MCours.com

1

Les bases d'Objective-C

Objective-C est un langage de programmation orienté objet basé sur le langage C, tout comme C++. Cette spécificité intéressante rend le langage compatible avec C et dans une certaine mesure avec C++. Objective-C est également un langage au typage dynamique, contrairement à C++. Il dispose en conséquence d'un environnement d'exécution dédié, similaire par exemple à Java ou C#, mais toutefois plus réduit en termes de compétences. De plus, contrairement à Java et C#, Objective-C est compilé directement en langage machine et non en langage intermédiaire.

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

Dans ce chapitre, nous abordons ce qui est indispensable : les éléments, les mots-clés et les concepts du langage. Chaque section sera étayée de plusieurs exemples.

Définition de id

```
id number = [NSNumber numberWithInt:5];
```

Le code ci-dessus montre qu'un objet est déclaré sans typage avec le mot-clé `id`.

Objective-C est un langage dynamique, mais il autorise une vérification des références lors de la compilation. Ces notions seront abordées plus loin, mais il est toutefois important de remarquer qu'Objective-C se rapproche davantage de Python, Ruby et des autres langages dynamiques que de Java ou C# dans ce domaine.

Objective-C permet d'introduire un objet sans pour autant préciser son type grâce au mot-clé `id` qui peut être lu "identifiant d'objet" et ne force aucun typage. `id` est donc un pointeur vers un objet. Plus précisément, `id` est défini comme un pointeur vers la structure de données de l'objet.

Voici un exemple pour bien se rendre compte de ce fait – vous pouvez ignorer pour l'instant le reste de la syntaxe qui est toutefois simple à comprendre – les deux méthodes, `exemple1` et `exemple2` sont toutes les deux de type `id`, mais renvoient deux objets de type différents :

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

Exemple 1

```
exemple1() {
   //le type renvoyé n'est pas spécifié
   //il n'y a pas d'avertissement du compilateur
   //pourtant la méthode renvoie un 'id' de
   //type NSNumber
   id number = [NSNumber numberWithInt:5];
   return number;
}
```

Exemple 2

```
exemple2(){
    id date = [NSDate date];
    return date;
}
```

Autre point important à noter : `id` est le type par défaut de la valeur retournée par n'importe quelle méthode Objective-C. Ceci est logique dans la construction d'Objective-C, mais diffère du type par défaut retourné par les fonctions en C où le type retourné par défaut est `int`.

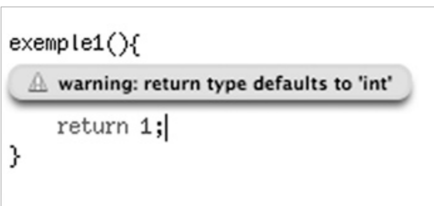

Figure 1.1 : Avertissement du compilateur, qui signale que le type par défaut de la fonction sera `int`.

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

Pour aller plus loin

Objective-C étant construit sur les fondations de C, il est possible de creuser un peu les éléments qui forment le cœur du langage. Par exemple, `id` est tout simplement défini comme un pointeur vers une structure `struct`, comme précisé dans le fichier d'en-tête `/Developer/SDKs/MacOSX10.5.sdk/usr/include/objc/objc.h` **:**

```
typedef struct objc_object {
    Class isa;
} *id;
```

Objet-classe

```
Class uneClasse = [unObjet class];
```

À la différence de la plupart des langages, une classe Objective-C est également un objet, mais un objet de type particulier. C'est ce que l'on appelle un *objet-classe*. C'est grâce à cette particularité qu'il est possible d'utiliser un nom de classe comme type lorsqu'un objet est déclaré.

De plus, Objective-C définit également un type particulier pour les objets-classes : `Class`, qui n'est autre qu'un pointeur vers une structure opaque de type `objc_class`.Vous trouverez ces définitions dans

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

le fichier d'en-tête `/Developer/SDKs/MacOSX10.5.sdk/usr/include/objc/objc.h` :

```
typedef struct objc_class *Class;
```

De plus, nous pouvons clarifier davantage la définition de `id` donnée à la section précédente :

```
typedef struct objc_object {
    Class isa;
} *id;
```

`id` est un pointeur vers une structure opaque de type `objc_object` (qui définit la structure des objets Objective-C). Chaque objet possède donc ce que l'on appelle un pointeur `isa` qui pointe vers un objet de type `Class`. Comme vous l'avez sans doute deviné maintenant, chaque objet pointe donc *via* son pointeur `isa` vers son objet-classe.

Autre point intéressant à noter : `id` peut donc également représenter un objet-classe.

Déclarer un objet

```
id declarationTypageDynamique;
ClasseGuideDeSurvie *declarationTypageStatique;
```

Il existe deux sortes de déclarations en Objective-C :

- Le typage statique. On précise le type de l'objet, c'est-à-dire la classe à laquelle appartient l'objet,

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

au moment de la déclaration. Le compilateur s'assure que ce type est respecté tout au long du code.

- **Le typage dynamique.** On déclare l'objet sans préciser son type, grâce au pointeur `id`. Le compilateur ne possède alors aucune connaissance sur le type de la référence, qui peut désormais représenter une instance de n'importe quelle classe.

Dans la grande majorité des cas, le type de l'objet est connu par le développeur et n'a pas de raison de changer au cours de l'exécution du programme – ou changera, mais restera dans la même hiérarchie de classe. Il est alors préférable d'utiliser le typage statique et de disposer ainsi de l'aide du compilateur afin réduire les risques de bogues.

Toutefois, il est possible que la référence soit amenée à pointer vers différentes classes, il faudra alors utiliser le typage dynamique. Auquel cas le compilateur ne pourra pas réaliser l'ensemble des vérifications qu'il fait lors du typage statique, et vous serez exposé à des erreurs lors de l'exécution du code (ce que l'on appelle un crash).

Le rôle du pointeur `isa`

Comme nous l'avons vu dans les deux sections précédentes, les objets sont typés de manière dynamique. Ceci signifie que le compilateur ne sait pas

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

MCours.com

forcément à quel type d'objet il a affaire. Il faut donc fournir à l'environnement d'exécution un moyen d'obtenir l'information. C'est là qu'intervient `isa` : par définition, chaque objet pointe *via* `isa` vers son objet-classe.

Toutefois, il est important de noter que vous n'emploierez jamais le pointeur `isa` directement. Il est utilisé exclusivement par l'environnement d'exécution. En revanche, vous pourrez envoyer le message `class` (voir section "Envoyer un message") à tout objet afin d'obtenir son objet-classe, du type `Class` :

```
ClasseGuideDeSurvie * instance =
➥ [[ClasseGuideDeSurvie alloc] init];
id MaClasse = [instance class];
Class ToujourMaClasse = [instance class];
NSLog(@"MaClasse : %@",  MaClasse);
NSLog(@"ToujoursMaClasse : %@",  ToujourMaClasse);
```

Vous devriez voir sur la console le résultat suivant :

```
GuideDeSurvie[6981:10b] MaClasse :
➥ ClasseGuideDeSurvie
GuideDeSurvie[6981:10b] ToujoursMaClasse :
➥ ClasseGuideDeSurvie
```

Veuillez noter qu'il est possible d'utiliser le type `id` ou `Class`, mais qu'il n'est pas possible d'utiliser le nom de la classe, puisque cette syntaxe est utilisée pour typer la définition des instances de ces mêmes

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

classes. Voici un exemple pour clarifier. La syntaxe de la Figure 1.2 est incorrecte :

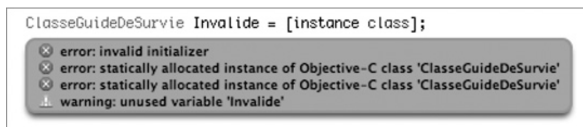

Figure 1.2 : Le compilateur génére une erreur si vous essayer d'utiliser le nom de la classe comme type de retour du message `class`.

La classe racine

Une classe racine est une classe ne possédant aucun parent dans la hiérarchie des classes et dont dérivent (directement ou indirectement) toutes les autres classes.

L'implémentation de l'environnement d'exécution d'Objective-C et des bibliothèques de base fournies par Apple contient la définition de la classe `NSObject` comme classe racine.

`NSObject` est donc l'équivalent en Objective-C de `Java.lang.Object` ou de `Object` en C#. Toute classe hérite de `NSObject`, ce qui est une bonne chose car vous n'avez ainsi pas à vous préoccuper d'écrire tout le code nécessaire à la bonne intégration de vos classes dans l'environnement d'exécution d'Objective-C. Il est en théorie possible d'écrire sa propre classe racine, mais en pratique, c'est une tâche extrêmement difficile, périlleuse et – soyons réaliste – inutile.

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

`NSObject` fournit une implémentation minimale de plusieurs méthodes que vous serez amené à utiliser très souvent, notamment :

```
+ (void)initialize;
- (id)init;
+ (id)new;
+ (id)alloc;
- (void)dealloc;
- (id)copy;
- (id)mutableCopy;
+ (NSString *)description;
```

Pour aller plus loin

Vous avez sans doute déjà bien compris l'idée principale de cette section : toutes les classes en Objective-C sur plateforme Apple dérivent de `NSObject`, qui est la classe racine.

En fait, ce n'est pas tout à fait exact. Comme nous l'avons vu, il est possible de définir plusieurs classes racines et, en de très rares occasions, ceci devient même une nécessité. C'est le cas par exemple de la classe `NSProxy`, qui est une classe racine, mais qui implémente le protocole `NSObject`.

La raison est tout à fait logique : le rôle de `NSProxy` est de fournir l'infrastructure nécessaire à la création d'objets distants (ainsi que d'autres utilisations en relation avec les objets distants et les systèmes distribués).

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

Différence entre `id` et `NSObject`

```
id objetDeclareAvecId; // cas 1
NSObject *objetDeclareAvecNSObject; // cas 2
```

Étant donné que `id` est un pointeur vers un objet et que toutes les classes en Objective-C héritent de `NSObject`, il est facile d'arriver à des conclusions trop hâtives. Nous allons donc dissiper toute confusion et profiter de cette opportunité pour faire un point intéressant et récapituler les cinq sections précédentes.

Prenons donc les deux cas précédents :

- Le cas 1 est le cas le plus courant. Nous avons vu précédemment que nous déclarons ici simplement un objet sans préciser son type. Le compilateur ne sait pas à quel type d'objet il est confronté. De plus, comme le typage dynamique le permet, il autorise l'envoi de n'importe quel message sans essayer d'appliquer la moindre contrainte. Les erreurs seront découvertes lors de l'exécution par l'environnement d'exécution.
- Dans le cas 2, nous déclarons un pointeur vers un objet de type `NSObject`. Il y a tout d'abord une petite différence syntaxique : nous utilisons l'étoile (`*`) qui n'est pas nécessaire avec `id`. De plus, ici, l'objet est nécessairement de type `NSObject`. Il est peut-être intéressant de rappeler ici que tous les objets n'héritent pas nécessairement de `NSObject` (voir la note "Pour aller plus

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

loin” de la section précédente). Cette situation ne devrait se produire que très rarement et dans des cas particuliers que vous aurez soigneusement délimités. Mais, ici, le compilateur va procéder à une analyse statique des messages envoyés à l’objet, et générer un avertissement pour chaque message auquel `NSObject` ne répond pas.

En conclusion, le cas 2 est celui utilisé par les développeurs Java et C# par exemple, mais étant donné qu’Objective-C dispose d’une syntaxe dédiée pour le typage dynamique, c’est cette syntaxe qui devra être utilisée dans notre code Objective-C.

nil, Nil et NULL

Le mot-clé `nil` désigne un objet nul, tandis que le mot-clé `Nil` désigne un objet-classe nul. La capitalisation de la première lettre permet de différencier l’instance nulle d’une classe de l’objet-classe nul. Il est intéressant de noter qu’il est équivalent de dire qu’un objet (ou un objet-classe) est nul ou que c’est une référence vers `nil` ou que la valeur du pointeur `id` est 0. En effet, il est alors possible de tester la valeur des pointeurs pour s’assurer que l’objet n’est pas nul et donc qu’il est possible de lui envoyer un message (d’invoquer une méthode).

`NULL` est une définition utilisée dans les API C/C++ et souvent précisée par une directive compilateur. Il n’est pas utilisé en Objective-C, sauf lors de l’utilisation de bibliothèques écrites en C ou C++.

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui

Comme nous l'avons vu précédemment, une classe Objective-C n'est autre qu'un type particulier d'objet appelé objet-classe.

`nil` et `Nil` sont tous deux définis dans le fichier `/Developer/SDKs/MacOSX10.5.sdk/usr/include/objc/objc.h` de la manière suivante :

```
#ifndef Nil
#define Nil __DARWIN_NULL /* id of Nil class */
#endif
#ifndef nil
#define nil __DARWIN_NULL /* id of Nil instance */
#endif
```

D'après cette définition, il est clair que `Nil` désigne un objet-classe nul, alors que `nil` désigne un instance de classe `NULL`. Vous pouvez aussi remarquer que `nil` et `Nil` représentent deux concepts distincts, mais que derrière les concepts se cache la même valeur : `__DARWIN_NULL`.

Il est alors possible de creuser encore un peu plus et chercher ce que signifie `__DARWIN_NULL`. Sa définition se trouve dans le fichier `/Developer/SDKs/MacOSX10.5.sdk/usr/include/sys/_types.h` :

```
#define __DARWIN_NULL ((void *)0)
```

Nous savons désormais que `__DARWIN_NULL` n'est donc qu'une redéfinition du pointeur nul de C.

MCours.com

© 2010 Pearson Education France – Objective-C 2.0 – Pejvan Beigui