social Networking	Réseaux sociaux
Healthcare & Fitness	Forme et santé	Sports	Sports

Tableau 14.2 : Catégories d'applications sur l'App Store

Anglais	Français	Anglais	Français
Lifestyle	Style de vie	Travel	Voyages
Medical	Médecine	Utilities	Utilitaires
Music	Musique	Weather	Météo

Après avoir saisi ces informations, il vous faudra attendre la validation d'Apple, de quelques jours à quelques semaines, avant de voir votre application sur l'App Store.

Autres services

Pour améliorer votre marketing, Apple vous autorise à utiliser gratuitement un logo afin de signaler la présence de votre application sur l'App Store, ainsi que les photos officielles d'un iPhone et d'un iPod touch pour y insérer vos copies d'écran.

Pour cela, renvoyez l'accord de licence signé. Consultez les informations sur le site http://developer.apple.com/iphone/appstore/.

Vous pouvez encore renforcer vos liens avec l'iTunes Store et vous affilier à l'iTunes : http://www.apple.com/uk/itunes/affiliates.

!

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

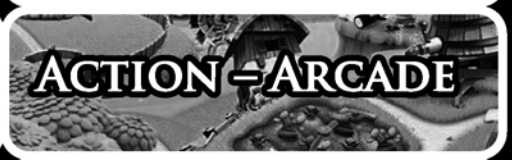
X

Z

Faites une pause détente !

1 Jeu par Jour

1 HEURE DE JEU GRATUIT
sur 365 jeux/an
et
VOTRE 1er JEU COMPLET OFFERT
lors de votre inscription !



www.1jeuparjour.com

D309

Composé en France par Jouve
11, bd de Sébastopol - 75001 Paris

LE GUIDE COMPLET

DÉVELOPPEZ VOS APPLICATIONS POUR iPHONE ET iPAD

« Le Guide Complet : la meilleure façon de faire le tour du sujet ! »

LE TOUR COMPLET DU SUJET TRAITÉ

- Programmation orientée objet
- Langage Objective-C
- XCode et Interface Builder
- Mécanisme Cible-Action
- Motifs MVC et KVC
- Délégation et notifications
- Cocoa Touch SDK
- Géo-localisation
- Accéléromètres
- 3D avec OpenGL
- Graphisme avec Quartz2D
- Spécificités de l'iPad
- Diffusion sur l'AppStore

LA PRATIQUE PAR L'EXEMPLE

- Manipulez des objets en Objective-C
- Détectez les fuites de mémoire avec Instruments
- Localisez une application
- Explorez les contrôles
- Exploitez Core Data pour la persistence des données
- Ajoutez des effets sonores
- Utilisez un analyseur de gestes
- Testez puis déployez pour iPad, iPod Touch ou iPhone

L'APPROFONDISSEMENT À TRAVERS DES EXERCICES

- Créez et utilisez une barre de navigation ou d'onglets
- Prenez, enregistrez, éditez et envoyez des photos
- Déterminez le positionnement géographique
- Orientez automatiquement les vues
- Traitez les événements gestuels
- Animez des images
- Créez un convertisseur
- Réalisez un mini billard

Jean-Pierre IMBERT Passionné du développement logiciel et supporteur des technologies Apple depuis le premier Macintosh, Jean-Pierre Imbert est actif sur plusieurs forums de développeurs Mac et iPhone, en français et en anglais. Il a enseigné successivement plusieurs disciplines de l'ingénierie du logiciel en écoles d'ingénieurs. Avec sa société jpi-conseil.com, il conseille les entreprises et anime des formations dans les techniques et méthodes les plus avancées en matière de développement.

Réf : 2802 / 65 4875 4
ISBN : 978-2-300-028021

www.microapp.com

Prix France : 20€ • 01005
9 782300 028021
Prix Belgique : 22€