2

Objective-C

Les applications iOS sont rédigées dans le langage de programmation Objective-C, qui se fonde sur la bibliothèque de fonctions Cocoa Touch. Objective-C est une extension du langage C. Cocoa Touch est une collection de classes Objective-C prédéfinies. Nous supposons dans ce livre que vous avez un minimum de connaissances du langage C et que vous comprenez les grands principes de la programmation objet. Si ces deux domaines ne vous sont pas familiers, nous vous conseillons d'étudier d'abord un livre tel que *Programmation Objective-C, le guide Big Nerd Ranch* dans la même collection.

Ce chapitre vous propose de découvrir les principes fondamentaux du langage Objective-C. Nous allons en profiter pour créer une application nommée RandomPossessions. Même ceux qui connaissent le langage Objective-C ont intérêt à parcourir et à pratiquer ce chapitre afin de créer la classe **`BNRItem`**, qui sera requise dans les chapitres suivants.

Notion d'objet

Supposons que nous ayons à modéliser l'organisation de soirées à thème. Chaque soirée possède plusieurs attributs spécifiques : son thème, sa date et la liste des invités. La fête peut également être associée à des actions telles que l'envoi d'un rappel par courriel à tous les invités, l'impression des étiquettes avec les noms ou même l'annulation de la fête.

En langage C, les données décrivant la soirée pourraient être stockées dans une structure. Une structure possède des données membres, une donnée par attribut de soirée. À chaque donnée membre correspond un nom et un type de donnée. Pour créer une soirée réelle, vous utiliseriez la fonction **`malloc`**, qui permet de réserver un espace dans la mémoire de travail pour y stocker le contenu de la structure. Vous définiriez des fonctions en langage C pour pouvoir stocker les valeurs des attributs et réaliser d'autres actions.

Dans le langage Objective-C, vous n'utilisez pas une structure mais une classe. Une classe ressemble à un modèle ou à une forme de découpe dont le résultat est un objet. La classe nommée **`Soiree`** permet donc de créer des objets. Les objets sont des instances de la classe **`Soiree`**. Chaque instance peut contenir les données d'un événement (voir Figure 2.1).

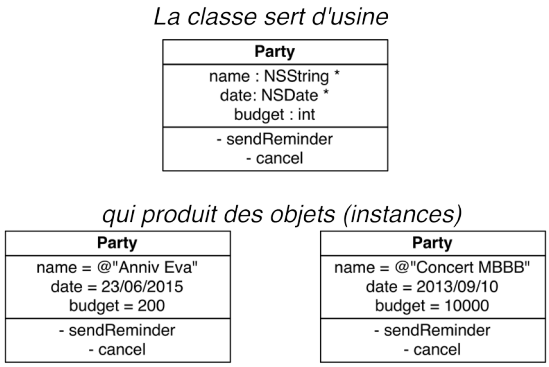


Figure 2.1
Une classe avec deux instances.

Une instance de la classe **`Soiree`** occupe un certain espace en mémoire pour y stocker ses données. Les valeurs de ces données sont stockées dans des variables d'instances. Chaque instance de **`Soiree`** possédera par exemple les variables d'instance `nom`, `date` et `budget`.

Pour résumer, une structure C et un objet sont des blocs de mémoire. Une structure C possède des données membres avec un nom et un type. De même, un objet possède des variables d'instance avec un nom et un type.

L'énorme différence entre la structure du langage C et la classe Objective-C est qu'une classe possède également des méthodes. Une méthode ressemble à une fonction : elle possède un nom, un type de valeur renvoyée et une liste de paramètres d'entrée. Les méthodes ont accès aux variables d'instance de l'objet. Pour faire exécuter le code d'une méthode d'un objet, vous lui envoyez un message portant le même nom que la méthode visée.

Utilisation des instances

Pour pouvoir utiliser une instance de classe (un objet), il vous faut d'abord disposer d'une variable qui contient un pointeur vers cet objet en mémoire. Une variable de type pointeur sert à mémoriser l'adresse à laquelle l'objet a été implanté. Elle ne contient pas l'objet lui-même. Voici comment déclarer une variable pointant sur un objet :

```
Soiree *instanceSoiree;
```

Ici, la variable s'appelle `instanceSoiree`. Elle va pointer sur une instance de la classe **`Soiree`**. Cette déclaration ne provoque pas la création d'un objet **`Soiree`**. Elle prépare une variable qui pourra pointer sur un objet du type **`Soiree`**.

Création d'un objet (instanciation)

Tout objet possède une durée de vie : il est créé, reçoit des messages puis est détruit quand il n'est plus requis.

Vous instanciez un objet au moyen du message **alloc** transmis à la classe. La classe y réagit en créant un objet en mémoire puis en renvoyant un pointeur sur cet objet, valeur que vous pouvez stocker dans le pointeur :

```
Soiree *instanceSoiree = [Soiree alloc];
```

Cette ligne crée une instance de **Soiree** et vous récupérez un pointeur qui est stocké dans la variable `instanceSoiree`. Dès que vous disposez de ce pointeur, vous pouvez envoyer des messages à l'objet. Le premier message que l'on envoie toujours à l'objet venant d'être créé est celui d'initialisation. L'objet est créé dès que vous avez émis le message **alloc**, mais il n'est pas utilisable tant qu'il n'a pas été initialisé.

```
[instanceSoiree init];
```

Puisque ces deux étapes sont toujours requises (allocation et initialisation), les deux messages sont en général combinés dans la même instruction :

```
Soiree *instanceSoiree = [[Soiree alloc] init];
```

Le code situé à droite de l'opérateur d'égalité (d'affectation) signifie ceci : "Créez une instance de **Soiree** et envoyez-lui le message **init**." Les deux messages **alloc** et **init** renvoient un pointeur vers l'objet venant d'être créé, ce qui permet ensuite de le manipuler.

Le fait de combiner deux messages dans la même ligne correspond à une émission de messages imbriqués. Les crochets intérieurs sont évalués d'abord. C'est pourquoi le message **alloc** est d'abord envoyé à la classe **Soiree**. Son résultat est une instance non initialisée de **Soiree** à laquelle est envoyé le message **init**.

Émission de messages

Que faire d'une instance une fois qu'elle est initialisée ? Vous pouvez lui envoyer d'autres messages.

Voyons plus en détail l'anatomie d'un message. Tout d'abord, tout message est délimité par des crochets droits. Au sein de ces crochets, le message compte trois parties :

- Récepteur. Pointeur sur l'objet qui doit exécuter la méthode.
- Sélecteur. Nom de la méthode à exécuter.
- Arguments. Valeurs à fournir en tant que paramètres d'entrée de la méthode.

Un objet **Soiree** aura normalement une liste d'invités que vous remplissez en envoyant un message **ajouterInvite** :

```
[instanceSoiree ajouterInvite:unePersonne];
```

Le fait d'envoyer le message **ajouterInvite:** à l'objet récepteur `instanceSoiree` provoque le déclenchement de la méthode **ajouterInvite:** (elle est citée dans le sélecteur) en lui transmettant un argument (`unePersonne`).

Dans l'exemple, le message **ajouterInvite:** ne possède qu'un argument, mais vous pouvez lui en fournir plusieurs ou aucun. C'est le cas du message **init**, qui n'a pas d'argument.

Un invité pourra choisir de venir avec un petit plat. Il faudrait alors définir une autre méthode nommée **`ajouterInvite:avecPlat:`**. Dans ce cas, le message attend deux arguments : le nom de l'invité et la nature du plat qu'il va apporter. À chaque argument correspond un label dans le sélecteur, le label se terminant toujours par un signe **`:`**. Ce que l'on appelle le sélecteur est l'ensemble des labels (voir Figure 2.2).

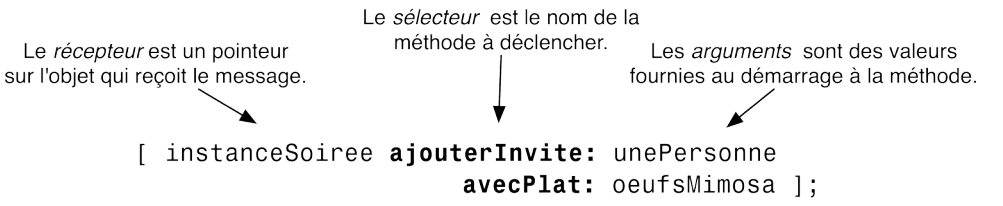


Figure 2.2

Portion d'un message à émettre.

Le couplage des labels avec les arguments est une fonction essentielle en Objective-C. La même méthode prendrait l'aspect suivant dans d'autres langages :

```
instanceSoiree.ajouterInviteAvecPlat(unePersonne, oeufsMimosa);
```

La lecture de cette déclaration ne permet pas aisément de savoir à quoi correspond chaque argument envoyé à la méthode. En Objective-C, à chaque argument correspond un label :

```
[instanceSoiree ajouterInvite:unePersonne avecPlat:oeufsMimosa];
```

Après une petite phase d'apprentissage, les programmeurs Objective-C apprécient beaucoup la clarté d'écriture des arguments dans les sélecteurs. Ce qu'il ne faut pas oublier, c'est qu'à chaque couple de crochets ne correspond qu'un seul message. Dans l'exemple, **`ajouterInvite:avecPlat:`** possède deux labels, mais cela ne forme qu'un message. L'envoi de ce message ne déclenchera qu'une méthode.

C'est le nom de la méthode qui permet de l'identifier uniquement dans Objective-C. Il est donc impossible d'avoir deux méthodes homonymes dans la même classe. En revanche, les labels des arguments de deux méthodes peuvent être les mêmes, tant que les noms complets des méthodes ne se confondent pas. Notre classe **`Soiree`** possède deux méthodes, **`ajouterInvite:`** et **`ajouterInvite:avecPlat:`**, qui ne sont pas confondues et qui ne partagent pas leur code.

Notez bien la distinction entre message et méthode. Une méthode est un bloc d'instructions à exécuter alors que le message est une demande d'exécution d'une méthode adressée à une classe ou à un objet. Le nom du message coïncide toujours avec le nom de la méthode qu'il faut exécuter.

Destruction d'un objet

Détruire un objet est fort simple, puisqu'il suffit de forcer la valeur de la variable pointeur à `nil` :

```
instanceSoiree = nil;
```

Cette instruction détruit l'objet désigné par la variable `instanceSoiree` en forçant la valeur de ce pointeur à `nil`. (En réalité, les choses sont un peu plus complexes, comme vous le verrez dans le chapitre suivant, qui traite de la gestion mémoire.)

La valeur `nil` est en réalité une pseudo-valeur correspondant à un pointeur annulé. (Les programmeurs C connaissent la valeur NULL et les programmeurs Java, la valeur `null`.) Lorsqu'un pointeur possède cette valeur, cela signifie que l'objet correspondant n'existe pas. Dans notre exemple, toutes les soirées possèdent un lieu. Tant que l'organisateur n'a pas choisi la salle, la variable `salle` contient la valeur `nil`. Cela permet d'envisager les choses suivantes :

```
if (salle == nil) {
  [organiser rappelTrouverSalleSoiree];
}
```

Les programmeurs Objective-C prennent l'habitude d'utiliser la forme abrégée pour vérifier si un pointeur vaut `nil` :

```
if (!salle) {
  [organiser rappelTrouverSalleSoiree];
}
```

L'opérateur **!** signifie non (il inverse une valeur vraie en valeur fausse et *vice versa*). Il en résulte que le test correspond à "Si la salle est NON fixée", la question sera vraie si la variable `salle` vaut effectivement `nil`.

Vous ne risquez rien lorsque vous tentez d'envoyer un message qui possède la valeur `nil`. Dans d'autres langages, cette opération est illégale, ce qui oblige à prendre ses précautions, comme ceci :

```
// Est-ce que salle est non-nil ?
if (salle) {
  [salle envoyerConfirmation];
}
```

En Objective-C, cette précaution est inutile parce que le message transmis à un objet inexistant est tout simplement ignoré. Voilà pourquoi vous pouvez tenter d'envoyer le message directement :

```
[salle envoyerConfirmation];
```

Si la salle n'a pas encore été choisie, vous n'enverrez aucune confirmation nulle part. (L'inconvénient de cette facilité est celui-ci : si votre programme ne fait rien alors que vous pensez qu'il devrait être en train de faire quelque chose, vous devrez aller vérifier s'il n'y a pas un pointeur `nil` inattendu quelque part.)

Développement de *RandomPossessions*

Avant de plonger dans les détails de la série de bibliothèques appelées UIKit, qui permet de créer des applications iOS, restons au niveau du langage Objective-C. Démarrez si nécessaire Xcode puis sélectionnez File/New/Project. Dans le volet gauche, cliquez dans la section OS X sur la catégorie Application puis, dans le volet de droite, choisissez le modèle Command Line Tool avant de cliquer sur le bouton Next (voir Figure 2.3).

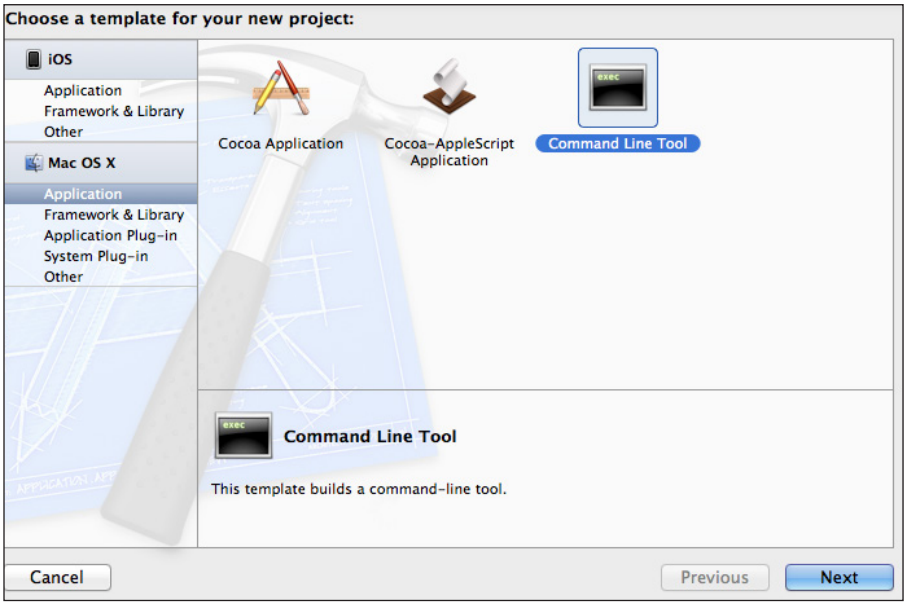
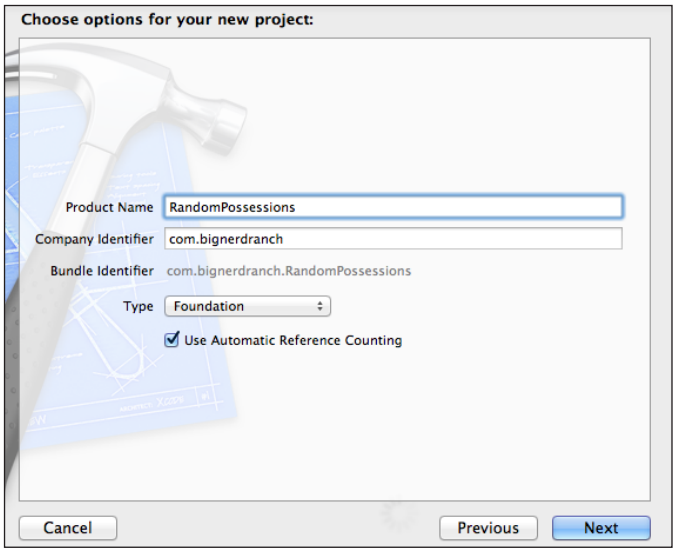


Figure 2.3
Création d'un outil sur ligne de commande.

Si vous ne trouvez aucun choix COMMAND LINE TOOL, vérifiez que vous avez bien choisi APPLICATIONS dans la section OS X. Sachez qu'Apple change de temps à autre les noms et les styles des modèles. Rendez-vous sur le site des développeurs Apple si nécessaire.

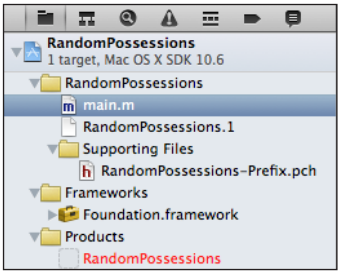
Dans l'écran suivant, commencez par saisir le nom du projet, **`RandomPossessions`**. Comme type, choisissez FOUNDATION et vérifiez que l'option USE AUTOMATIC REFERENCE COUNTING est bien cochée (voir Figure 2.4). Cliquez sur NEXT. Acceptez la suggestion d'enregistrer le projet dans son état initial. Choisissez un endroit logique, car nous allons réutiliser des portions de ce code dans les projets suivants.

Figure 2.4
Choix du nom du projet.



Un fichier de code source nommé main.m a déjà été créé dans le groupe RandomPossessions du navigateur de projet (voir Figure 2.5).

Figure 2.5
Le navigateur de projet avec le modèle COMMAND LINE TOOL.



Cliquez dans le nom main.m pour l'ouvrir dans l'éditeur. Vous constatez qu'une fonction **main** minimale a déjà été insérée. C'est le point de démarrage de n'importe quelle application C ou Objective-C.

Nous allons pouvoir mettre à l'épreuve vos connaissances du langage Objective-C. Supprimez dans le corps de la fonction la ligne de journalisation basée sur **NSLog**. Remplacez-la par les lignes proposées ci-après. Elles permettent de créer puis de détruire une instance de la classe standard Objective-C nommée **NSMutableArray** :

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
  @autoreleasepool {

    // insert code here...
    NSLog(@"Hello, World!");

    // Création d'un objet tableau muable et stockage de son adresse dans items
    NSMutableArray *items = [[NSMutableArray alloc] init];

    // Destruction du tableau pointé par items
    items = nil;
  }
  return 0;
}
```

Puisque nous disposons maintenant d'une instance de **NSMutableArray**, nous pouvons lui envoyer des messages, par exemple **addObject:** ou **insertObject:atIndex:**. Le récepteur des messages va être la variable items, qui pointe sur notre objet **NSMutableArray** venant d'être créé. Insérons quelques chaînes de caractères dans cette instance de tableau :

```
int main (int argc, const char * argv[])
{
  @autoreleasepool {
    // Create a mutable array object, store its address in items variable
    NSMutableArray *items = [[NSMutableArray alloc] init];

    // Envoi du message addObject: au tableau NSMutableArray pointé par
    // la variable items, en transmettant une chaîne à chaque fois.
    [items addObject:@"Un"];
    [items addObject:@"Deux"];
    [items addObject:@"Trois"];
```

```
    // Envoi d'un autre message, insertObject:atIndex:, au même tableau
    [items insertObject:@"Zéro" atIndex:0];

    items = nil;
  }
  return 0;
}
```

Lorsque cette application s'exécutera, elle créera un tableau **`NSMutableArray`** puis le remplira avec quatre instances de chaîne **`NSString`** (une classe standard Objective-C). Nous allons pouvoir vérifier que les chaînes ont bien été insérées dans le tableau. Toujours dans `main.m`, après avoir ajouté le dernier objet au tableau, nous allons insérer une boucle pour balayer le tableau et afficher le contenu de chaque chaîne sur la console :

```
int main (int argc, const char * argv[])
{
  @autoreleasepool {
    // Création d'un objet tableau muable et stockage de son adresse dans items
    NSMutableArray *items = [[NSMutableArray alloc] init];

    // Envoi du message addObject: au tableau NSMutableArray pointé par
    // la variable items, en transmettant une chaîne à chaque fois.
    [items addObject:@"Un"];
    [items addObject:@"Deux"];
    [items addObject:@"Trois"];

    // Envoi d'un autre message, insertObject:atIndex:, au même tableau
    [items insertObject:@"Zéro" atIndex:0];
    // Pour chaque élément item du tableau d'après le nombre count envoyé à items...
    for (int i = 0; i < [items count]; i++) {
      // nous récupérons le ième objet du tableau et l'envoyons en argument à
      // NSLog, qui par défaut affiche le message description à l'objet
      NSLog(@"%@", [items objectAtIndex:i]);
    }
  items = nil;
  }
  return 0;
}
```

Nous reviendrons dans quelques instants sur les points de syntaxe remarquables dans le code précédent. Pour l'instant, essayons le résultat au moyen du bouton Run. Vous pourriez croire qu'il ne s'est rien passé puisque le programme s'est arrêté. Pourtant, le navigateur de journal (Log Navigator) nous donne quelques informations.

Pour rendre visible ce navigateur, utilisez l'icône ( ) ou le raccourci clavier Commande-7. Ce navigateur de journal affiche tous les résultats de la génération et tous les affichages de console qui ont été produits par l'application. En haut de la fenêtre du navigateur de journal, sélectionnez Debug RandomPossessions pour afficher ce qui a été envoyé à la console dans la zone de l'éditeur (voir Figure 2.6).

Défis

La plupart des chapitres du livre se terminent par au moins un défi qui vous invite à aller plus loin dans ce qui a été présenté au cours du chapitre, afin de vérifier que vous avez bien assimilé son contenu. Nous vous conseillons de tenter de relever le plus de défis possible pour consolider vos connaissances, en sorte de passer de l'apprentissage d'iOS au *développement concret* sous iOS et de devenir autonome.

Les défis sont proposés en trois niveaux de difficultés :

- Défis de bronze : ces défis vous demandent de prolonger un peu ce que vous avez pratiqué dans le chapitre. Ils vous permettent de renforcer vos connaissances en codant des instructions proches de celles déjà vues, mais sans les avoir face à vous. C'est en forgeant que l'on devient forgeron.
- Défis d'argent : ce niveau de défi vous demande de faire un minimum de recherches et de réfléchir au problème. Vous devrez savoir trouver et utiliser des méthodes, des classes et des propriétés qui n'ont pas été citées dans le chapitre. Cependant, la problématique reste la même que dans l'exemple.
- Défis d'or : ce niveau de défi est ardu et peut prendre plusieurs heures. Il suppose que vous ayez compris les concepts et que vous sachiez réfléchir à la modélisation du problème de façon autonome. En réussissant ce niveau de défi, vous vous préparez à devenir un développeur iOS opérationnel.

Une mise en garde essentielle : avant de commencer n'importe quel défi, *faites toujours une copie du dossier du projet dans le Finder et travaillez à partir de la copie pour réaliser le défi.* En effet, nombreux sont les chapitres qui dépendent de l'état correct du projet à la fin du chapitre précédent. Si vous faites des modifications dans un défi, vous mettez en péril ce passage de relais entre chapitres.

Défi de bronze : trouver un bogue

Créez volontairement un bogue dans le programme en demandant d'accéder au onzième élément du tableau. Lancez l'exécution et notez l'exception qui est déclenchée.

Défi d'argent : un autre initialiseur

Définissez une autre méthode initialiseur pour l'objet **`BNRItem`**. Elle ne sera *pas* l'initialiseur désigné. La méthode attendra en entrée un objet **`NSString`** pour identifier le nom de l'élément `itemName` et un autre objet **`NSString`** pour le numéro de série `serialNumber`.

Défi d'or : une autre classe

Créez à partir de **`BNRItem`** une sous-classe nommée **`BNRContainer`** qui possédera un tableau d'éléments `subitems` correspondant à des instances de **`BNRItem`**. L'affichage de la description de **`BNRContainer`** doit donner le nom du conteneur, la valeur monétaire (qui est la somme de tous les éléments du conteneur plus la valeur du conteneur lui-même) et la liste de tous les éléments **`BNRItem`** trouvés dans le contenu. Si la classe **`BNRContainer`** est bien conçue, elle pourra contenir des instances de **`BNRContainer`**. Elle doit pouvoir renvoyer la valeur totale et celle de chaque élément du contenu sous un format lisible.

Pour les plus curieux

De nombreux chapitres comprennent après la section des défis une ou plusieurs sections destinées aux personnes qui veulent en savoir un peu plus. Nous y rassemblons des descriptions détaillées qui viennent en complément des concepts présentés dans le chapitre. Le contenu de ces sections n'est pas indispensable pour poursuivre la lecture du livre, mais nous sommes certains que vous y trouverez des éléments intéressants.

Pour les plus curieux : noms de classes

Les applications minimales comme **`RandomPossessions`** ne définissent qu'une poignée de classes, mais les applications plus complexes voient ce nombre augmenter fortement. Vous finirez par aboutir à une situation dans laquelle vous pourriez donner le même nom à deux classes différentes, ce qui est une mauvaise chose. Lorsqu'il y a deux classes homonymes, le programme ne peut pas savoir laquelle il doit utiliser. Cela correspond à une *collision d'espaces de noms*.

Dans d'autres langages, le problème est évité par la création d'espaces de noms dans lesquels les classes sont définies. Un *espace de noms* est un groupe qui englobe un ensemble de classes. Pour pouvoir utiliser une classe, il faut dans ce cas spécifier l'espace de noms en préfixe du nom de la classe.

Le concept d'espace de noms n'existe pas en Objective-C. Cependant, nous avons par convention l'habitude d'ajouter un préfixe sur deux ou trois lettres à tous nos noms de classes. C'est pourquoi nous avions choisi le nom **`BNRItem`** au lieu de **`Item`** dans ce chapitre (BNR étant l'acronyme de l'éditeur américain de ce livre, *Big Nerd Ranch*).

Les bons programmeurs Objective-C ajoutent toujours un préfixe aux classes de leurs modèles et de leurs vues. En général, le préfixe a un rapport avec le nom du projet ou de la bibliothèque. Si j'écris par exemple une application nommée "**`MelangeurVideo`**", toutes mes classes commenceraient par **`MV`**. Les classes que vous utilisez d'un projet à un autre porteront un préfixe suggestif, comme vos initiales, le nom de votre entreprise (**`BNR`**) ou l'abréviation d'une bibliothèque (par exemple, une bibliothèque de gestion de cartographie pourrait avoir le préfixe **`MAP)`**.

En revanche, les objets contrôleurs ne sont que rarement réutilisables d'une application à une autre. Ils n'ont donc pas besoin de préfixe. Notez que ce n'est pas une règle stricte. Si vous le désirez, vous pouvez préfixer les noms de vos objets contrôleurs, d'autant plus si vous comptez malgré tout leur trouver d'autres emplois.

Les classes prédéfinies par Apple portent elles aussi des préfixes. Elles sont classées en plusieurs infrastructures (nous reverrons cela au Chapitre 4) et chaque infrastructure possède un préfixe. C'est pourquoi la classe nommée **`UILabel`** tire son nom du fait qu'elle est définie dans l'infrastructure *UIKit*. De même, les classes **`NSArray`** et **`NSString`** appartiennent à l'infrastructure *Foundation*. Sachez que le **`NS`** provient de NeXTSTEP, qui était la plateforme pour laquelle ces classes avaient été conçues à l'origine, avant que Steve Jobs revienne chez Apple.