Memento de Swift

ghislain Oudinet

Table des matières

Chapitre 1

Conventions de codage

Philosophie générale

L'affectation ne renvoie pas la valeur affectée, mais le type vide **Void**. Les affectations ne peuvent donc pas être utilisées comme en *C* pour des comparaisons implicites à la non nullité.

Swift sait manipuler variables et constantes, ces dernières étant des variables dont le contenu ne peut être affecté qu'une première fois et plus modifié par la suite.

Le typage des variables n'est pas obligatoire car il peut souvent être déduit du contenu qui lui est affecté. Par défaut, l'inférence de type favorise l'**Int** pour les entiers et le **Double** pour les flottants.

Swift prend en compte qu'un bloc de code puisse être quitté par un **return**, un **break** ou une exception. Il propose donc la clause **defer** qui introduit un bloc dont l'exécution sera repoussé à la sortie du bloc de code englobant. Les clauses **defer** sont exécutées dans l'ordre inverse de leur déclaration.

Les variables sont initialisées par défaut.

L'accès aux cases d'un tableau est vérifié avant accès. Les débordements de valeur sont aussi suivis par le système (l'arithmétique en modulo doit être *explicitement* spécifiée).

La mémoire désallouée ne peut plus être utilisée avec la même référence.

Le compilateur analyse les accès mémoire pour détecter ceux qui peuvent être non sûrs car concurrentiels.

Le langage est basé sur des *modules*, qui sont des unités de code que l'on peut *importer* dans une autre unité de code.

Conventions syntaxiques

Les blocs de code doivent obligatoirement être encadrés par des accolades, même s'ils ne contiennent qu'une seule ligne de code.

Le point-virgule à la fin d'une instruction n'est pas obligatoire, il n'est seulement pour séparer plusieurs instructions sur une même ligne.

Plusieurs variables peuvent être déclarées sur une même ligne, séparées par des virgules.

Le type d'une variable est seulement annoté, c'est-à-dire indiqué après le ou les noms de variable et après le symbole **:**.

Les noms des variables peuvent contenir n'importe quel caractère *Unicode*, mais ne peuvent contenir ni espace, ni flèche, ni symbole mathématique, et ne peuvent débuter par un chiffre. Une fois déclarée, une variable ne peut être redéclarée avec le même ou un autre type, ou passer de variable à constante ou vice-versa.

Une variable ne devrait pas être nommée à l'identique d'un mot-clé du langage *Swift*, cependant c'est possible, en mettant le nom de la variable entre apostrophe inversées `.

Les littéraux numériques peuvent utiliser le symbole _ dans le flot de leurs chiffres pour améliorer la lisibilité des valeurs avec beaucoup de chiffres.

Commentaires

Les commentaires monoligne sont introduits par deux symboles de division successifs `//`.

Les commentaires multilignes sont introduits par les symboles ouvrants `/*` et terminés par les symboles fermants `*/`. Des commentaires multilignes peuvent être inclus dans d'autres commentaires multilignes, au contraire du *C*.

Chapitre 2

Données et types de données

Les noms de types de données de base commencent par une majuscule.
var : sert à déclarer les variables
let : sert à déclarer les constantes

L'opérateur **is** permet de tester si une donnée est d'un type particulier (incluant le test des super-classes), et renvoie vrai ou faux. L'analyse statique de type rend cet opérateur seulement utile lors d'un héritage.

L'opérateur **as** permet de convertir une instance dans un type différent de super-classe. Il peut être utilisé sous les formes **as ?** et **as !**.

Les définitions de type peuvent être imbriquées, elles seront alors liées à leur contexte de définition. On peut accéder explicitement à ces définitions à l'aide de l'opérateur **.**.

Liste des types de base du langage

Bool : booléen (**true** et **false** seules valeurs possibles)
Int : entier signé taille machine
Int8 : entier 8 bits signé
Int16 : entier 16 bits signé
Int32 : entier 32 bits signé
Int64 : entier 64 bits signé
UInt : entier non-signé taille machine
UInt8 : entier 8 bits non-signé
UInt16 : entier 16 bits non-signé
UInt32 : entier 32 bits non-signé
UInt64 : entier 64 bits non-signé
Float : flottant simple précision
Double : flottant double précision
Float80 : flottant 80 bits
Character : caractère
String : chaîne de caractères
[**Type**] **:** tableau
Array<Type> : tableau
[**KeyType : ValueType**] **:** dictionnaire
Dictionary<KeyType, ValueType> : dictionnaire
() : t-uplet
Set<Type> : ensemble
enum : mot-clé introduisant une énumération
nil : valeur nulle pour un pointeur
Any : n'importer quel type d'instance de quoi ue ce soit (fonctions comprises)
AnyObjet : n'importe quel type d'instance de classe

t-uplets

Les t-uplets sont utilisés pour grouper des valeurs. Ils peuvent être vus comme des structures (de préférence temporaires et simples) aux champs anonymes (mais typés), sont introduits par des parenthèses et les champs séparés par des virgules. L'ordre des champs est donc important, les champs peuvent d'ailleurs être accédés par leur numéro (en partant de 0, et après l'opérateur d'accès **.**).

Un t-uplet peut aussi être déclaré avec des noms pour chaque champ ; chacun de ces champs de ce t-uplet particulier est alors accessible par le nom correspondant. Lorsque le nom de champ est utilisé lors de la création d'une valeur d'initialisation, on peut affecter à ce champ une valeur avec l'opérateur **:**.

Type *optionnel*

Le type *optionnel* indique l'absence potentielle de valeur, il peut être utilisé avec tout type de données (et pas seulement un pointeur sur un objet comme en *Objective-C*). Ce modificateur de types se fait en ajoutant le symbole **?** après le type dont la présence est optionnelle.

Un type optionnel qui ne contient aucune valeur contient la valeur particulière **nil**.

Si un type optionnel est connu comme contenant une valeur (par exemple après comparaison à **nil**), l'ajout du symbole **!** après le nom force l'utilisation de cette valeur.

Création de nouveaux types

De nouveaux types peuvent référencer d'anciens types avec le mot-clé **typealias**, où le nom du nouveau type est suivi du symbole = puis de l'ancien nom de type.

Spécificités des entiers

Les types (mais pas les variables) entiers signés et non-signés possèdent les propriétés **.min** et **.max** qui fournissent les plus petite et plus grande valeurs accessibles.

Les entiers possèdent un initialiseur de conversion utilisant le label **exactly**, qui renvoie un type optionnel car la conversion a pu ne pas être exacte.

Spécificités des types composés (*String*, *Array*, *Set* et *Dictionary*)

Ces types possèdent la propriété **count** qui indique le nombre d'éléments, ainsi que la propriété **isEmpty** qui indique si l'objet est vide ou pas.

Le type **RangeReplaceableCollection** possède les méthodes **insert(_ :at :)** et **insert(contentsOf :at :)**, ainsi que les méthodes **remove(at :)**, **removeLast()** et **removeSubrange(_ :)** permettant d'insérer ou d'enlever éléments ou séquences d'éléments.

Le type **Collection** possède les méthodes **startIndex**, **endIndex**, **index(_ : offsetBy :)**, **index(before :)** et **index(after :)** permettant de parcourir **String**, **Array**, **Dictionary** et **Set**.

Spécificités des chaînes de caractères

Les chaînes sont des types passés par valeur (bien que le compilateur se permette des optimisations si possible).

Le type **String** possède la propriété **.indices** qui renvoie un intervalle permettant de parcourir tous les caractères d'une chaîne.

Le type **String** possède les propriétés **.utf8**, **.utf16** et **.unicodeScalars** pour récupérer une forme particulière de la chaîne en question.

Une **String** possède les méthodes **hasPrefix(_ :)** et **hasSuffix(_ :)** pour tester si une chaîne démarre ou finit par une chaîne donnée.

Un tableau de caractères doit être converti en chaîne de caractères si approprié, car ce n'est pas un objet chaîne.

Les sous-chaînes font référence à leur chaîne source, ainsi une fois qu'elles ne sont plus utiles il faut les convertir en **String**.

Une chaîne peut contenir des valeurs variables, placées entre parenthèses dont la parenthèse ouvrante est précédée du caractère d'échappement \.

Spécificités des tableaux

Le type **Array** propose le constructeur non-paramétré, ainsi qu'un constructeur **(repeating : count :)** pour créer des valeurs par défaut par répétition. On peut aussi créer un tableau en listant les valeurs de départ entre crochets.

Spécificités des ensembles

Le type **Set** possède la méthode **insert(_ :)**, ainsi que la méthode **removeAll()**. Il peut être initialisé à partir d'un tableau.

Le type **Set** possède les méthodes **intersection(_ :)**, **union(_ :)**, **substracting(_ :)** et **symmetricDifference(_ :)**. Il possède aussi les méthodes **is[Strict]Subset(of :)**, **is[Strict]Superset(of :)** et **isDisjoint(with :)**.

Spécificités des dictionnaires

Le type **Dictionary** possède la méthode **updateValue(_ : forKey :)**, qui agit comme l'opérateur d'accès, mais qui renvoie l'ancienne valeur associée à la clé si celle-ci existait avant l'appel. La même fonction permet donc aussi de créer une nouvelle entrée et pas seulement de la mettre à jour, car son type de retour est un type *optionnel* du type de la valeur.

Le type **Dictionary** possède aussi la méthode **removeValue(forKey :)**.

Spécificités des énumérations

Les énumérations sont des types passés par valeur (bien que le compilateur se permette des optimisations si possible).

Une énumération est définie entre accolades, et chaque valeur introduite par un **case**. Plusieurs valeurs peuvent être introduites par un seul **case**, séparées alors par des virgules.

Une énumération définit un nouveau type de donnée. Une fois qu'une variable est d'un type connu d'énumération, il est possible de lui affecter de nouvelles valeurs sans nommer de nouveau l'énumération, mais simplement en utilisant la notation pointée pour désigner la valeur voulue.

Chaque **case** d'énumération peut être associé à une *valeur brute*, qui lui est affectée lors de la définition du type. L'énumération doit alors être typée. Si un premier **case** est affecté d'une valeur, les suivants se voient automatiquement affectés les valeurs consécutives. Si une énumération est typée avec **String**, la valeur de la chaîne de caractères correspondant au nom est automatiquement affectée au **case**. La valeur brute peut être lue avec la propriété **.rawValue** du **case**. On peut créer une valeur d'énumération à partir de sa valeur brute en utilisant le constructeur du type, c'est-à-dire le nom du type suivi entre parenthèses de **rawValue :** et de la valeur brute. Ce constructeur renvoie une valeur optionnelle.

Une énumération indirecte est une énumération pour laquelle au moins un des **case** fait référence à l'énumération. Ce **case** est alors précédé de **indirect**, ou bien si tous les **case** sont indirects, on peut mettre **indirect** devant **enum**.

Un cas d'énumération peut être défini comme ayant une *valeur associée* : le type de cette valeur est alors défini par un t-uplet juste après le nom de la valeur. L'affection d'une valeur d'énumération à une variable se fait en plaçant un t-uplet de valeurs correspondant aux types après la valeur de l'énumération, et la lecture de ces valeurs se fait à l'aide d'un **switch**, où chaque **case** est suffixé d'un t-uplet de variables (ou constantes), avec une variable par type composant la valeur associée. S'il y a plusieurs variables dans la valeur associée, le **var** ou **let** peut être placé juste après le **case** et omis dans les parenthèses du t-uplet.

L'initialisation d'une énumération à partir d'une valeur brute (à l'aide du label **rawValue**) renvoie un optionnel.

Conversions de type

Les conversions de type se font à l'aide d'une écriture fonctionnelle, dont le nom de la fonction est le nom du type cible.

Certaines conversions implicites sont autorisées en *Swift*, mais les conversions vers **Bool** sont obligatoirement explicites.

Chapitre 3

Valeurs littérales

Littéraux caractères/chaînes de caractères

Les littéraux chaînes de caractères sont entourées de guillemets simples (").

Les littéraux chaînes de caractères multilignes démarrent et se terminent par de triples guillemets simples (""""). Les retours à la ligne font alors partie de la chaîne, sauf le premier après les triples guillemets et le dernier avant les triples guillemets, à moins que la fin de la ligne soit signalée par un symbole \.

L'indentation devant les trois guillemets fermants sera prise comme référence pour ignorer les indentations identiques dans les lignes précédentes. Placer un caractère d'échappement devant un ou plusieurs guillemets permet de considérer le ou les guillemets comme des caractères affichables, et pas comme une fin de chaîne.

Un caractère *Unicode* peut être explicitement généré par la séquence d'échappement **\u** suivie d'une valeur numérique encadrée par des accolades.

Littéraux entiers

Une préfixe **0b** indique une valeur littérale binaire, un préfixe **0o** une valeur littérale octale et un préfixe **0x** une valeur littérale hexadécimale.

Littéraux flottants

Les littéraux flottants décimaux peuvent utiliser un exposant avec le symbole **e**.

Les littéraux flottants hexadécimaux (introduits par **0x**) doivent utiliser un exposant, introduit avec le symbole **p**.

Chapitre 4

Opérateurs

Opérateurs unaires : + -
Opérateurs arithmétiques : + - * / %
Opérateurs bit à bit : ~ & | ^
Opérateurs de décalage : « »
Opérateurs d'affectation : = += -= *= /= %=
Opérateurs de comparaison : == != < > <= >=
Opérateurs de test d'identité : === !==
Opérateurs booléens : && || !
Opérateurs d'intervalles : ..< ...
Opérateur ternaire : ? :
Opérateurs de concaténation de chaînes de caractères : + +=
Opérateur de coalescence : ? ?
Opérateur d'accès/d'indexation : [ ]
Opérateurs avec overflow : &+ &- &* &/ &%

Spécificites des opérateurs de décalage

Le décalage à droite est par défaut arithmétique sur des nombres signés, logique sur des nombres non signés.

Spécificités des opérateurs d'affectation

Les opérateurs d'affectation ne renvoient aucune valeur (évite les confusions avec == pour =).

Spécificités des opérateurs booléens

Les opérateurs booléens agissent par court-circuit, comme en *C*, et sont associatifs à gauche.

Spécificités des opérateurs arithmétiques

L'opérateur % n'est pas vraiment un opérateur modulo au sens mathématique du terme, mais plutôt un opérateur *reste de la division* (la différence est importante pour les nombres négatifs).

Par défaut les opérateurs arithmétiques en *Swift* n'autorisent pas les *overflow* qui sont monnaie courante avec les arithmétiques à module (comme en *C*). Il existe donc des opérateurs explicites les autorisant. Ces opérateurs sont les opérateurs classiques mais précédés par une esperluette.

Spécificités des opérateurs de comparaison

Les opérateurs de comparaison peuvent aussi comparer des t-uplets, de gauche à droite, à condition que ceux-ci aient le même nombre de champs et qu'ils soient de type identique, et que le nombre de champs soit inférieur à 7.

Spécificités de l'opérateur de coalescence

L'opérateur de coalescence a **? ?** b s'applique sur un optionnel *a*, et équivaut à : a **!= nil ?** a **! :** b. *b* n'est pas évalué si *a* n'est pas **nil**.

Spécificités des opérateurs d'intervalle

Pour les opérateurs d'intervalle, l'opérande gauche doit être plus petit ou égal à l'opérande droite.

Les opérateurs d'intervalle permettent d'omettre la première ou la seconde borne, à condition qu'il n'y ait pas d'ambiguïté sur le point de départ ou d'arrivée (par exemple un intervalle d'entiers, ou un indice de tableau).

Spécificités des opérateurs de concaténation

L'opérateur de concaténation composé de chaîne de caractères peut être remplacé par la méthode **append(_ :)**.

Spécificités des opérateurs d'accès

L'opérateur d'accès [ ] permet d'accéder aux cases d'un tableau comme aux clés d'un dictionnaire.

Dans le cas d'un dictionnaire c'est un type *optionnel* qui est renvoyé, car la clé demandée peut ne pas exister. Si l'on est sûr que la clé existe, on peut utiliser le symbole **!** après la valeur indicée afin de forcer l'utilisation directe de la valeur. On peut dans le cas du **Dictionary** retirer une clé en lui affectant **nil**.

Chapitre 5

Fermetures et fonctions

Les fonctions en *Swift* sont des fermetures nommées. Les fonctions globales ne capturent aucune valeur.

Les fonctions sont introduites par le mot-clé **func** et prennent leurs paramètres (typés comme les variables) entre parenthèses.

Le type de retour, s'il existe, est précisé après la parenthèse fermante des paramètres, et après une flèche **->**. S'il n'existe pas, **Void** est implicitement utilisé et consiste en un t-uplet vide **()**.

Le retour de valeurs multiples se fait en retournant un t-uplet. Ce t-uplet peut être *optionnel*.

Le type fonction est constitué d'un t-uplet des paramètres, suivi de la flèche puis du type de retour.

Les fonctions peuvent être surchargées. Le nombre et le type de paramètres peuvent ainsi varier pour un même identifiant de fonction. *Swift* autorise à surcharger avec le même nombre et le même type de paramètres, à condition que les labels des paramètres soient différents.

Spécificités des paramètres et du passage de paramètres

Il est possible d'affecter une valeur par défaut à chaque paramètre, comme on initialise une variable.

Suffixer le type d'un paramètre avec **...** permet d'accepter un nombre quelconque de paramètres de ce type (séparés par des virgules lors de l'appel), qui seront regroupés dans un tableau portant le nom du paramètre dans la fonction appelée. Cela ne peut être fait que pour un seul paramètre par fonction.

Les noms des paramètres doivent être passés lors de l'appel. Un nom de paramètre peut être précédé d'un label, qui sera alors utilisé à la place du nom lors de l'appel, mais uniquement là. L'utilisation d'un _ à la place du label permet d'omettre l'utilisation du label.

Les paramètres sont constants par défaut. Un paramètre peut être déclaré en entrée-sortie en utilisant le mot-clé **inout** devant le type, sa valeur sera alors obtenue en plaçant **&** devant la variable (ni constante ni littéral) associée. Un paramètre **inout** ne peut être variadique ni avoir de valeur par défaut.

Les variables peuvent cacher la visibilité des paramètres, ce qui permet de redéfinir un paramètre (constant) en tant que variable mais en lui attribuant le même nom.

Fermetures et fonctions incorporées

Des fonctions peuvent être définies à l'intérieur d'autres fonctions. Elles ne sont pas explicitement visibles de l'extérieur du bloc englobant bien qu'elles puissent être retournées par la fonction englobante.

Les fermetures sont des fonctions sans nom. Elles sont contenues entre accolades, et le bloc principal est introduit par le mot-clé **in**. Les paramètres ne peuvent avoir de valeur par défaut. Mais quand les paramètres et le type de retour peuvent être inférés, il est possible de ne mettre que les noms des paramètres avant le mot-clé **in**. Une fermeture composée d'une seule expression peut aussi omettre le mot-clé **return** car cette seule et unique expression sera alors retournée par défaut. Il est enfin possible d'omettre les noms des paramètres, qui seront accessibles avec leur numéro précédé de **$**.

Un opérateur peut même selon les conditions être utilisé en tant que fermeture, réduisant l'expression au minimum.

Une fermeture utilisée en tant que dernier argument d'un appel de fonction peut être écrite en dehors des parenthèses d'appel de la fonction et en omettant le label de ce dernier paramètre (si la fermeture est le seul paramètre de la fonction, les parenthèses de la fonction peuvent aussi être omises). Elle fait pourtant partie intégrante des paramètres de la fonction appelée. Le nommage des paramètres de la fermeture à l'aide du symbole **$** suivi d'une valeur numérique montre alors tout son intérêt.

Une fermeture peut capturer des variables et éventuellement en modifier le contenu, même si le contexte de définition de ces variables n'existe plus. Si une fermeture est *échappante* (en plaçant **@escaping** devant le type de la fermeture), elle ne sera appelée qu'une fois la fonction la prenant en paramètre terminée, elle doit donc faire référence éventuelle à **self** explicitement si le contexte d'un objet doit être précisé.

Une *fermeture automatique* est constituée d'une simple expression sans paramètre, qui peut capturer des variables externes et retourne sa valeur. C'est une formulation pratique pour passer du code en paramètre à une fonction ou à une autre fermeture, qui ne nécessite d'ajouter que les accolades autour du code. Si le type du paramètre de la fonction est marqué avec **@autoclosure**, les accolades ne sont même plus nécessaires lors du passage de paramètre.

Chapitre 6

Fonctions courantes

La fonction **print** permet d'afficher une chaîne de caractères.

La fonction **assert** permet de tester une condition, puis d'afficher un éventuel message si celle-ci est fausse (et éventuellement de passer aussi le nom du fichier et la ligne de code en question). Elle n'est active que dans la version de développement du code.

La fonction **precondition** agit de même, mais est valide en version de développement et de production.

La fonction **assertionFailure** part du principe que l'assertion a déjà été vérifiée comme étant fausse, et permet d'utiliser le mécanisme des assertion pour afficher l'erreur.

La fonction **fatalError** fonctionne de la même manière, et stoppe l'exécution du programme quelle que soit la configuration du compilateur (ignorant les assertions et préconditions ou pas).

La fonction **swap** permet d'échanger le contenu de deux valeurs de même type, quel que soit le type (grâce à la généricité de type).

Un nom de variable peut masquer le nom d'une fonction, et le compilateur pourra indiquer une erreur sibylline dans ce cas.

Chapitre 7

Structures de contrôle de flux

Boucle *pour*

La boucle **for-in** permet d'itérer sur tous les éléments d'une structure de données itérable, comme par exemple les tableaux, ensembles, dictionnaires, ou bien les intervalles (**...** et **..<**). Si l'indice qui permet d'itérer sur les valeurs n'est pas utile, il peut être nommé _.

Les fonctions **stride(from : to : by :)** (borne haute ouverte) et **stride(from : through : by :)** (borne fermée) permettent d'itérer par une autre valeur que 1.

Boucle *tant que*

Les boucles **while** et **repeat-while** répètent un test et des commandes jusqu'à ce que le test devienne faux, ce qui implique la sortie immédiate de la boucle. **while** exécute le test avant les commandes, **repeat-while** après les commandes.

Court-circuit des structures de boucle

continue permet de passer à l'itération suivante de la boucle englobante.

break interrompt immédiatement et sort de la boucle englobante ou du **switch** englobant (il est obligatoire en cas de **case** pour lequel il n'y aurait aucune action à exécuter).

Un label peut être créé en mettant deux points après le nom du label, qui précède une instruction, et les instructions **break** et **continue** peuvent faire référence à ce label, permettant ains de traverser plus d'une structure de contrôle de flux.

Structures d'alternative

La structure de contrôle classique **if-else** (le **else** étant facultatif) permet de proposer une structure d'alternative simple dans le code.

La structure de contrôle **guard**, dont le but est des tester la non nullité, est toujours accompagnée d'une clause **else** mais jamais de bloc *then*. Elle ne restreint pas la portée des variables déclarées dans la clause **guard**, qui sont accessibles après la structure de contrôle. Mais le code du bloc **else** doit contenir un mot-clé **return**, **continue**, **break** ou **throw**, pour sortir de l'itération de boucle de la fonction englobante, afin que la variable de la clause **guard** ne puisse être accédée par la suite du code.

La structure de contrôle **switch-case** permet de comparer une valeur à plusieurs autres. Il n'y a pas de **break** à ajouter à la fin de chaque **case** comme en *C*, mais **fallthrough** force le passage d'un case à la clause suivant directement, sans tester l'éventuelle clause correspondante.

La clause **default** est présente, elle doit apparaître en dernier et doit impérativement apparaître si toutes les valeurs possibles ne sont pas gérées par les **case** individuels.

Plusieurs valeurs peuvent être regroupées dans un même **case**, en les séparant par des virgules, ou bien en utilisant des intervalles ouverts ou fermés. De plus, une clause **case** ne peut pas être vide (sans code exécutable).

La correspondance dans un **case** peut être faite avec un wildcard (_), ou bien en utilisant à la place une constante, ce qui permet d'en réutiliser la valeur dans le traitement du case. Une clause **where** peut être ajoutée après une clause **case** pour ajouter des conditions supplémentaires sur les variables mises en correspondance.

Spécificités liées aux itérations

Pour itérer sur un tableau, on peut parcourir le tableau avec un élément dans une boucle **for**, ou bien utiliser la méthode **enumerated()** qui renvoie des t-uplets contenant index et valeur plutôt que la valeur seule.

Pour itérer sur un ensemble, on peut parcourir l'ensemble avec un élément dans une boucle **for**, ou bien utiliser la méthode **sorted()** qui renvoie un tableau des éléments triés par $<$.

Pour itérer sur un dictionnaire, on peut parcourir le dictionnaire avec un t-uplet (key, value) dans une boucle **for**, ou bien itérer indépendamment sur toutes les clés ou toutes les valeurs avec respectivement les propriétés **.keys** et **.values**. Les clés et valeurs ne sont pas ordonnées, mais la méthode **sorted()** peut être utilisée sur les tableaux associés aux propriétés **.keys** et **.values**.

L'affectation d'une variable est possible dans la partie test d'une structure **if** ou **while**. Dans le cas de l'affectation d'un *optionnel* (*optional binding*), elle renvoie vrai si la conversion a pu se faire, faux sinon.

Chapitre 8

Classes et structures

Il y a beaucoup de points communs entre les classes et les structures, mais les structures sont passées par valeur (alors que les classes sont les seules à être passées par adresse) et ne supportent pas l'héritage ni le comptage de références.

class définit une classe, **struct** une structure. La définition est faite entre accolades, et l'on peut affecter des valeurs par défaut aux champs. Les champs sont accédés à l'aide de l'opérateur **.**.

Initialisation

Les structures possèdent par défaut un initialiseur des membres, où il suffit entre parenthèses après le nom du type de la structure de nommer les champs puis de leur donner une valeur après **:**.

Les classes possèdent l'initialiseur nommé **init**, dont les paramètres servent au code de l'initialiseur pour fixer l'instance dans un état particulier. L'initialiseur ne retourne pas de valeur. Un **init** sans paramètre est fourni par défaut uniquement si aucun **init** n'a été défini.

Il est possible de définir un désinitialiseur qui sera appelé juste avant que l'instance de la classe ne soit libérée de la mémoire.

Lorsque l'on affecte une valeur initiale à une propriété stockée, soit directement soit par un initialiseur, les observateurs ne sont pas appelés.

Un initialiseur *de convenance* appelle un initialiseur de la même classe, et de manière ultime un des initialiseurs désignés de la classe. Il est introduit par le mot-clé **convenience**.

Un initialiseur d'une classe fille ne peut pas fixer la valeur d'une propriété stockée constante de sa classe mère, alors que l'initialiseur de la classe mère le peut.

Un initialiseur peut échouer, dans ce cas il faut le déclarer avec un point d'interrogation après le nom **init**. Il peut alors retourner explicitement la valeur **nil**, mais le retour de valeur n'est accepté que dans ce cas là ; ceci brise le précepte de la programmation structurée qui veut qu'il y ait un seul point de sortie à chaque bloc.

Désinitialisation

Le désinitialiseur d'une instance de classe porte le nom **deinit**, sans type de retour ni paramètre (donc pas de parenthèses). Il est unique par classe. Il est appelé implicitement (mais jamais explicitement), lorsque le système de comptage de références détecte que l'instance n'est plus nécessaire.

Propriétés

Les propriétés stockées peuvent être des variables ou des constantes. Elles peuvent être affectées d'une valeur par défaut lors de leur définition, ou affectées lors de la construction. On peut initialiser de manière

paresseuse une variable, en utilisant le mot-clé **lazy** devant **var** ou **let** : le calcul de la valeur est alors repoussé lors de la première utilisation. Attention : le mot-clé **lazy** n'est pas thread-safe.

Les propriétés calculées sont définies uniquement comme des **var** (jamais avec **let**, car une pr) avec un type, puis un bloc contenant deux autres blocs : le bloc **get** et le bloc **set**. Le premier bloc renvoie le type, le second utilise le type en tant que paramètre (si le paramètre n'est pas nommé, il porte le nom **newValue** par défaut). Une propriété peut être *uniquement calculée*, pour laquelle le bloc **set** n'existe pas ; il est alors possible de supprimer aussi le mot-clé **get** et ses accolades, pour ne conserver que le code de **get** dans les accolades englobantes.

Les propriétés calculées tout de même sont accédées avec un syntaxe de variable, pas un appel de fonction.

Les variables (et constantes) globales sont des propriétés paresseuses stockées, les variables (et constantes) locales sont des propriétés stockées. Variables globales et locales peuvent aussi être *calculées*, et utilisent la même syntaxe que pour les propriétés des objets.

Observateurs de propriétés

Il est possible d'ajouter des observateurs de modification de propriété à toutes les propriétés stockées, sauf les propriétés paresseuses. Il est par contre possible d'ajouter des observateurs aux propriétés héritées, qu'elles soient stockées ou calculées. Ces observations se font au travers de l'utilisation des blocs de code **willSet** (en définissant un paramètre contenant la prochaine valeur de la propriété, ou en utilisant le nom **newValue** par défaut) et **didSet** (en définissant un paramètre contenant l'ancienne valeur de la propriété, ou en utilisant le nom **oldValue** par défaut).

Les observateurs d'une classe ne sont pas appelés tant que l'initialiseur de la classe n'a pas été appelé. Les sous-classes peuvent donc déclencher les observateurs de la super-classe.

Propriétés de type

Il est possible de déclarer des propriétés liées au type et pas à une instance. Les propriétés peuvent être stockées ou calculées. Les propriétés stockées doivent alors avoir une valeur par défaut car il n'y a pas d'initialiseur de classe/type. Elles sont cependant *paresseuses* sans avoir besoin d'être marquées comme telles, et assurées d'être initialisées une seule fois, même lorsqu'accédées par plusieurs threads.

Les propriétés de type sont déclarées à l'aide du mot-clé **static**, mais les propriétés calculées peuvent utiliser à la place le mot-clé **class** pour autoriser la redéfinition de la propriété dans les sous-classes.

Méthodes

Les classes, mais aussi les structures ainsi que les énumérations peuvent accepter la définition de méthodes, qu'elles soient d'instance ou de type. Les méthodes d'instance sont déclarées comme le sont les fonctions, mais à l'intérieur de la structure de type englobante.

Le mot-clé **self** est une propriété implicite qui désigne l'instance en cours et dont l'utilisation permet à une méthode de faire explicitement référence à l'objet actuel.

Par défaut, les méthodes des types passés par valeur (structures et énumérations) ne peuvent pas modifier les propriétés d'instance. Ce comportement peut être modifié en plaçant le mot-clé **mutating** devant le mot-clé **func**, afin d'autoriser la modification des propriétés et même de la propriété implicite **self**. Les méthodes **mutating** ne peuvent cependant pas être appelées sur les types par valeur constants.

Toutes les modifications apportées à l'aide d'une méthode **mutating** sont appliquées à la structure originelle une fois la méthode terminée.

L'attribut **@discardableResult** permet d'indiquer qu'il est possible d'ignorer la valeur de retour d'une fonction/méthode.

Méthodes de type

Les méthodes de type sont déclarées à l'aide du mot-clé **static**, mais peuvent utiliser à la place le mot-clé **class** pour autoriser la redéfinition de la méthode dans les sous-classes. Dans les méthodes de type, le mot-clé **self** fait référence à l'objet type lui-même, et pas à une instance de ce type.

Chaînage des propriétés, méthodes et indiçages

Les accès aux propriétés, méthodes et indiçages peuvent s'enchaîner même si les valeurs de retour peuvent renvoyer **nil**. L'opérateur utilisé est **?.**, il interrompt la chaîne si un objet cible est **nil** mais ne génère par d'erreur d'exécution. On peut cependant à la fin tester si l'on récupère **nil** ou pas, car le résultat est toujours un type optionnel (même si les types de retour des méthodes chaînées ne le sont pas).

L'opérateur d'indexation peut être utilisé à la place d'une propriété ou d'un appel de méthode. On remplace alors le **.** à droite du **?** par les crochets.

Lors de l'affectation à un chaînage, l'opérande droit de l'opérateur d'affectation n'est pas évalué si le chaînage échoue.

Lors d'un appel de fonction sans valeur de retour, on peut toujours tester si l'appel a été un succès car alors le type de retour passe de **Void** à **Void ?**. Il en est de même lorsque l'on affecte une valeur à une propriété : le type étant aussi **Void**, il passe de même à **Void ?**.

Redéfinition d'opérateur

Les classes, les structures et les énumérations peuvent définir leurs opérateurs d'indexation (plusieurs car ils peuvent être surchargés), à l'aide du mot-clé **subscript** à la place du nom de la méthode. Un ou plusieurs paramètres doivent alors être fournis, ainsi qu'un type de retour, afin de permettre des indexations multi-dimensionnelles.

Ces opérateurs peuvent être en lecture seule ou en lecture-écriture, comme les propriétés calculées, et sont définis comme ces dernières avec **get** et **set**.

Les opérateurs d'indexation ne peuvent prendre de paramètres en entrée-sortie, ni fournir de valeur par défaut à leurs paramètres. Il s'utilisent avec la notation tableau (entre [et]), avec un adressage multi-dimensionnel qui se fait à l'aide de virgules (et non pas de paires de crochets successives, comme c'est l'habitude en *C*).

Chapitre 9

Héritage

Seules les classes peuvent hériter en *Swift*.

Une classe définie comme **final class** ne peut cependant pas être héritée.

Swift utilise l'héritage simple d'implémentation. La super-classe est nommée après le nom de la classe qui en hérite, et après **:**.

Il n'existe pas de classe de base universelle comme en *Objective-C*. Toute classe qui n'hérite d'aucune autre classe peut donc devenir une classe de base.

Les opérateurs **as ?** et **as !** peuvent imposer une conversion de type vers les feuilles de la hiérarchie, renvoyant un optionnel ou forçant la conversion, selon la situation et le besoin.

Initialiseur

Les propriétés stockées sont d'abord initialisées, puis l'initialiseur désigné de la classe mère est appelé, puis l'initialiseur désigné de la classe fille est exécuté, puis l'initialiseur de convenance est exécuté. Le chaînage des initialiseurs doit être fait explicitement (par **super.init()**), sinon il pourra se dérouler dans l'ordre classe fille puis classe mère.

La première phase d'une initialisation est l'initialisation des propriétés stockées (que ce soit par une valeur par défaut ou par une initialisation dans le corps de l'initialiseur) en remontant des classes filles vers les classes mères. La seconde phase est le parcours inverse des classes mères vers les classes filles, avec possibilité de modification des propriétés et d'appel des méthodes d'instance.

Les initialiseurs ne sont normalement pas hérités par défaut. Mais si une classe ne fournit pas d'initialiseur désigné, elle hérite de tous ceux de sa classe mère. De plus, si une classe procure une implémentation de tous les initialiseurs désignés de sa classe mère (explicitement ou par la régle précédente), elle hérite de tous les initialiseurs de convenance de sa classe mère.

Lorsqu'un initialiseur échoue (renvoie **nil**), la chaîne d'initialisation est arrêtée immédiatement, et plus aucun code d'initialisation n'est exécuté.

Il n'est pas possible de redéfinir un initialiseur qui n'échoue pas à l'aide d'un initialiseur qui peut échouer (l'inverse est possible).

Il est aussi possible de désempaqueter explicitement un initialiseur, en lui ajoutant un **!** après **init**. Une assertion poura être générée si cela échoue.

Le mot-clé **required** devant un initialiseur indique que l'initialiseur doit être présent dans les classes filles. Il peut cependant être hérité, mais doit être redéfini sans le mot-clé **override** mais avec le mot-clé **required**.

Une fermeture peut être utilisée pour définir la valeur initiale d'une propriété stockée. Cela se fait à l'aide de l'opérateur =, du code entre accolades, et du couple de parenthèses ouvrante-fermante afin d'appeler explicitement le code de la fermeture pour réaliser l'initialisation. Mais vu que rien n'est garanti d'être initialisé, la fermeture ne peut faire appel aux propriétés de l'instance, ou même à **self**.

Désinitialiseur

Les désinitialiseurs sont hérités et appelés automatiquement.

Redéfinition

Le mot-clé **override** est obligatoire en cas de redéfinition de méthode d'une classe mère dans une classe fille.

Le mot-clé **super** permet d'accéder à la super-classe (appel de méthode, utilisation de propriété ou d'indexation).

Il est possible de fournir un *getter* ou un *setter* pour toute propriété héritée, qu'elle soit stockée ou calculée, à l'aide respectivement des mot-clés **get** et **set**. Mais fournir un *setter* implique de fournir un *getter*.

Il n'est pas possible de redéfinir des observateurs sur des constantes ou des propriétés calculées en lecture seule. Il n'est pas non plus possible de redéfinir à la fois un *setter* et un observateur sur la même propriété.

Le mot-clé **final** permet d'interdire une future redéfinition dans une classe fille.

Il est possible de redéfinir des initialiseurs, mais il ne faut pas utiliser le mot-clé **override** avec la redéfinition d'un initialiseur de convenance, car celui-ci doit tout de même appeler un initialiseur désigné de sa propre classe, et n'est donc pas en tant que tel une vraie redéfinition de l'initialiseur de convenance de la classe mère.

Protocole

Un protocole est défini à l'aide du mot-clé **protocol**. Il est l'équivalent d'un type, au même titre qu'une classe par exemple. Les protocoles peuvent hériter d'autres protocoles, en les listant séparés par des virgules après deux points. En faisant hériter un protocole du protocole **AnyObject**, seules les classes pourront adopter le protocole (pas les structures ou les énumérations) : ceci peut être utile pour s'assurer que la donnée utilise une sémantique à référence plutôt qu'une sémantique à valeur.

On hérite d'un protocole en listant ce protocole après le symbole deux points placé après le nom de l'entité qui hérite du protocole. Plusieurs protocoles peuvent être listés, séparés par des virgules. Mais si l'entité hérite aussi d'autre chose qu'un protocole, il faut placer les protocoles en dernier, toujours derrière une virgule.

Si un protocole impose qu'une propriété soit accessible en lecture, l'implémentation peut aussi fournir un accesseur en écriture. L'inverse n'est pas vrai.

Les exigences de propriété sont introduites par le mot-clé **var**, celle de méthode par le mot-clé **func**.

Les exigences de propriété à *getter* ou *setter* sont remplies en indiquant **{ get set }** ou **{ get }** après la déclaration.

Il est de plus possible de déclarer des propriétés de type, avec le mot-clé **static**.

Les méthodes qui modifient l'instance de structure ou d'énumération à laquelle elles appartiennent doivent utiliser le mot-clé **mutating**.

L'implémentation d'un initialiseur imposé par un protocole doit se faire avec le mot-clé **required**. Ce mot-clé est compatible avec le mot-clé **override**, dans le cas où ce même initialiseur redéfinirait un initialiseur de la classe mère.

On peut créer une liste de protocoles, par exemple pour utiliser plusieurs protocoles en tant qu'un seul lors d'un passage de paramètres, en séparant ces protocoles avec un **&**. Il est même possible d'ajouter un nom de classe pour imposer l'utilisation d'une classe de base.

Par compatibilité avec *Objective-C*, il peut exister des exigences optionnelles dans la conformance au protocole, indiquées par le mot-clé **optional**. Ces exigences, ainsi que le protocole en questions, doivent être marquées par **@objc**.

Chapitre 10

Contrôle d'accès

Utilisation de modules

On importe un module externe grâce au mot-clé **import**. Ce module aura dû être créé à l'aide du compilateur avec l'option **-emit-module**.

Un fichier source est associé à un seul module. Plusieurs fichiers source peuvent appartenir à un même module.

Niveaux d'accès

Aucune entité d'un niveau d'accès donné ne peut être définie grâce à une ou plusieurs entités de niveau d'accès plus restrictif. Par défaut, le niveau d'accès si rien n'est spécifié est le niveau d'accès interne.

Pour pouvoir exécuter les tests unitaires, marquer l'importation d'un module, compilé cependant avec l'option de test activé, avec **@testable**.

Différents niveaux d'accès sont disponibles en *Swift* :

Accès ouvert

L'accès ouvert permet de donner un accès illimité aux entités définies dans le code *Swift*. Il ne peut être appliqué qu'à des classes et aux membres de ces classes.

Le mot-clé pour indiquer un accès ouvert est **open**.

Accès public

L'accès public permet de donner un accès illimité aux entités définies dans le code *Swift*.

Les classes à accès public ne peuvent être héritées en dehors de leur module de définition. Les membres à accès public ne peuvent être redéfinis que dans leur module de définition. Les membres d'une entité en accès public sont en accès interne.

Le mot-clé pour indiquer un accès public est **public**.

Accès interne

L'accès interne n'ouvre l'accès qu'aux fichiers source du même module. Les membres d'une entité en accès interne sont aussi en accès interne.

Le mot-clé pour indiquer un accès interne est **internal**.

Accès *privé de fichier*

L'accès *privé de fichier* ne permet l'accès aux entités qu'au code du même fichier source. Les membres d'une entité en accès *privé de fichier* sont aussi en accès *privé de fichier*.

Le mot-clé pour indiquer un accès *privé de fichier* est **fileprivate**.

Accès privé

L'accès privé ne permet l'accès qu'à la déclaration englobante, ou aux extensions de l'entité définies dans le même fichier source. Les membres d'une entité en accès privé sont aussi en accès privé.

Le mot-clé pour indiquer un accès privé est **private**.

Niveau d'accès des t-uplets

Il n'est pas possible de spécifier explicitement le niveau d'accès d'un t-uplet, celui-ci sera déduit comme égal au niveau d'accès le plus restreint de ses composants.

Niveau d'accès d'une fonction

Il est possible de spécifier explicitement le niveau d'accès d'une fonction. Cependant, celui-ci pourra être déduit comme égal au niveau d'accès le plus restreint de ses paramètres ou de son type de retour. La spécification explicite est obligatoire lorsque le niveau d'accès déduit ne correspond pas au niveau par défaut dans le contexte actuel.

Niveau d'accès d'une énumération

Les valeurs d'une énumération ont le même niveau d'accès que le type énumération à laquelle elles appartiennent.

Les types des valeurs brutes ou associées ne peuvent pas avoir de niveau d'accès moins restreint que celui de l'énumération à laquelles ils appartiennent.

Niveau d'accès des types incorporés

Les types incorporés dans un type de niveau d'accès privé ou *privé de fichier* sont de même niveau.

Les types incorporés dans un type de niveau d'accès interne ou public ont un niveau d'accès interne.

Niveau d'accès de types dérivés

Les types dérivés ne peuvent pas être plus permissifs que la classe de base. Mais les membres peuvent être promus à un niveau plus haut que celui du membre redéfini.

Spécificités des variables, constantes, propriétés et opérateurs d'indexation

Les variables, constantes et propriétés ne peuvent être plus publics que leur type.

Un opérateur d'indexation ne peut être plus public que son indice ou son type de retour.

Les *getters* et *setters* reçoivent le même niveau de visibilité que celui de la variable, constante, propriété ou indexation à laquelle ils appartiennent.

Un *setter* peut avoir un niveau d'accès plus restreint que son *getter* correspondant, afin de pouvoir mieux contrôler les accès en lecture ou en écriture. On utilise alors les écritures **internal(set)**, **fileprivate(set)** ou **private(set)** devant la définition de **var** ou **subscript** correspondante. Il est possible d'utiliser ces écritures pour des *propriétés stockées* pour lesquelles *Swift* synthétise automatiquement les accesseurs, afin de conserver la souplesse d'accès en lecture-écriture.

Spécificités des *initialiseurs*

Un initialiseur *requis* doit avoir le même niveau d'accès que la classe elle-même.

Les autres initialiseurs peuvent voir leur niveau d'accès restreint par rapport au niveau de la classe associée.

Les initialiseurs sont soumis aux mêmes restrictions que les fonctions vis-à-vis du niveau d'accès de leurs paramètres et de leur type de retour.

L'initialiseur par défaut fourni par *Swift* possède le même niveau d'accès que celui de la classe elle-même, sauf si cette classe est publique, auquel cas il est interne ; ceci imposera de produire soi-même l'initialiseur non paramétré si l'on veut que celui-ci soit accessible de manière publique.

L'initialiseur membre à membre d'une structure est respectivement considéré comme privé ou *privé de fichier* dès qu'une propriété stockée de la structure est privée ou *privée de fichier*. Hormis cela, l'initialiseur est interne. Suivant le besoin, il sera adéquat de fournir soi-même un initialiseur qui sera public.

Spécificités des *protocoles*

Il est possible de définir le niveau d'accès d'un protocole. Toutes les exigences d'un protocole sont du même niveau d'accès que le protocole lui-même, afin d'assurer que toute entité adoptant le protocole ait accès à ces exigences (pour un protocole public, les exigences sont elles aussi publiques et non pas internes).

Un protocole ne peut hériter que d'un autre protocole qui a un niveau d'accès aussi permissif que le sien, pas moins permissif.

Par contre, un type peut adopter un protocole moins permissif que son niveau d'accès, si tant est que l'utilisation des exigences de ce protocole est faite dans le bon niveau d'accès. Le respect d'un protocole peut donc être fait à un niveau autre que le niveau d'accès du type lui-même, ce n'est pas en contradiction avec le respect des niveaux d'accès.

Spécificités des *extensions*

Une classe, structure ou énumération peut être étendue dans n'importe quel contexte dans laquelle elle est disponible.

Les entités ajoutées par l'*extension* ont par défaut le même niveau d'accès que le niveau du type étendu lui-même.

Un niveau d'accès peut cependant être spécifié pour une *extension*, de même que pour chacune des entités ajoutées.

Si l'extension est utilisée pour ajouter l'adoption d'un protocole, c'est le niveau d'accès du protocole qui est utilisé, pas celui de l'*extension*. Il n'est pas possible de modifier cela.

De par leur nature, les extensions ont accès aux membre privés du type étendu, ainsi qu'aux membres privés des autres extensions du type. Leurs membre privés peuvent aussi être accédés par le type étendu.

Spécificités des types génériques

Le niveau d'accès d'un type ou d'une fonction générique est le niveau le plus restrictif entre le niveau du type lui-même et le niveau des types paramètres.

Spécificité des alias de type

Un alias aura toujours un niveau aussi ou plus restreint que le type aliasé.

Chapitre 11

Extensions

Les extensions permettent d'ajouter de nouvelles fonctionnalités à des classes, des structures, des énumérations ou des protocoles existants, sans en avoir le code source. Elles ne peuvent qu'ajouter du code ou des définitions de type, jamais ajouter de donnée à proprement parler ou de modification à une fonctionnalité existante.

De nouveaux types (dits *incorporés*) peuvent aussi être ajoutés par la définition de l'extension.

Les extensions sont déclarées avec le mot-clé **extension**, suivi du type modifié (suivi éventuellement des protocoles ajoutés).

L'impact de l'utilisation d'une extension est global sur le programme, même pour les instances créées avant l'extension du type.

De nouveaux initialiseurs de convenance sont autorisés, mais pas d'initialiseur désigné ou de désinitialiseur.

Les méthodes qui modifient l'instance d'une structure ou d'une énumération doivent marquer la méthode comme **mutating**.

Les extensions peuvent adopter des protocoles, qui font que la classe étendue adopte automatiquement par effet de bord les protocoles utilisés. Il est possible de créer une extension vide, qui adopte un protocole, pour faire en sorte qu'une classe qui n'adopte pas un protocole mais qui s'y conforme soit finalement considérée comme adoptant ce protocole.

Les extensions peuvent aussi étendre les protocoles, leur permettant ainsi de fournir des implémentations par défaut pour telle ou telle exigence. Si une classe étendue propose une implémentation des exigences du protocole, c'est cette implémentation qui sera retenue, pas celle fournie par l'extension (pour rester conforme au fait de ne pas modifier une fonctionnalité existante).

Les extensions de protocole peuvent être conditionnées par le mot-clé **where**.

Chapitre 12

Surcharge d'opérateurs

Les opérateurs par défaut de *Swift* ne gère pas les dépassements de dynamique, qui sont considérés comme des erreurs à l'exécution.

De nouveaux opérateurs arithmétiques, de nom similaire aux précédents mais préfixés par **&**, ont été ajoutés au langage.

Tous les opérateurs peuvent être surchargés en *Swift*, à l'exception de l'opérateur d'affectation (=) et de l'opérateur ternaire (**? :**), et leurs priorité et associativé peuvent être personnalisées. Ces opérateurs peuvent se trouver dans des *extensions* et ainsi être ajoutés à des types déjà existants.

Les opérateurs peuvent être surchargés comme les méthodes d'un objet sont définies, en remplaçant le nom de la fonction définie par le symbole de l'opérateur lui-même. Ces surcharges peuvent aussi être des méthodes de classe plutôt que d'instance. Les paramètres de la fonction sont les opérandes de l'opérateur.

Dans le cas de la surcharge d'un opérateur composé, le premier paramètre doit être qualifié par **inout** car il joue à la fois le rôle de valeur d'entrée et de valeur de sortie.

En utilisant les mots-clés **prefix**, **infix** et **postfix** devant le mot-clé **func**, il est possible de définir des versions respectivement préfixées et postfixées d'un opérateur.

Opérateurs personnalisés

Les opérateurs personnalisés viennent s'ajouter aux opérateurs standards de *Swift*.

Les caractères autorisés pour définir des opérateurs personnalisés sont définis dans une liste particulière. Un opérateur ne peut par contre pas consister en un unique **?**, et les opérateurs postfixés ne peuvent démarrer ni par un **?** ni un **!**.

Les opérateurs peuvent démarrer par un **.**. Ils peuvent alors posséder un point ailleurs dans la suite de leurs symboles. Si un opérateur ne démarre pas par un point, ils ne peuvent pas contenir de point par ailleurs.

Précédence des opérateurs infixés

Il est possible d'attribuer une précédence à une déclaration d'opérateur avec la même syntaxe que la notion d'héritage : après **:**, on place le qualificatif.

Chapitre 13

Types génériques

Utilisation des types génériques

Un type générique peut être défini entre < et > après la définition d'une fonction, u accolé à la définition d'une structure, énumration ou classe.

Dans le cas de la fonction, celle-ci sera appelée et le type générique inféré à partir des paramètres passés à la fonction.

Dans le cas d'un nouveau type générique, celui-ci devra être paramétré lors de sa définition, pour remplacer le type abstrait entre < et > par un type concret.

L'extension d'un type générique se fait sans rappeler la syntaxe du type générique ajouté, seulement en reprenant le nom de base.

Lors de l'utilisation d'un type générique dans une déclaration, il est possible d'imposer des contraintes en listant après les **:** un ou d'autres types dont le type en question doit hériter.

Les types génériques peuvent aussi être appliqués sur des opérateurs d'indexation. L'opérateur seul peut ainsi être spécialisé, pour spécifier un type d'indice particulier par exemple.

Types associés

Un type associé est un type lié à la définition d'un protocole. C'est un identifiant, défini dans la définition du protocole, et qui est introduit à l'aide du mot-clé **associatedtype**. Dans l'objet adoptant le protocole, le type associé pourra être défini par lui-même ou par l'utilisation de **typealias**.

Contrainte par *where*

Des contraintes supplémentaires sur les types génériques peuvent être imposées à l'aide du mot-clé **where**, qui sera suivi de comparaisons de types ou de relations d'héritage entre types. Plusieurs exigences de type peuvent être utilisées, en les séparant par des virgules.

La clause **where** peut aussi être appliquée sur un type associé.

Chapitre 14

Gestion des erreurs

Les valeurs d'erreur doivent se conformer au protocole **Error**.

Déclencher une erreur de ce type se fait par le mot-clé **throw**.

Une fonction qui peut renvoyer une erreur doit être déclarée avec en mot-clé suffixe **throws** avant l'éventuel type de retour. Une telle fonction doit être appelée avec le mot-clé préfixe **try**. Une structure de contrôle **do-catch** peut englober le **try**, ou bien la fonction englobante peut elle-même lancer une erreur.

try ? permet de renvoyer **nil** si une exception est lancée lors de l'évaluation de la valeur de droite.

try ! permet au contraire d'assurer le compilateur qu'une exception ne sera pas lancée lors de l'évaluation de la valeur de droite. Des optimisations peuvent alors être mises en œuvre par le compilateur.

Une clause **catch** peut référencer une valeur d'erreur qui contient des paramètres, et ces paramètres peuvent être nommés et donc utilisés dans le bloc dépendant de la clause. Pour une clause **catch** sans paramètre, toutes les erreurs sont possibles, et l'erreur est alors référencée par une constante nommée **error**.

Une clause **where** peut être ajoutée après une clause **catch** pour ajouter des conditions supplémentaires sur l'erreur traitée.

Chapitre 15

Comptage de références automatique (*ARC*)

Le comptage de références utilisé par *Swift* est un comptage de références classique. Il ne s'applique donc qu'aux classes, car ce sont les seuls types qui sont manipulés par référence et pas par valeur.

Pour éviter les cycles de références, qui empêcheront la libération automatique des objets, il est possible d'utiliser des références faibles ou des références *sans possesseur (**unowned**)*.

Références faibles

Une référence faible est une référence qui n'incrémente ni ne décrémente le compteur de références d'un objet. Elle est toujours de type optionnel et s'introduit avec le mot-clé **weak** devant l'optionel qu'elle qualifie. Cet optionel doit de plus être une variable, car il doit pouvoir être promu à **nil** lorsque l'objet est désalloué (attention, l'observateur de propriété n'est pas appelé lorsque l'objet est désalloué et le pointeur faible mis à **nil**).

Références sans possesseur (*unowned*)

Une référence sans possesseur est similaire à une référence faible, mais s'introduit avec le mot-clé **unowned**. Elle est cependant supposée tout le temps avoir une valeur, et ne peut donc être forcée à **nil** comme le peut être une référence faible. Elles ne sont donc pas des optionnels, et il ne faut les utiliser que si l'on est sûr que l'objet existe et n'a donc pas été désalloué.

Une référence **unowned** peut aussi être qualifiée de *non-sûre* par le mot-clé **unowned(unsafe)**, auquel cas le compilateur n'émettra pas de code de vérification d'accès à un objet qui aurait pu être désalloué. Ce cas ne devrait jamais arriver, mais *Swift* s'en prémunit, ce qui peut poser un problème de performance. Le qualificatif **unsafe** permet de s'affranchir de ce test qui peut être inutile.

Fermeture créant une référence forte

Lorsqu'une fermeture utilise un objet, elle crée une référence forte vers cet objet. Si la fermeture est affectée à une variable membre de l'instance, il y a création d'un cycle de références fortes.

Pour montrer explicitement ce lien, *Swift* impose d'utiliser **self.** devant toute utilisation par une fermeture d'une variable ou fonction membre d'un objet dont la fermeture est membre.

Swift propose pour régler ce problème de définir une *liste de capture*, en listant entre crochets les références vers une instance de classe ou une variable initialisée avec une valeur, chacune précédée de **weak** ou **unowned**.