Le Langage Swift

Apprendre la Programmation avec Méthode, Clarté et Concision

Publié par les Éditions CristalSwift

Première édition

Historique des mises à jour
14/02/2022 Première version

ISBN : 979-10-359-7179-3

Dépôt légal : Février 2022

Les Éditions CristalSwift® 53B Avenue du Général de Gaulle - 90380 ROPPE (France)

Licence de livre

En achetant ce livre, vous disposez de la licence suivante :

- Vous êtes autorisé à utiliser et/ou modifier le code source dans autant d'applications que vous le souhaitez, sans aucune restriction.
- Vous êtes autorisé à utiliser et/ou à modifier toutes les illustrations, images et conceptions dans autant d'applications que vous le souhaitez, mais vous devez inclure cette ligne quelque part dans votre application, en guise de citation de la source : [*Illustration/Image/Conception : adaptée du livre Le Langage Swift, disponible sur www.cristalswift.com*].
- Le code source inclus dans le livre est réservé à votre usage personnel. Vous n'êtes PAS autorisé à le distribuer ou à le vendre sans autorisation préalable. Vous pouvez le télécharger depuis Github : https://github.com/Cristalswift/LeLangageSwift.
- Ce livre est réservé à votre usage personnel. Vous n'êtes PAS autorisé à le vendre sans autorisation préalable, ni à le distribuer à des amis, collègues ou étudiants. Ces derniers doivent acheter leurs propres exemplaires.
- Tous les documents fournis avec ce livre sont fournis "tels quels", sans garantie d'aucune sorte, expresse ou implicite, y compris, mais sans s'y limiter, les garanties de qualité marchande, d'adéquation à un usage particulier et de non-contrefaçon. En aucun cas, les auteurs ou les détenteurs des droits d'auteur ne pourront être tenus responsables de toute réclamation, dommage ou autre responsabilité, que ce soit dans le cadre d'une action contractuelle, délictuelle ou autre, découlant de, ou en lien avec le logiciel ou d'autres aspects dans le logiciel Xcode et ses déclinaisons.

Sommaire

Avant-propos

Vous avez fait le choix de vous initier au développement d'applications natives dans l'écosystème Apple et peut-être aussi pour le web et vous souhaitez opter pour un langage reconnu et largement utilisé dans le monde professionnel.

Le langage de programmation Swift se révèle être optimal car c'est un langage d'avenir. Son apprentissage est donc un investissement très sûr et rentable au plan professionnel. Par ailleurs, il constitue également un levier pour votre épanouissement personnel dans l'univers actuel de l'implémentation des technologies relatives à l'intelligence artificielle.

Pour autant, vous devez accepter et vous préparer cette réalité : la programmation informatique est une activité intellectuelle exigeante dont l'aboutissement économique nécessite de la créativité et une bonne dose d'imagination artistique sans sacrifier la culture scientifique.

Aussi, je vous encourage à développer voire consolider votre passion par la pratique et l'expérimentation... C'est précisément l'objectif poursuivi par les éditions Cristalswift en publiant le présent ouvrage que vous lisez.

Pour bien suivre les didacticiels contenus dans ce livre, vous aurez besoin de :

- Un Mac exécutant macOS Big Sur 11.6 ou version ultérieure. avec la dernière version intermédiaire et les correctifs de sécurité installés. Cela vous permet d'installer la dernière version de l'outil de développement requis : Xcode.

- Xcode 13.2.1 ou une version ultérieure. Xcode est le principal outil de développement pour écrire du code Swift. Vous avez besoin au minimum de Xcode 13, car cette version inclut le Playground associé à Swift 5.6. Vous pouvez télécharger gratuitement la dernière version de Xcode sur le Mac App Store.

Si vous n'avez pas installé la dernière version de Xcode, assurez-vous de le faire avant de continuer avec le livre. Le code couvert dans ce livre dépend de Swift 5.6 et Xcode 13 au minimum. Vous risquez de vous perdre si vous essayez de travailler avec une version plus ancienne ou de travailler en dehors de l'environnement de jeu que ce livre suppose.

Si vous êtes un débutant en programmation, ce livre est fait pour vous ! Il y a de courts résumés des points étudiés dans chaque chapitre tout au long du livre pour vous donner une pratique de programmation et tester vos connaissances en cours de route.

Tout le code source ainsi que les fichiers Unicode qui accompagnent ce livre sont en accès libre par téléchargement depuis Github à https://github.com/Cristalswift/LeLangageSwift.

Introduction

Swift est un langage de programmation crée par Apple. Ce langage a été mis à la disposition des développeurs du monde entier en 2014 lors de la conférence mondiale des développeurs d'applications pour l'écosystème Apple (WWDC : World Wide Developers Conference). L'engouement de la communauté mondiale pour ce langage, en raison de sa puissance et de sa fiabilité, a poussé Apple à le rendre open source en décembre 2015. Il devint donc possible de faire la programmation en Swift également sur les systèmes Linux et bientôt sous Windows et Android. L'ambition de porter Swift sur les systèmes Linux est de le positionner en tête de liste des langages de développement d'applications web. Ce qui est d'ores et déjà une réalité. Grâce aux contributions très organisées de la communauté mondiale, le langage Swift ne cesse d'être amélioré. C'est ainsi que la sortie de la version 5 du langage en 2019 a abouti à la stabilisation de l'interface binaire d'application (**ABI : Application Binary Interface**). Ce qui signifie que tout fragment de code ou toute librairie logicielle écrit en Swift à partir de la version 5 restera compatible avec les versions ultérieures du langage sans avoir besoin d'être compilé à nouveau.

Ce livre est consacré à l'apprentissage méthodique de Swift 5.6 qui est la dernière version en date au moment de la rédaction de cet ouvrage. Malgré sa jeunesse, Swift est devenu le langage de référence pour des applications de toutes sortes pour macOS, iPadOS, iOS, tvOS et watchOS. Par ailleurs, il est ne cesse de gagner en popularité quant au développement d'application web hébergée sur Linux ou dans le cloud. Ce que vous allez apprendre dans ce livre vous sera donc très utile pour le développement et la consolidation de vos compétences pour devenir un développeur Swift professionnel.

Dans un premier temps, la lecture appliquée du présent ouvrage vous permettra d'apprendre davantage sur les concepts de base comme les constantes, les valeurs, les opérations et les types de données intégrés au langage. Ensuite, l'attention sera portée sur les concepts intermédiaires comme les structures, les classes, et les énumérations. Enfin, vous finirez par acquérir des connaissances approfondies sur les extensions de protocole, les opérateurs personnalisés, la programmation orientée protocole, les génériques et la programmation parallèle et asynchrone. Swift vous permet de créer de belles abstractions pour résoudre des problèmes du monde réel que vous découvrirez dans ce livre en guise d'apprentissage.

Swift est aussi un langage très ludique d'un point de vue intellectuel pour des expérimentations. Il est facile d'essayer de petits extraits de code lorsque vous testez de nouvelles idées. La programmation est une expérience pratique, et Swift le rend rapide et facile à suivre avec ce livre ainsi qu'à explorer par vous-mêmes.

Partie I - BASES TECHNIQUES ET SYNTAXIQUES

Chapitre 1 : Concepts de base et le logiciel de programmation Xcode

Chapitre 2 : Types de données élémentaires et les opérateurs associés

Chapitre 3 : Comment prendre des décisions en faisant des choix ?

Chapitre 4 : Comment affiner les flux d'instructions en comptant sur les répétitions ?

Chapitre 5 : Comment tirer avantage des types optionnels de données ?

Chapitre 6 : Comprendre et exploiter les fonctions dans l'univers du langage Swift

Chapitre 7 : Comment composer des blocs d'algorithmes grâce aux clôtures ?

Chapitre 8 : Comment programmer avec les listes, les dictionnaires et les ensembles ?

Chapitre 1

Concepts de base et le logiciel de programmation Xcode

Lorsqu'un développeur d'applications conçoit un programme, il doit, entre autres, définir les données ainsi que les objets nécessaires à la réalisation du programme. Cette définition consiste à nommer ces objets et à décrire leur contenu afin qu'ils puissent être stockés en mémoire. C'est pourquoi nous étudions dans ce chapitre le fonctionnement du code à travers le mécanisme de stockage des données en mémoire de l'ordinateur. Les notions de variables, valeur et constante seront ensuite abordées ; puis l'aventure continuera dans les entrailles de Xcode, l'environnement de développement intégré pour écrire des programmes en langage Swift. Une attention particulière sera accordée à **Playground**, le concept-outils d'aires de jeux mis à disposition par Apple dans Xcode pour expérimenter du code.

1.1. Fonctionnement synoptique de l'ordinateur

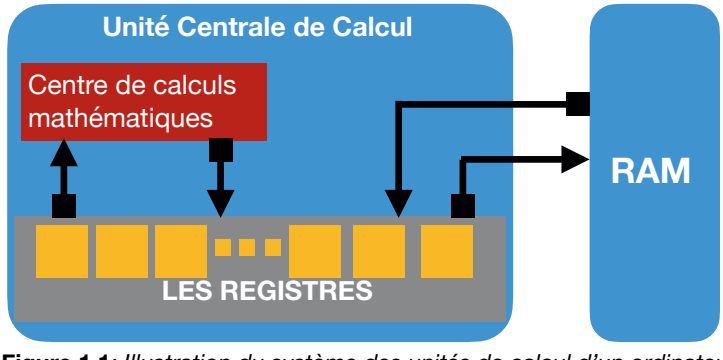

Figure 1.1*: Illustration du système des unités de calcul d'un ordinateur*

L'ordinateur (ordinateur de bureau, tablette, smartphone, ...) n'est en réalité d'un objet technologique sans aucune émotion ni une quelconque intelligence ; et pourtant il a tout l'air du contraire, aux yeux du commun des mortels. La vérité, c'est que sa puissance lui vient de la façon dont les développeurs, comme vous et moi, le programment. Si vous avez la ferme intention et la détermination sans faille de dompter la puissance des ordinateurs et j'espère que c'est le cas, il est important de comprendre comment ces derniers fonctionnent.

Au cœur de l'ordinateur se trouve une **Unité Centrale de Calcul (UCC)** ou **Central Processing Unit** (**CPU**) en anglais. Il s'agit essentiellement de l'endroit où s'opèrent tous les calculs mathématiques comme l'addition, la soustraction, et bien d'autres opérations arithmétiques.

Aussi, tout ce que vous accomplissez avec votre ordinateur est le résultat de millions de calculs par seconde. C'est juste impressionnant !

Le CPU stocke les nombres (les données en général), à utiliser dans les calculs, dans de petites unités de mémoire appelées registres. Ces nombres peuvent être récupérés par le CPU depuis la mémoire centrale de l'ordinateur appelée la **Mémoire vive** ou **Random Access Memory** (**RAM**) en anglais. Le CPU peut également écrire dans la mémoire vive, les nombres stockés dans les registres. Ceci permet au CPU de pouvoir travailler avec une très grande quantité de données qui n'auraient pas pu être toutes stockées dans les registres.

Chaque fois que le CPU fait une opération d'addition, de soustraction ou de lecture depuis la mémoire vive ou d'écriture dans cette dernière, il exécute ce qu'on appelle une ***simple instruction***. Chaque programme d'ordinateur accomplit la tâche qu'on lui confie en exécutant des milliers voire des millions de ces simples instructions. Le fonctionnement d'un programme très complexe comme les systèmes d'exploitation macOS, iOS, iPadOS, watchOS ou tvOS (en effet, ce sont aussi des programmes d'ordinateur) consiste en l'exécution de millions d'exécution.

Dans le cadre du développement d'applications, au lieu d'écrire des instructions machine très élémentaires, on écrit du code source dans un langage de programmation spécifique, en l'occurrence le langage Swift. Ce code est converti en instructions machine pour être exécuté par le CPU grâce à un autre programme qu'est le ***compilateur***. Ainsi, chaque ligne de code du langage Swift est compilée en plusieurs instructions machines élémentaires en arrière-plan.

1.2. Secrets de la représentation des données

En informatique, toutes les informations ou données sont en définitive des nombres. Aussi, quelle que soit l'information envoyée au compilateur, cette dernière sera convertie en nombre pour être exécutée. Par exemple, chaque caractère composant un bloc de texte est représenté par un nombre. Les images n'échappent pas à cette règle. Une image est subdivisée en des milliers voire des millions de petits éléments graphiques appelés pixels. Chaque pixel une couleur composite, laquelle est aussi une combinaison linéaire de trois nombres en dernier ressort. Chacun de ces trois nombres représente une quantité de couleur élémentaire à savoir le rouge, le vert et le bleu. Par exemple, un pixel entièrement rouge serait composé de 100% de rouge, 0% de vert et 0% de bleu.

Évidemment, le système de numération qui sert de référence au CPU pour les calculs à exécuter diffère du nôtre. Au quotidien, nous raisons et faisons nos calculs avec des nombres allant de 0 à 9. C'est le **système décimal** ou le **système de numération en base 10** (c'est-à-dire 10 chiffres). Étant très habitués au système décimal, vous comprenez son fonctionnement de manière intuitive.

Par exemple en base 10, donc le système décimal, le nombre **423** contient **trois unités**, **deux dizaines** et **quatre centaines** : **423 = 4*100 + 2*10 + 3*1**

En base 10, chaque chiffre d'un nombre peut avoir la valeur de 0, 1, 2, 3, 4, 5, 6, 7, 8, ou 9 ; ce qui donne un total de 10 possibilités ; d'où l'appellation base 10. Mais la vraie valeur d'un chiffre dépend de sa position dans le nombre concerné. En partant de droite à gauche, chaque chiffre est multiplié par une puissance de 10 conformément à sa position. Aussi, le chiffre le plus à droite occupe la position 0, le chiffre suivant occupe la position 1, celui d'après la position 2 et ainsi de suite. En suivant ce raisonnement, le nombre 423 peut être décomposé comme suit :

Puissance de 10	10^2	10^1	10^0
Chiffres du nombre	4	2	3
Position des chiffres de droite à gauche	2	1	0

$4*10^2 + 2*10^1 + 3*10^0 = 400 + 200 + 3 = 423$

1.2.1. Représentation des nombres binaires

Étant donné que vous avez été habitués à faire tous vos calculs en base 10, il n'y a aucun intérêt *a priori* pour vous de réfléchir à comment lire et manipuler plus de nombres ; ce qui est naturel ! Cependant, la base 10 est trop compliquée pour les ordinateurs. Rappelez-vous, ils ne sont pas intelligents ! Ils sont conçus pour travailler en base 2.

Le système de numération en ***base 2*** est aussi qualifié de **système binaire**. Dans ce système, deux options sont seulement possibles pour les chiffres : **0** ou **1**. Et pour cause, dans les circuits électroniques numériques qui composent principalement les unités de calcul des ordinateurs, la présence de tension électrique est signalée par le chiffre 1 et l'absence par le chiffre 0. ***C'est la base 2***. Presque tous les ordinateurs modernes utilisent le système binaire car au niveau physique, il est plus facile de gérer seulement deux options pour chaque chiffre.

Remarque : Il existe des ordinateurs réels mais également imaginaires qui utilisent le système de numération ternaire, c'est-à-dire trois valeurs possibles au lieu de deux. Plusieurs scientifiques, ingénieurs et hackers continuent d'explorer les possibilités des ordinateurs en base 3. Vous pouvez consulter utilement les liens suivants : https://fr.qwe.wiki/wiki/Ternary_computer et https://fr.quora.com/Les-ordinateurs-commenceront-ils-jamais-à-utiliser-le-ternaire-au-lieu-du-binaire-Si-oui-pourquoi

Essayons maintenant de représenter le nombre 1101 en base 2 :

Puissance de 2	2^3	2^2	2^1	2^0
Chiffres du nombre	1	1	0	1
Position des chiffres de droite à gauche	3	2	1	0

La version décimale de ce nombre est : $1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 8 + 4 + 0 + 1 = 13$

Maintenant, si nous voulons convertir le nombre 423 de la base 10 en base 2, il suffira de le

décomposer en puissances de 2 comme suit :

$(1 * 256) + (1 * 128) + (0 * 64) + (1 * 32) + (0 * 16) + (0 * 8) + (1 * 4) + (1 * 2) + (1 * 1)$

$1 * 2^8$ $1 * 2^7$ $0 * 2^6$ $1 * 2^5$ $0 * 2^4$ $0 * 2^3$ $1 * 2^2$ $1 * 2^1$ $1 * 2^0$

C'est ainsi qu'on obtient le nombre ***110100111*** en ***base 2*** pour le nombre ***423*** en ***base 10***.

Le terme informatique par lequel on désigne chaque chiffre de la base 2 est **bit** (contraction de binary digit, c'est-à-dire, chiffre binaire). Un ensemble de 8 bits est désigné par **byte** (prononcez ***baïte***) ou **octe**t en français. Un ensemble de 4 bits s'appelle **nibble**.

Entre autres caractéristiques, la capacité mémoire d'un ordinateur permet de se faire une idée de sa puissance. Aussi, il est habituel de parler d'ordinateurs dont les registres des processeurs sont à 32-bits ou 64-bits chacun. On parle de processeur à 32-bits ou 64-bits. Dans le monde des ordinateurs de la marque Apple, c'est le 64-bits qui prédomine. D'ailleurs en novembre 2020, Apple a lancé sa gamme d'ordinateurs Mac dotés de la puce M1que la firme qualifie de révolution dans l'univers des ordinateurs. En effet, cette puce conçue par Apple réunit pas moins de 16 milliards de transistors et intègre le processeur central (CPU), le processeur graphique (GPU), le Neural Engine, les E/S (entrées/sorties) et bien d'autres éléments sur un unique et minuscule circuit intégré. Avec des performances étourdissantes, des technologies sur mesure et la meilleure efficacité énergétique du marché, la puce M1 est plus qu'une simple évolution pour le Mac. Un registre d'un processeur 32-bits peut gérer des nombres dont la valeur maximale en base 10 peut atteindre **4 294 967 295** ; ce qui donne en base 2 le nombre **11111111111111111111111111111111**. Ceci représente un alignement de 32 fois le chiffre 1. Notons qu'il est possible pour un ordinateur de travailler avec des nombres largement plus grands que ce que requiert la capacité de son processeur ; mais dans ce cas, les calculs doivent être compartimentés et prennent beaucoup plus de temps.

1.2.2. Représentation des nombres hexadécimaux

Vous vous doutez certainement qu'il pourrait rapidement devenir très fastidieux de travailler constamment avec des nombres binaires en raison du temps qu'il faut pour les écrire sans compter le risque d'erreur pour un humain. En conséquence, on a souvent recours en programmation informatique à l'usage d'un autre système de numération à savoir le système hexadécimal qui n'est rien d'autre que la ***base 16***.

Bien entendu, nous ne disposons que de 10 chiffres (0, 1, 2, 3, 4, 5, 6, 7, 8 et 9). Pour compléter ces derniers, on utilise les six premières lettres de l'alphabet : a, b, c, d, e et f. Il est trivial que tous les chiffres d'un système de numération d'une ***base x*** sont contenus dans le système de numération de la ***base y*** si ***y > x***. Aussi, dans la base 16 les lettres ***a*** à ***f***, ont pour valeurs comme indiquées dans le tableau ci-contre :

Lettre	a	b	c	d	e	f
Valeur	10	11	12	13	14	15

Intéressons-nous maintenant à l'exemple suivant du nombre ***c0de*** en base 16 en appliquant la même logique de décomposition et de calcul (puissance de 16 conformément aux positions des chiffres dans le nombre à partir de la droite) qu'en base 2 et en base 10.

Puissance de 16	16^3	16^2	16^1	16^0
Chiffres du nombre	c	0	d	e
Valeur du chiffres en base 10	12	0	13	14
Position des chiffres de droite à gauche	3	2	1	0

Remarquons d'abord que dans le système hexadécimal, les nombres ressemblent à des mots ; ce qui laisse un peu de place à l'imagination mais aussi à l'amusement professionnel !

La valeur de chaque chiffre se réfère aux puissances de 16. Ainsi, comme précédemment, on obtient en base 10, l'équivalent suivant :

$12*16^3 + 0*16^2 + 13*16^1 + 14*16^0 = 12*4096 + 0*256 + 13*16 + 14*1 = 49374$

Mais quelle est l'utilité du système de numération hexadécimal en programmation informatique ? En réalité, chaque chiffre hexadécimal peut représenter exactement quatre chiffres binaires. Le nombre binaire ***1111*** est l'équivalent du chiffre hexadécimal ***f***. En conséquence, il devient très pratique de concatener des chiffres binaires en leur équivalent hexadécimal pour créer des nombres hexadécimaux plus compacts que leurs représentations binaires ou décimales. Par exemple, en considérant le nombre c0de ci-dessus, on obtient :

c* = *1100 ; ***0* = *0000*** ; ***d* = *1101*** ; ***e* = *1110*** et ***c0de* = *1100 0000 1101 1110***

À l'évidence, le système de numération hexadécimal s'avère très utile étant donné la façon dont les ordinateurs utilisent de longs nombres binaires 32-bits ou 64-bits. Rappelons que le plus grand nombre binaire 32-bits en base 10 est ***4 294 967 295***. Son équivalent hexadécimal est ***ffffffff*** ; plus court compact et clair.

1.3. Comment écrire du code ?

Comme je l'ai mentionné en début de chapitre, les ordinateurs sont très limités et à eux seuls, ils ne peuvent accomplir qu'un petit nombre de tâches. En revanche, leurs puissances et valeurs ajoutées résident dans le fait de savoir combiner avec élégance et dextérité ces petites tâches pour produire quelque chose de plus important, de créatif et de très utile. Et c'est au développeur que revient ce privilège du codage en grande partie.

Le codage est un peu comme écrire une recette. Vous assemblez les ingrédients (données, variables, constantes, opérateurs, instructions, etc.) et vous confier cette recette à l'ordinateur étape par étape. Supposons par exemple qu'on veuille retoucher une photo et que l'ensemble du processus puisse se décrire et s'organiser en quatre étapes. On pourrait avoir ce qui suit :

```
Étape 1  : Charger la photo depuis un disque dur dans une application appropriée
Étape 2  : Redimensionner la photo à 400 pixels de large par 300 pixels de haut
Étape 3  : Appliquer le filtre «  beauté  » à la photo
Étape 4  : Imprimer la photo
```

Cette description des étapes 1 à 4 se qualifie de **pseudocode**. À l'évidence, cette « recette » n'est pas rédigée dans un langage de programmation valide pour un ordinateur, mais elle représente un **algorithme** que vous voudrez utiliser pour accomplir votre travail de retouche de photo. C'est un algorithme très simple, mais c'en est un quand même !

L'écriture de code en Swift se fait de la même façon : concevoir étape par étape, une liste d'instructions à exécuter par l'ordinateur. Ces instructions deviendront de plus en plus complexes au fur et à mesure de votre progression dans la lecture de ce livre, mais le principe restera toujours le même : vous aller simplement indiquer à l'ordinateur la tâche à accomplir à chaque étape.

Chaque langage de programmation, à un niveau supérieur, est un canal prédéfini pour exprimer ces étapes. Le compilateur sait comment interpréter votre code pour le convertir en instructions exécutables par le processeur. Les langages de programmation ont tous leurs avantages et inconvénients. Swift est un langage extrêmement moderne et puissant. Il reprend à son compte les avantages de la plupart des langages modernes de très haut niveau (Objective-C, Python, Java, Ruby, C# et Haskell) pour les perfectionner tout en reniant leurs faiblesses. Le tableau ci-après résume très brièvement quelques fonctionnalités, non exhaustives, très appréciées du langage Swift :

Fonctionnalité	Description sommaire
Inférence de type	À partir de la valeur initiale d'une variable ou celle d'une constante, Swift peut automatiquement reconnaître et déduire son type.
Programmation générique	Swift donne la possibilité d'écrire une seule fois, un ensemble de codes génériques dont le but est d'exécuter une tâche identique avec différents types de données.
Mutabilité des collections	Swift ne conçoit pas d'objets séparés pour les conteneurs de collections mutables ou non mutables. Au lieu de cela, il faut vous donne la possibilité de préciser la mutabilité en définissant le conteneur comme une constante ou une variable.
Syntaxe des fermetures	Les fermetures sont des blocs autonomes de fonctionnalités qui peuvent être transmis et utilisés dans du code comme variable ou argument de fonction. Ce sont aussi des fonctions sans nom.
Variables optionnelles	Il s'agit de variables dotées de la faculté de posséder ou non une valeur. L'absence de valeur (donc rien) est la valeur **`nil`**.
Instruction à choix multiples	Il s'agit de l'instruction de branchement conditionnel « switch » (commutateur) qui a été considérablement amélioré dans le langage Swift.

Fonctionnalité	Description sommaire
Types ad hoc (Tuple)	Il s'agit d'une structure de données que l'on peut former à la volée selon les besoins contextuels.
Surcharge d'opérateurs	Selon les besoins, les classes les structures peuvent fournir leurs propres implémentations d'opérateurs déjà existants.
Énumérations avec valeurs associées	En Swift les énumérations offrent des fonctionnalités au-delà de la simple définition d'items valeurs.
Protocole et conception orientée protocole	Depuis la version 2 du langage, Apple a introduit le paradigme de la programmation orientée protocole. C'est une nouvelle approche très opérationnelle qui change à la fois la façon de développer des applications mais également la manière de penser la programmation.
Délégation de propriété	Il s'agit d'un mécanisme permettant d'automatiser la logique de définition et d'utilisation des propriétés dont l'usage peut s'avérer répétitif dans le code. Cette approche est intégrée dans le langage depuis la version 5.1.
Surveillance de propriété	Il s'agit d'un mécanisme de déclenchement d'évènement juste avant ou juste après la modification de la valeur d'une propriété d'un type de donnée.
Extension de type	C'est un mécanisme très puissant permettant d'augmenter les fonctionnalités d'un type, qu'on en soit ou non le propriétaire.

1.4. Comment stocker l'information : notions de variable, de type, de valeur et de constante

Dans la section précédente, nous avions décrit en 4 étapes un algorithme en pseudocode. Lorsqu'un développeur d'application conçoit un tel programme, il doit définir les données ainsi que les objets nécessaires à sa réalisation. Cette définition consiste à nommer ces objets et à décrire leur contenu afin qu'ils puissent être stockés en mémoire. C'est pourquoi nous étudions dans cette section ce qu'est une variable et comment la définir. Nous examinons ensuite comment placer une valeur dans une variable par l'intermédiaire de l'instruction d'affectation. Enfin, une clarification sémantique entre valeur et type nous permettra de fixer définitivement l'usage des constantes en Swift.

1.4.1. Notions de variable, de type et de valeur

Une variable permet la manipulation de valeurs (un ensemble de données). Elle est caractérisée par les éléments suivants :

▪ Un **nom**, qui sert à repérer un emplacement en mémoire dans lequel une valeur est placée. Le choix du nom d'une variable est libre. Il existe cependant quelques règles ou contraintes à respecter que je vous présente à la section suivante « Comment nomme-t-on une variable? ».

▪ Un **type**, qui détermine la façon dont la valeur est traduite en code binaire ainsi que la taille de l'emplacement mémoire. Nous examinerons ce concept dans la section 1.4.3 « Qu'est-ce qu'un type ? ». Plusieurs types sont prédéfinis dans le langage Swift et nous en explorerons les plus élémentaires au chapitre 2.

1.4.2. Comment nomme-t-on une variable ?

Le choix des noms de variable n'est pas limité. Il est cependant recommandé d'utiliser des noms évocateurs. Par exemple, les noms des variables utilisées dans une application de gestion de produits seront plus évocateurs comme `article`, `codeBarre`, `prix`, `prixNet` que `xyz1`, `xyz2`, `prix1`, `prix2`. En effet, les premiers évoquent mieux la sémantique de l'information stockée que les seconds. Au moins, les quatre contraintes suivantes sont à respecter dans l'écriture des noms de variables :

1▪ Le premier caractère d'une variable doit obligatoirement être différent d'un chiffre.

2▪ Aucun espace ne doit figurer dans le nom d'une variable.

3▪ Les majuscules sont différentes des minuscules, et tout nom de variable possédant une majuscule est différent du même nom écrit en minuscule. On parle de la sensibilité à la casse ; autrement dit, Swift est un langage sensible à la casse. Dans les exemples de noms de variables cités au paragraphe précédent figurent *codeBarre* et *prixNet*. En Swift, lorsque plusieurs mots composent un non de variable, il est recommandé d'utiliser la « **casse de chameau** » (***Camel case*** en anglais), en faisant référence au dos de chameau, pour l'écrire. Il s'agit d'une pratique qui consiste à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en majuscule la première lettre de chaque mot. En informatique, on distingue le « ***lowerCamelCase*** » où la première lettre du premier mot est en bas-de-casse c'est-à-dire en minuscule; et le « ***UpperCamelCase*** » encore appelé « ***PascalCase*** » où la première lettre du premier mot est en majuscule. Swift recommande donc le lower camel case pour l'écriture des noms de variable.

4▪ Les caractères suivants : &, ˜, «, #, ‘, {,}, (,), [,], -, |, `, \, /, ^, @, =, %, *, ?, :, §, !, <, >, ainsi que ; et , ne peuvent pas être utilisés dans l'écriture d'un nom de variable. Tout autre caractère peut être utilisé, y compris les caractères accentués, le caractère de soulignement (_), les caractères grecs (*µ* par exemple) et les symboles monétaires ($, €, etc.).

Le nombre de lettres ou de mots composant le nom d'une variable est indéfini. ***Néanmoins, l'objectif d'un nom de variable étant de renseigner le développeur sur le contenu de la variable, il n'est pas courant de rencontrer des noms de variables de plus de trente lettres***.

Défi

Parmi les noms de variables suivants, quels sont ceux qui ne sont pas autorisés ou recommandés et pourquoi ?

prix, Tarif, pourquoi#Pas, Num_2, -Plus, +moins, Undeux, testDeControle, 2021Espace, @adresse, VALEUR_temporaire, ah!ha!, Remise$solde.

Solution

Les noms de variables suivants ne sont pas autorisés : *pourquoi#Pas, -Plus, +moins, @adresse, ah! ha!,* car les symboles #, -, +, @ et ! sont interdits dans le nommage des variables.

2021Espace, car il n'est pas possible de placer un chiffre devant le non d'une variable.

Les noms de variables suivants sont admis (ne constituent pas une erreur de programmation en soi !) mais ne sont pas valides car ils ne respectent pas les recommandations et ne s'alignent pas sur la philosophie du langage Swift : *Tarif, Undeux, VALEUR_temporaire, Remise$solde*

Les noms de variables suivants sont autorisés et valides : *prix, testDeControle*

1.4.3. Qu'est-ce qu'un type ?

Un programme doit gérer des informations de natures diverses. Ainsi, les valeurs telles que ***340*** ou ***12.4*** sont de type numérique tandis que ***Einstein*** est un mot composé de caractères, donc une chaîne de caractères. Si l'être humain sait, d'un simple coup d'œil, faire la distinction entre un nombre et un mot, l'ordinateur n'en est pas capable (rappelez-vous qu'il n'a aucune intelligence dans son état brut). Le développeur doit donc « ***signifier*** » à l'ordinateur la nature de chaque donnée. Cela passe par la notion de type.

Le type d'une valeur permet de différencier la nature de l'information stockée dans une variable. Le type de la valeur affectée à la variable correspond donc au type de cette dernière. On pourra donc parler indifféremment du type de la valeur d'une variable ou type de la variable en faisant référence à la même chose. A chaque type sont associés les éléments suivants bien que leurs interactions puissent sembler transparentes de prime abord aux développeurs débutants :

- Un code spécifique permettant la traduction de l'information en binaire et réciproquement.
- Un ensemble d'opérations réalisables en fonction du type de la variable. Par exemple, si la division euclidienne est une opération cohérente pour deux valeurs numériques, elle ne l'est pas pour deux valeurs de type chaîne de caractères.
- Un intervalle de valeurs possibles dépendant du codage utilisé. Par définition, à chaque type correspond un même nombre d'octets et, par conséquent, un nombre limité de valeurs différentes.

1.4.4. Comment fabrique-t-on une variable en Swift ?

La fabrication d'une variable dans un programme se réalise par l'intermédiaire de l'instruction de déclaration de variable. Cette instruction consiste à utiliser conjointement le mot réservé ***var*** et **l'opérateur de désignation du type** de la variable "**:**" selon la syntaxe suivante :

```
var nomDeLaVariable : type
```

ou, dans le cas de plusieurs variables du même type

```
var nomDeLaVariable1, nomDeLaVariable2 : type
```

Voici un exemple :

```
var monTexte : String
```

où *type* correspond dans l'exemple au mot-clé *String* qui est le type des chaînes de caractères en Swift. Pour préciser à l'ordinateur que l'instruction de déclaration est terminée, il n'y a rien d'autre à faire. La fin de la ligne suffit au compilateur pour le savoir. Bien que Swift donne la possibilité de matérialiser cette fin d'instruction par un point-virgule (;), la communauté des développeurs Swift ne s'embarrasse pas à l'utiliser pour des raisons de clarté et de lisibilité du code. L'opérateur "**:**" de désignation du type d'une variable a pour signification « ***est un*** ». Dans l'exemple ci-dessus, la variable monTexte « ***est un*** » String. On parle aussi d'***annotation de type***.

1.4.5. Comment affecter une valeur à une variable en Swift ?

Une fois la variable déclarée, il deviendra nécessaire, avant sa première utilisation, de stocker une valeur à l'emplacement mémoire désigné. Pour ce faire, nous utilisons l'instruction d'affectation qui nous permet d'initialiser (avant la première utilisation) ou de modifier, en cours d'exécution du programme, le contenu de l'emplacement mémoire (le contenu d'une variable n'étant pas, par définition, constant).

L'***affectation*** ou l'***assignation*** de valeur est donc le mécanisme qui permet de placer une valeur dans un emplacement mémoire. On peut utiliser l'une ou l'autre des syntaxes suivantes :

```
var monTexte : String = "Je me forme à la programmation en Swift"   (1)
```

Ou bien

```
var monTexte : String                                               (2)
monTexte = "Je me forme à la programmation en Swift"                (3)
```

Le signe égal "**=**" est l'opérateur d'affectation de valeur en Swift. Il symbolise le fait qu'une valeur soit placée dans une variable. Pour éviter toute confusion sur ce signe mathématique bien connu, nous prendrons l'habitude de le traduire par les termes « ***prend la valeur*** ». Sur la ligne (1), la déclaration de la variable et l'affectation d'une valeur à celle-ci se sont faites simultanément. Une autre possibilité est de procéder en deux temps : déclaration (2) et affectation (3). Dans tous les cas on lit « la variable monTexte ***est un*** String et ***prend la valeur*** 'Je me forme à la programmation en Swift' ».

Le langage Swift est doté d'un mécanisme très puissant que l'on appelle « ***inférence de type*** ». Il s'agit de la capacité du langage à déduire, dans un contexte sans ambiguïté, le type d'une variable à partir de la valeur initiale qui lui a été affectée. Aussi, il est admis ce qui suit :

```
var monTexte = "Je me forme à la programmation en Swift"
```

Et le compilateur comprend automatiquement que `monTexte` est un `String`.

1.4.6. La notion de constante en Swift

Nous avons passé beaucoup de temps autour des variables qui, comme le nom l'indique, changent de valeur au cours de l'exécution du programme. Il existe également des éléments d'information dont la valeur ne change. Par exemple, la constante mathématique **`PI`** ne change jamais. De tels éléments sont appelés ***constantes***.

Absolument tout ce que nous avons vu à propos des variables s'applique aux constantes à l'exception d'une seule chose et pas des moindres : le mot réservé. En Swift le mot réservé au travail avec les constantes est **`let`**.

Bien entendu, comme son nom l'indique, une constante ne peut changer de valeur contrairement à une variable. En vertu de l'inférence de type du langage Swift, on peut déclarer et affecter une valeur à une constante comme suit :

```
let maConstante = "Je suis une chaîne de caractères affectée"
```

Maintenant que nous nous sommes familiarisés avec quelques concepts et notions de base, le moment est venu de commencer par explorer par la pratique le langage Swift dans Xcode.

1.5. L'environnement de développement intégré Xcode

Le logiciel Xcode est l'Environnement de Développement Intégré ou *Integrated Development Environment* (IDE) en anglais fourni gratuitement par Apple pour les développeurs. Il s'agit du principal outil de développement du développeur Swift. Vous commencerez par explorer les parties de Xcode, qui offrent un environnement léger pour expérimenter du code.

Si vous ne l'avez pas déjà fait, téléchargez et installez la dernière version de Xcode disponible pour macOS sur l'App Store.

À l'instar de tous les produits Apple (pardonnez-moi si je vous donne l'impression d'un utilisateur inconditionnel), Xcode est d'une simplicité et d'une aisance en abondance à l'usage. Cependant, il regorge d'outils extrêmement puissants avec une capacité d'extension incroyable permettant au développeur d'étendre les fonctionnalités de Xcode via l'ajout d'applications complémentaires disponibles sur l'App Store. Je suis en train de préparer un livre sur les « ***Secrets de Xcode pour le développeur professionnel*** ». Pour les besoins d'apprentissage du langage Swift, le présent livre se focalise sur **Playgrounds**, comprenez « aires de jeux ».

1.5.1. Les aires de jeu (Playgrounds) dans Xcode

Les ***Playgrounds*** ou ***terrains de jeux*** ont été intégrés à Xcode depuis sa version 6. Il s'agit d'un ensemble d'outils permettant de jouer et de faire des expérimentations avec du code. C'est donc un environnement interactif de travail favorisant l'écriture et l'exécution du code avec un aperçu instantané du résultat. Un terrain de jeu ne nécessite pas que vous compiliez et exécutiez un projet complet. Au lieu de cela, les terrains de jeux évaluent votre code Swift à la volée. Ils sont donc idéaux pour tester et expérimenter le langage Swift dans un environnement léger lors du processus d'apprentissage.

Vous utiliserez fréquemment des terrains de jeux tout au long de ce livre pour obtenir des commentaires rapides sur votre code Swift. Le résultat de chaque modification apportée au code est observé en temps réel sans devoir compiler le code grâce à ce qu'on appelle « ***Boucle de lecture, évaluation, affichage*** » ou « ***Read-Evaluate-Print-Loop*** » (***REPL***) en anglais. Aussi, l'un des objectifs opérationnels des Playgrounds est d'essayer, dans un cadre d'apprentissage, de nouvelles **API** (*Application Programming Interface*), de prototyper de nouveaux algorithmes, de démontrer ou de tester le fonctionnement d'un code, etc. Tout au long de ce livre, nous utiliserons régulièrement des `fichiers.playground` (remarquez l'extension `.playground` au même titre que `.docx` ou `.pdf`) pour montrer et expliquer comment fonctionnent les codes que nous allons écrire ensemble. Par conséquent, avant de vraiment nous lancer dans la programmation avec le langage Swift, passons un peu de temps à apprendre et à nous familiariser avec les outils Playgrounds. Xcode dispose d'un type de document très pratique appelé « playground » (remarquez l'absence de la lettre "s" à la fin du mot playground) qui vous offre la possibilité d'écrire et d'expérimenter très rapidement et très facilement du code sans devoir passer tous les processus de compilation.

1.5.2. Création d'un document de type playground

Une fois Xcode installé, lancez-le. L'écran d'accueil apparaît ; fermez-le. Il a des options qui ne sont pas pertinentes pour le moment. Vous allez créer un document qualifié de "*terrain ou aire de jeu*". Pour générer une aire de jeux dans Xcode, il suffit donc de créer un nouvel environnement Playground sous la forme d'un document playground (à l'instar d'un fichier word) c'est-à-dire un fichier ayant pour extension `.playground`. Pour ce faire, il faut choisir `Nouveau` dans le menu `Fichier`, puis cliquer sur **Playground...** (`Fichier` ▸ `Nouveau` ▸ `Playground...`) ou bien lancer la combinaison simultanée des 4 touches suivantes : `control+command+Shift+N` (`^⌘⇧N`) lorsque l'application Xcode a le focus. La capture d'écran illustrée par la figure 1.2 vous en donne l'aperçu.
Ensuite, la fenêtre illustrée par la figure 1.3 vous apparaîtra pour vous donner la possibilité d'indiquer si votre projet concerne la plateforme macOS, iOS (iPhone et/ou iPad) ou tvOs afin que Xcode charge les bonnes librairies de codes en arrière-plan et de façon totalement transparente pour vous. Pour tous les exemples et démonstrations du présent livre et sauf indication spécifique, peu importera votre choix. La plupart des développeurs règlent leur choix

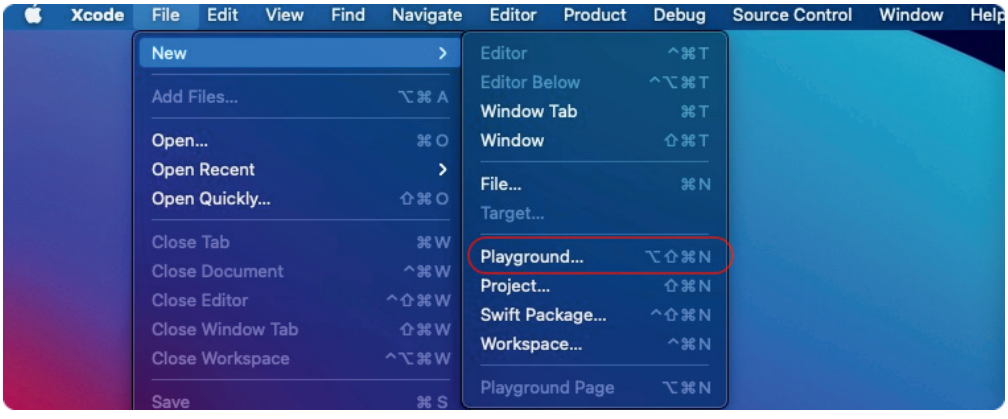

Figure 1.2*: Capture d'écran des séquences de commandes pour créer un Playground*

par défaut sur iOS. Vous devez ensuite choisir le type de document `playground` entre « Blank », « Game », « Map » et « Single View ». Pour nos besoins d'apprentissage dans ce livre, « Blank » conviendra tout le temps car nous n'allons pas écrire du code à une plateforme en particulier mais les principes fondamentaux du langage Swift.

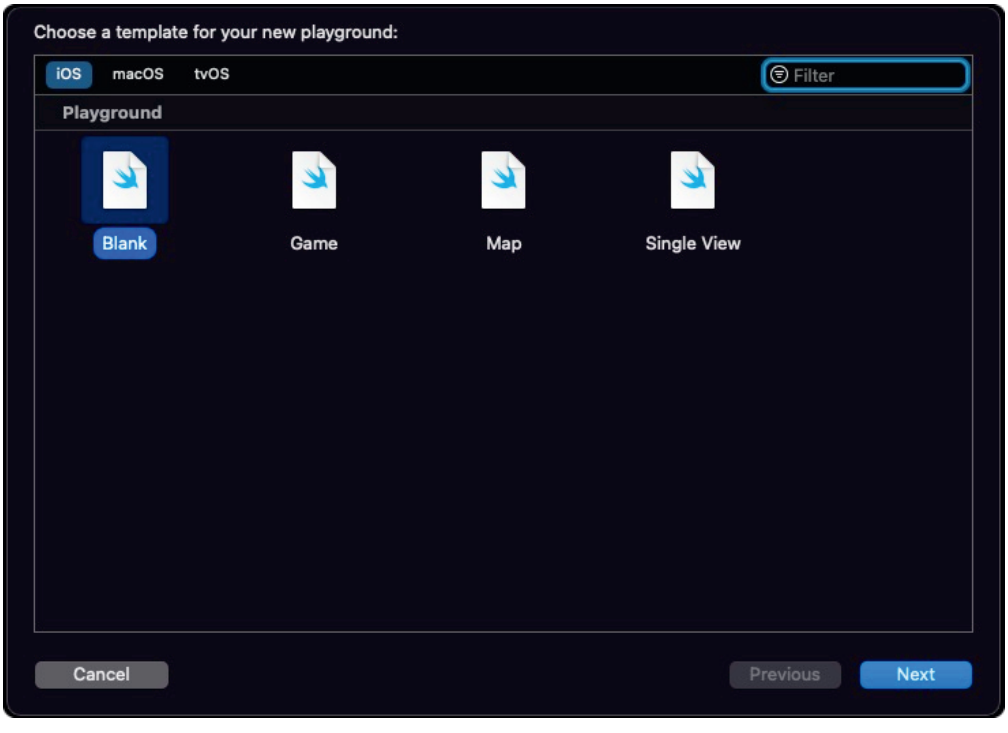

Figure 1.3 *: Capture d'écran pour choisir le modele de document playground*

Cliquez sur « `Next` » puis indiquez le nom de votre projet (Chapitre1 par exemple), choisir le répertoire de destination puis validez en cliquant sur « `Create` ». Lorsque vous créez un document playground, sentez-vous libre de lui attribuer le nom de votre choix. Dans tous les cas, il vaut mieux choisir un nom logique et contextuel. Par exemple, notre premier document playground a pour nom « Chapitre1 » comme montré dans la figure 1.4.

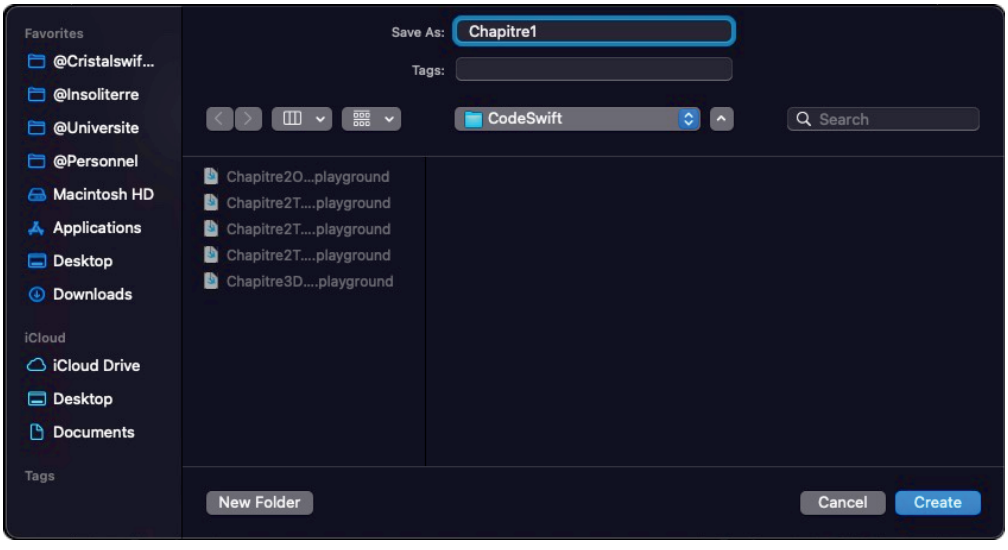

Figure 1.4 *: Capture d'écran pour nommer le document playground (Chapitre1)*

Votre espace de travail, Playground, s'ouvre alors devant vous (figure 1.5).

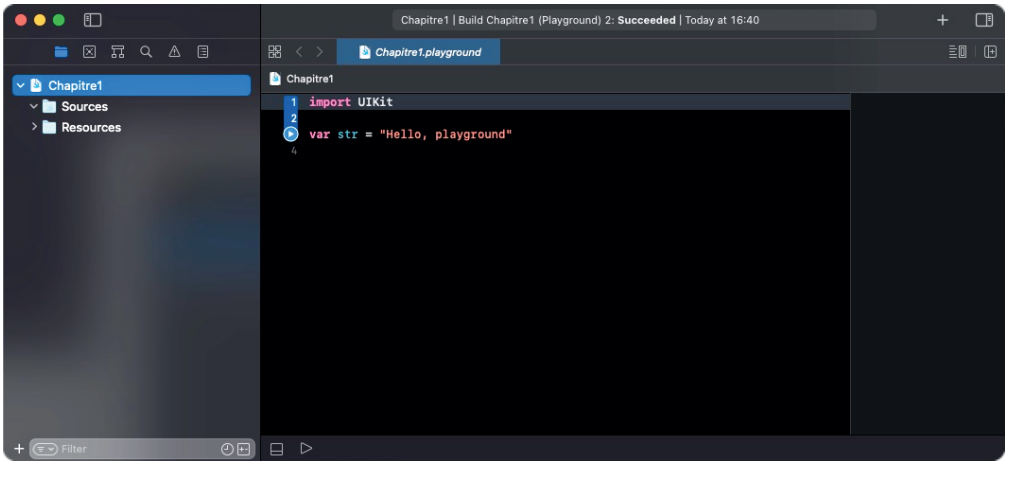

Figure 1.5 *: Aperçu de l'environnement de travail playground vierge (Chapitre1.playground)*

Remarquons que bien qu'on ait choisi un document playground vide, il y quand même deux lignes de code pour démarrer ! Nous allons clarifier tout ceci !

1.5.3. Aperçu de votre environnement de travail Playground

La figure 1.5 montre une nouvelle aire de jeux en Swift. Elle s'ouvre avec trois sections. Sur la gauche se trouve la zone du navigateur. Au milieu, vous avez l'éditeur de code Swift. Et sur la droite se trouve la barre latérale des résultats. Le code dans l'éditeur est évalué et exécuté, si possible, à chaque fois que la source change. Les résultats du code seront affichés dans la barre latérale des résultats.

À première vue, un playground ressemble plutôt à un éditeur de texte très sophistiqué et c'est bien le cas. La capture d'écran précédente met en évidence des fonctionnalités très importantes à connaître afin de maximiser votre productivité. Je vais vous les détailler sur la base de la figure 1.6 qui suit :

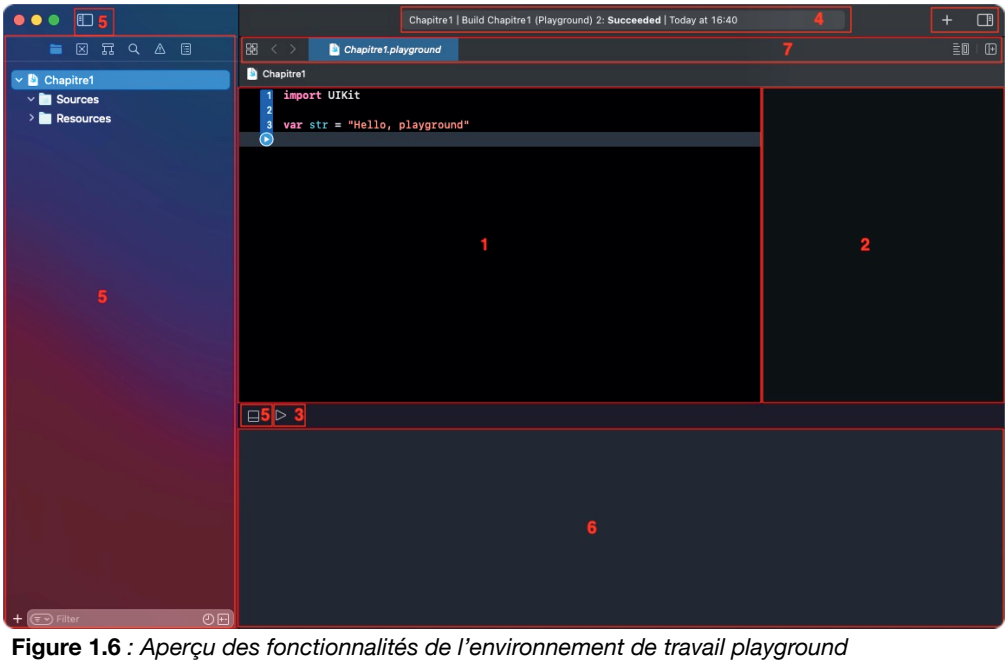

Figure 1.6 *: Aperçu des fonctionnalités de l'environnement de travail playground*

1. Éditeur de code source : c'est la zone dans laquelle vous allez écrire votre code Swift. Cela ressemble beaucoup à un éditeur de texte tel que le `Bloc-notes` ou `TextEdit`. Vous remarquerez l'utilisation de ce que l'on appelle une police à espacement fixe, ce qui signifie que tous les caractères ont la même largeur. Cela rend le code beaucoup plus facile à lire et à formater.

2. Barre latérale des résultats : cette zone affiche les résultats de votre code.

Vous en apprendrez plus sur la manière dont le code est exécuté dans la suite du livre. La barre latérale des résultats sera le principal endroit où vous vérifierez que votre code fonctionne comme prévu.

3. Bouton d'exécution : ce bouton vous permet d'exécuter le fichier `.playground` entier ou d'en effacer l'état afin que vous puissiez l'exécuter à nouveau. Par défaut, l'environnement Playground ne s'exécute pas automatiquement. Vous pouvez modifier ce paramètre pour qu'il s'exécute à chaque changement en appuyant longuement sur le bouton d'exécution et en sélectionnant "Exécuter automatiquement".

4. Visualiseur d'activité : il montre l'état de vie du document playground. Dans la capture d'écran, cela montre qu'il a fini de s'exécuter et est prêt à gérer plus de code dans l'éditeur de source. Lorsqu'il est en cours d'exécution, le visualiseur l'indiquera avec un *spinner* (fileur en français), c'est-à-dire un cercle qui se matérialise par une succession de traits discontinus.

5. Panneaux de commande : ces interrupteurs à bascule affichent et masquent d'autres panneaux. Les panneaux affichent chacun des informations supplémentaires auxquelles vous pourriez avoir besoin d'accéder de temps en temps. Vous les garderez généralement cachés, comme ils le sont sur la capture d'écran. Vous en apprendrez plus sur chacun de ces panneaux en parcourant le livre. Amusez-vous à cliquer respectivement sur les boutons pour observer et analyser le comportement des fenêtres.

6. Fenêtre de débogage : c'est un espace multifonctionnel encore appelé **Console**. En effet, il est également utile de voir les résultats de ce que fait votre code. Dans Swift, vous pouvez y parvenir grâce à l'utilisation de la commande d'impression via une fonction `print()` qui affichera ce que vous voulez dans la zone de débogage. Vous pouvez masquer ou afficher la fenêtre de débogage à l'aide du bouton de commande **5** situé à gauche du bouton d'exécution **3** dans l'image ci-dessus. Vous pouvez également cliquer sur Affichage ▸ Zone de débogage ▸ Afficher la zone de débogage (View ▸ Debug Area ▸ Show Debug Area) pour faire la même chose.

7. Autres panneaux de commandes : ces boutons manifestent leurs utilités dans le cadre de développement d'applications graphique. Pour ne pas alourdir les choses, nous n'en faisons pas cas dans le cadre de ce livre qui doit se concentrer sur l'apprentissage du langage Swift.

1.5.4. Premiers pas avec Swift

En général, vous n'utiliserez pas la zone de navigation de votre environnement de travail Playground lors de votre apprentissage de Swift avec ce livre. Vous pouvez le fermer avec le bouton juste au-dessus dans la barre d'outils de la fenêtre de navigation **5**.

Jetons un œil au code de votre nouveau document playground dans l'éditeur de code. La ligne **1** importe le framework UIKit par défaut. Rappelez-vous dans la figure 1.3 on avait positionné le choix du modèle de document playground à iOS. Si on avait choisi macOS, la ligne **1** aurait importé le framework Cocoa. Vous pouvez essayer tvOS pour voir...

Cette instruction d'importation signifie que votre playground a un accès complet à toutes les interfaces de programmation d'application (API) dans le framework UIKit. (Une API est similaire à une prescription - ou un ensemble de définitions - sur la manière dont un programme peut être écrit.).

Sous l'instruction d'importation se trouve une ligne qui lit `var str = "Hello, playground"`. Cette instruction obéit à tout ce que nous avions déjà vu dans la section 1.4.5 "Comment affecter une valeur à une variable en Swift".

Exécution du code

Un playground est un endroit où vous pouvez écrire et expérimenter le code Swift selon vos conditions. Vous pouvez choisir quand le code que vous écrivez sera réellement exécuté par Xcode. Par défaut, un nouveau playground n'exécutera du code que lorsque vous le lui demanderez. Remarquez le petit bouton de lecture ▶ dans la gouttière de gauche à côté de votre code (Figure 1.5 ou Figure 1.6). Ce symbole signifie que le Playground est actuellement en pause sur cette ligne et ne l'a pas exécutée. Si vous déplacez votre curseur de haut en bas de la gouttière (sans cliquer), le bouton vous suivra. Cliquez sur le bouton de lecture à côté de n'importe quelle ligne et tout le code s'exécutera jusqu'à cette ligne. Dans notre cas, on obtient le résultat comme illustré par la figure 1.7. La déclaration de `str` est évaluée, ce qui fait apparaître sa valeur dans la barre latérale droite.

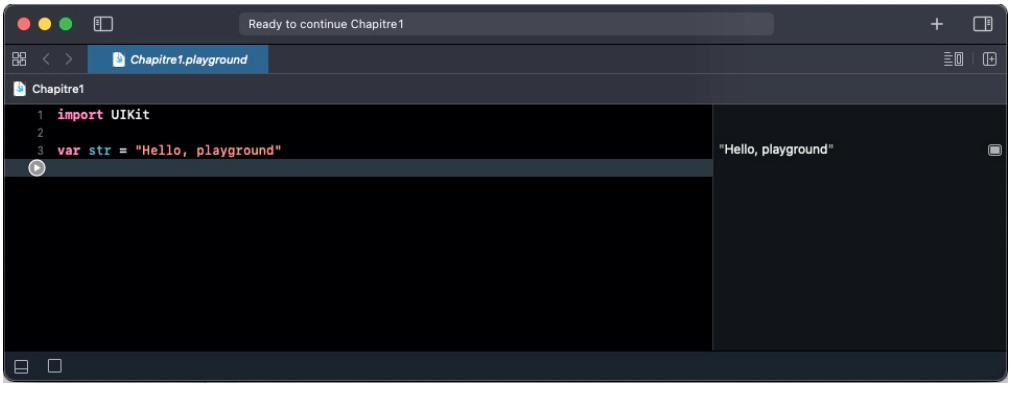

Figure 1.7 *: Aperçu du résultat d'exécution du code dans la barre latérale*

Au lieu de cliquer sur le bouton lecture pour exécuter votre code, vous pouvez aussi positionner le curseur de votre souris à la fin de la portion de code que vous souhaitez exécuter puis actionner un raccourci clavier par la combinaison simultanée des 2 touches Shift⇧ et Retour⏎(⇧+⏎).

L'exécution manuelle de tout ou partie de votre code est une fonctionnalité pratique des Playgrounds lorsque vous explorez par vous-même, mais cela peut devenir fastidieux lorsque vous travaillez sur un livre comme celui-ci.

Bonne nouvelle : vous pouvez dire à Xcode d'exécuter automatiquement votre playground chaque fois que vous apportez des modifications. Pour ce faire, cliquez et maintenez le bouton de lecture (cela peut être un carré si vous venez de gérer votre Playground) en bas à gauche de la fenêtre Playground (Figure 1.8).

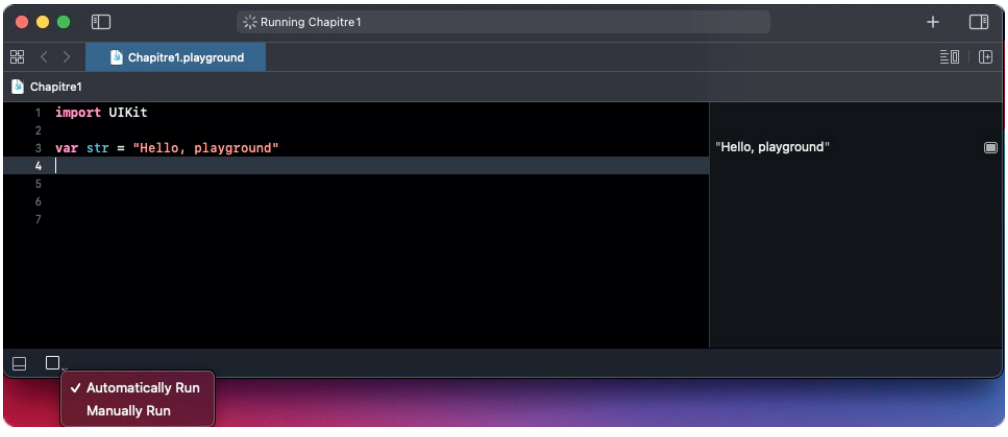

Figure 1.8 *: Paramétrage de l'exécution automatique du code dans Playground*

Dans la fenêtre contextuelle, sélectionnez `Exécuter automatiquement`. Cela amènera Xcode à réévaluer l'ensemble de votre Playground chaque fois que vous apportez des modifications, de sorte que vous n'ayez plus à le faire vous-même.

Dépannage

Xcode est une application comme les autres. Parfois, il a des bugs et d'autres comportements étranges. Il peut arriver des moments où votre Playground se bloque ou arrête de mettre à jour la barre latérale. Si cela vous arrive, l'une de ces étapes de dépannage peut vous aider :

- Fermez et rouvrez votre Playground.
- Quittez et relancez Xcode.
- Remettez le Playground sur Exécuter manuellement et utilisez le bouton de lecture dans la gouttière pour exécuter périodiquement votre code jusqu'à la ligne sélectionnée.
- Copiez votre code dans un nouveau fichier `.playground`.

Ces étapes peuvent également être utiles si vous rencontrez un problème différent avec vos documents `.playground`.

Les commentaires simples de code

Le compilateur Swift génère du code exécutable à partir de votre code source. Pour ce faire, il utilise un ensemble détaillé de règles que vous découvrirez au fur et à mesure dans ce livre.

Parfois, ces détails peuvent obscurcir la vue d'ensemble de la raison pour laquelle vous avez écrit votre code d'une certaine manière ou même du problème que vous résolvez. Pour éviter cela, il est bon de documenter ce que vous avez écrit afin que vos collègues ou vos collaborateurs soient en mesure de comprendre et de donner un sens à votre travail. Cette documentation pourra vous être personnellement utile ultérieurement quand il s'agira de vous replonger dans votre code à toutes fins utiles.

Swift, comme la plupart des autres langages de programmation, vous permet de documenter votre code en utilisant ce qu'on appelle des **commentaires**. Ceux-ci vous permettent d'écrire n'importe quel texte directement à côté de votre code et sont ignorés par le compilateur.

La première façon d'écrire un commentaire est la suivante :

```
// Ceci est un commentaire. Il ne sera pas exécuté !
```

Il s'agit d'un commentaire sur une seule ligne. Il est possible d'empiler des lignes de commentaires pour en faire un paragraphe :

```
// Ceci est un commentaire.
// Il ne sera pas exécuté !
```

Bien entendu, il existe une meilleure façon d'écrire des commentaires sur plusieurs lignes :

```
/* Ceci est aussi un commentaire.
   Organisé sur plusieurs lignes
   Plusieurs lignes
   Des lignes additionnelles */
```

Ce commentaire multi-lignes est délimité par « /* » au début et par « */ » à la fin. Swift donne aussi la possibilité d'imbriquer les commentaires comme ceci par exemple :

```
/* Ceci est aussi un commentaire.
        /* Et à l'intérieur se trouve un autre commentaire */

  Organisé sur plusieurs lignes
 */
```

Cela peut ne pas sembler particulièrement intéressant à condition d'avoir déjà essuyé les frustrations engendrées par l'utilisation d'autres langages qui n'autorisent pas l'imbrication de commentaires. Il existe une multitude de fonctionnalités relatives aux Playgrounds que nous découvrirons au fur et mesure de notre progression. Insérons tous ces exemples de commentaires dans Xcode pour voir ce que ça donne (figure 1.9).

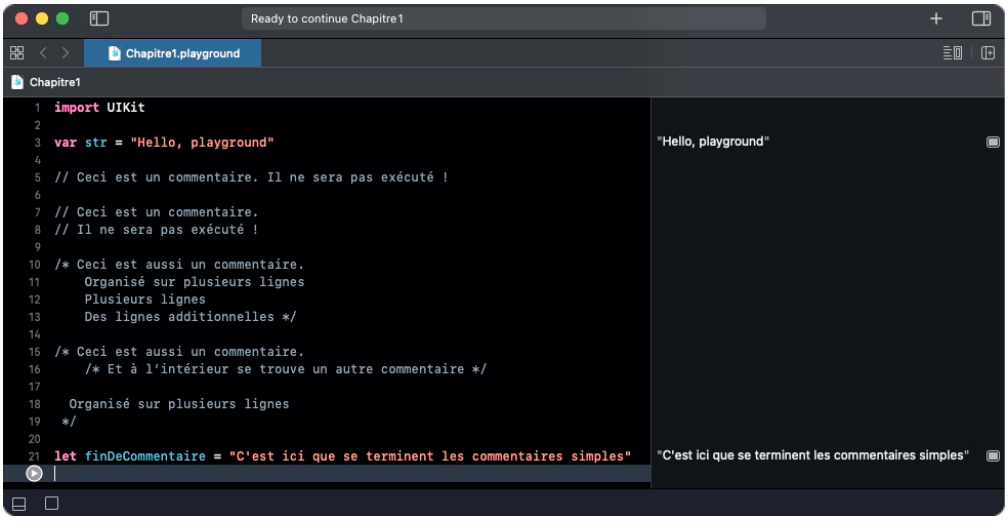

Figure 1.9 *: Exemples de commentaires simples*

Points à retenir

Les ordinateurs, à leur niveau le plus fondamental, exécutent des calculs mathématiques simples.

- Swift est un langage de programmation qui vous permet d'écrire du code, que le compilateur convertit en instructions que le CPU peut exécuter.

- Les ordinateurs fonctionnent avec des nombres en base 2, également appelés nombres binaires.

- L'IDE que vous utilisez pour écrire du code Swift est nommé Xcode et fourni par Apple.

- En fournissant une rétroaction immédiate sur la façon dont le code s'exécute, les Playgrounds vous permettent d'écrire et de tester du code Swift rapidement et efficacement.

- Les constantes et les variables sont caractérisées par un nom et un type. Le nom sert à repérer un emplacement mémoire. Le type détermine la taille de cet emplacement, ainsi que la manière dont l'information est codée, les opérations autorisées et la plage des valeurs représentables.

- Une fois que vous avez déclaré une constante, vous ne pouvez pas modifier ses données, mais vous pouvez modifier les données d'une variable à tout moment.

- Donnez toujours des noms significatifs aux variables et aux constantes pour vous éviter, ainsi qu'à vos collègues, des maux de tête plus tard.

- L'instruction d'affectation permet de placer une valeur dans une variable (ou dans une constante). Pour une variable, elle est de la forme `var maVariable = expression` ou pour une constante, `let maConstante = expression`. Elle calcule d'abord la valeur de l'expression mentionnée à droite du signe =, puis elle l'affecte à la variable ou la constante placée à gauche du signe. Mais que signifie expression dans ces deux formes de définition de l'instruction d'affectation ?

En Swift, comme dans d'autres langages de programmation, ***une expression est une écriture mathématique qui exprime une relation contextuelle entre des variables et/ou des constantes avec des symboles arithmétiques***. Il y a quatre sortes d'expressions : les ***expressions primaires***, les ***expressions en préfixe***, les ***expressions binaires***, et les ***expressions en postfixe***. Conceptuellement, la forme la plus simple d'une expression est l'expression primaire. C'est elle qui donne accès à la valeur d'une constante ou d'une variable. Les expressions binaires et en préfixe s'obtiennent par l'application d'un ou plusieurs opérateurs à des expressions plus petites.

Chapitre 2
Types de données élémentaires et les opérateurs associés

Maintenant que vous avez compris les concepts et les outils associés à un document `.playground` ainsi que la différence entre une variable et une constante, il est temps pour nous de commencer notre voyage exploratoire du langage Swift. Le thème central de ce voyage s'articule autour des types de données élémentaires dans le cadre de ce chapitre. Comme tout langage de programmation, Swift propose un ensemble de types de base permettant la manipulation de valeurs numériques entières, réelles ou autres. Ces types sont :

- Représentés par un mot-clé prédéfini (donc réservé) pour le langage lui-même
- Dits simples ou élémentaires, car, à un instant donné, une variable de type simple ne peut contenir qu'une et une seule valeur.

À l'opposé de ces types, Swift dispose et donne aussi la possibilité de concevoir des types plus élaborés qui permettent le stockage, sous un même nom de variable, de plusieurs valeurs de même type ou non. Il s'agit des tableaux, des dictionnaires, des ensembles et autres structures ainsi que des objets. Nous les étudierons en détail dans la suite du livre.

Formellement, un type, comme nous l'avions vu dans le chapitre précédent, décrit un ensemble de valeurs ainsi que les opérations possibles qui leur sont associées. Vous allez donc travailler avec différents types très pratiques à savoir les chaînes de caractères (String, sur lesquels nous reviendrons en profondeur au chapitre 8) qui permettent de composer du texte, les entiers naturels et relatifs, les nombres rationnels et les nombres réels. Vous apprendrez également les opérations de conversion possible d'un type en un autre tout en mettant en évidence les mécanismes d'inférence et de sécurisation de type qui, au passage, facilitent grandement la vie des développeurs et font de Swift un langage d'exception. Nous terminerons ce chapitre par l'exploration de deux types élémentaires mais spéciaux : les tuples qui vous donnent la possibilité de fabriquer vos propres types ad hoc à partir de plusieurs valeurs de n'importe quel autre type d'une part et les énumérations (dont nous approfondirons le concept et les usages au chapitre 9) d'autre part.

2.1. Caractère et chaîne de caractères

Les informations que doivent traiter les ordinateurs sont composées de nombres, de lettres, de chiffres et des symboles particuliers.

Intuitivement, vous savez que les nombres sont indispensables en programmation, mais ils ne sont pas les seuls types de données dont vos applications auront besoin pour bien fonctionner. Les textes sont également des types de données extrêmement utiles pour indiquer le nom d'une personne et son adresse par exemple. Les langages de programmation stockent du texte par le biais du type de données appelé **String** qui signifie littéralement chaîne de caractères. Mais qu'est-ce qu'un caractère et comment est-il codé dans le langage Swift ? Afin de vous aider à bien comprendre comment bien utiliser les textes en Swift, il est très important de présenter les fondements qui président au concept de `String` et ses relations avec les caractères dont le type en Swift est **`Character`**.

2.1.1. Comment l'ordinateur se représente-t-il le type String ?

Pour l'ordinateur, un texte est une collection ordonnée de caractères, donc une chaîne de caractères représentés par le type `String`. Au chapitre précédent, vous avez appris que les nombres constituent le langage du CPU, et que tout code, dans n'importe quel langage de programmation, peut être réduit à une suite de nombres binaires. Le texte n'en n'est pas moins différent !

Par exemple, lorsque vous appuyez sur une touche de votre clavier, vous communiquez en réalité le nombre associé au caractère de la touche à votre ordinateur. Votre logiciel de traitement de texte convertit ce nombre en une image correspondant au dessin du caractère et finalement vous affiche cette image à l'écran.

Cela pourrait vous paraître étrange, car comment des caractères peuvent être des nombres ? En effet, l'ordinateur a besoin de traduire chaque caractère en langage machine, et cela n'est possible qu'en assignant à chaque caractère un nombre différent. Ce mécanisme constitue une transformation bidirectionnelle entre caractère et nombre ou une table de correspondance qui associe à un caractère donnée une valeur numérique. Cette table est aussi appelée « **Jeu de caractères** ». Pris individuellement et de façon isolée, chaque ordinateur est libre d'adopter le jeu de caractères qui lui convient. Si un ordinateur veut faire correspondre le nombre **10** à la lettre **a**, il le pourrait. Mais qu'adviendrait-il lorsque des ordinateurs doivent communiquer entre eux ? La nécessité d'un jeu de caractères communs s'impose alors !

2.1.2. Le standard Unicode

Afin de résoudre la problématique de l'universalité des jeux de caractères dans un contexte de l'extension mondiale de l'informatique et de la diversité de plus en plus importante des caractères à stocker, plusieurs standards se sont succédé au fil des années. Le standard qui fait actuellement référence en la matière est **Unicode** (**Universal code**) dont la première publication remonte à octobre 1991. La dernière version (au moment où je rédige ce livre), Unicode 13.0, a été publiée en mars 2020. Unicode est un standard informatique permettant des échanges de textes dans différentes langues à l'échelle mondiale. Il est développé par le Consortium Unicode (http://unicode.org).

Son rôle est de veiller au codage du texte écrit en donnant à tout caractère de n'importe quel système d'écriture un nom et un identifiant numérique, et ce de manière unifiée, indépendamment de la plateforme informatique ou le logiciel utilisés.

Concrètement, l'ensemble des caractères provenant des différents alphabets internationaux (européens, africains, asiatiques, etc.) est répertorié dans une charte internationale appelée jeu de caractères Unicode. La table Unicode dresse la liste de tous les caractères utilisés dans le monde et définit pour chacun d'eux une valeur numérique appelée ***code-point***.

Pour connaître le code-point d'un caractère de l'alphabet latin, consultez les fichiers *UnicodeDe000A007F.pdf* et *UnicodeDe0080A00FF.pd*f contenus dans le dossier Unicode placé dans les ressources à télécharger qui accompagnent ce livre. Notez bien ce qui suit :

- Les code-point compris entre 0000 et 001F correspondent aux caractères qui ne peuvent être affichés tels que le caractère de tabulation (HT), le saut de ligne (CR), ou le bip sonore (BEL), etc.

- Les code-point compris entre 0020 et 007F correspondent aux caractères du code **ASCII** (*American Standard Code for Information Interchange*) qui était, avant la mise en place de l'Unicode, le code définissant tout caractère. Dans cet intervalle, tous les caractères de base sont définis, c'est-à-dire l'ensemble des lettres de l'alphabet, en minuscules et en majuscules, ainsi que les signes de ponctuation et les symboles mathématiques.

- Les code-point suivants (de 0080 à FFFF) correspondent à des caractères spéciaux tels que les caractères accentués, les caractères de l'alphabet russe, les idéogrammes chinois, les symboles du copyright et de l'euro, etc.

Considérons par exemple, le mot **cafe**. La table Unicode permet le codage informatique comme suit :

Caractère	**c**	**a**	**f**	**e**
Code-point	99	97	102	101

Bien sûr, le mot **cafe** ne veut pas dire grand-chose en français; en revanche, le mot **café** a bien un sens et son codage informatique en Unicode est :

Caractère	**c**	**a**	**f**	**é**
Code-point	99	97	102	233

C'est ainsi qu'on aurait pu traduire le mot **café** en chinois ou toute autre langue et procéder à son codage Unicode. La même chose est possible avec les ***emoji*** dont la signification est d'origine japonaise où la lettre « ***e*** » signifie image et « ***moji*** » signifie caractère. On comprend maintenant d'où vient le mot ***émoticône*** !

2.1.3. Utilisation des types Character et String en Swift

Swift permet de travailler directement avec les caractères représentés par le type `Character`

et les chaînes de caractères représentés par le type `String`. Avant de commencer à écrire du code qui travaille avec les caractères et les chaînes de caractères, arrêtons-nous brièvement sur leur codage dans la mémoire de l'ordinateur.

En effet, quel que soit le jeu de caractères choisi (alphabet latin, alphabet russe, idéogramme chinois, signes cunéiformes, etc.), chaque caractère est représenté dans la mémoire de l'ordinateur par une valeur numérique unique : cette opération s'appelle le **codage de caractères** ou encore **l'encodage** c'est-à-dire le ***Format de Transformation Unicode*** (UTF, Unicode Transformation Format).

Selon la taille du jeu de caractères utilisé, l'encodage pourra s'effectuer sur 1, 2 ou 4 octets. La valeur numérique attribuée à un caractère est alors calculée à l'aide de son code-point et codée sur 1, 2 ou 4 octets. Ainsi, lorsque l'encodage s'effectue sur :

- 1 octet, on dispose au total de 256 valeurs (2^8). Cette forme d'encodage suffit par exemple pour coder l'ensemble de l'alphabet latin. Par exemple, la lettre **k** a alors **6B** pour code-point.

- 2 octets, le nombre de valeurs disponibles est plus important (2^8x2^8 soit 2^{16} soit 65 536), ce qui permet de coder par exemple la totalité des idéogrammes chinois. Dans ce cas, le caractère **k** aura pour code **00 6B**.

- 4 octets, le nombre de valeurs disponibles s'élève à 2^{32}. Le caractère **k** aura pour code-point **00 00 00 6B**.

Suivant le nombre d'octets utilisés (1, 2 ou 4), ces formes d'encodage se nomment respectivement UTF-8, UTF-16 et UTF32.

Quel est alors l'UTF utilisé par le langage Swift pour l'encodage des caractères et les chaînes de caractères ? En réalité, dans les faits, Swift se comporte de façon totalement agnostique dans ce domaine, et il est le seul langage de programmation à adopter un tel comportement pour des raisons de performance. Nous en détaillons les tenants et les aboutissants au Chapitre 8 : Comment travailler avec les chaînes de caractères ?

À partir de ce point, je vous encourage à lancer Xcode et à créer un document playground auquel pouvez donner le nom ''Chapitre2TypeString'' afin d'exprimer votre apprentissage des types `String` et `Character` en Swift.

Le type `Character` permet de stocker un seul caractère. Par exemple, le code ci-dessous permet de stocker le caractère **a**.

```
let caractereA: Character = "a"
```

Il est possible de stocker n'importe quel caractère y compris les émoji, mais un seul caractère à la fois. Pour faire apparaître la liste des émoji, faites la combinaison des touches ^⌘Espace ou

bien cliquez sur Edit ▸ Emoji & Symbols.

```
let forceEmoji: Character = "💪"
```

En revanche, le type `String` permet de stocker plusieurs caractères en même temps pour former une chaîne de caractères ou un mot ou même un texte long. Par exemple, on peut stocker le mot chien de façon très simple

```
let animal: String = "chien"
```

La partie droite du signe d'affectation =, s'appelle une chaîne de caractères (**String literal** en anglais). Nous venons donc de stocker par affectation à la constante animal, la chaîne de caractères de valeur `chien`. En mettant à profit le mécanisme d'inférence de type du langage Swift, on pourrait réécrire

```
let animal = "chien"
```

⚠️L'inférence de type n'est pas compatible avec le type `Character`. En effet, si vous avez besoin de travailler avec des caractères, il faudra l'expliciter lors de la déclaration comme `let caractereA: Character = "a"`. Il s'agit de l'application de l'annotation de type dont je vous avais parlé dans au chapitre précédent à la fin de la section 1.4.4.

Par exemple, saisissez `let caractereB = "b"` dans Xcode puis pointez le curseur de votre souris sur `caracterB` et appuyez simultanément sur la touche option (⌥) tout en cliquant sur `caracterB`. Vous verrez apparaître une petite fenêtre vous indiquant que `caracterB` est de type `String` (Figure 2.1). Dans ce cas, un caractère est simplement une chaîne de caractères contenant un seul élément. En revanche, si vous cliquez sur `caracterA` tout en appuyant sur la touche option, Xcode vous indique que `caracterA` est de type `Character`.

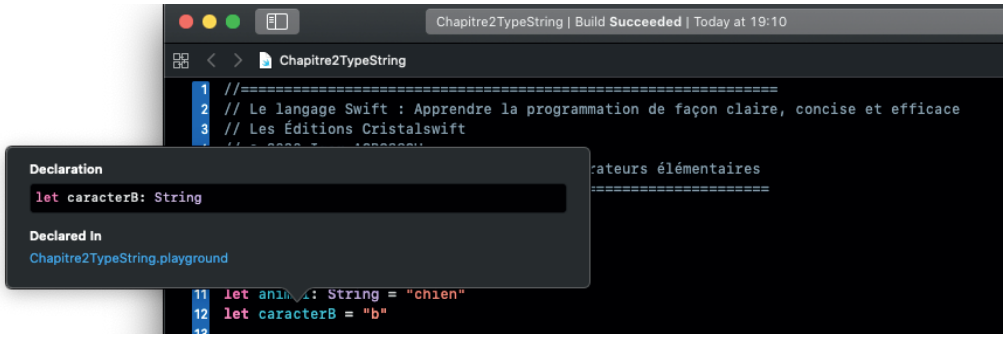

Figure 2.1 *: Aperçu de vérification de type dans Xcode*

L'astuce consistant à cliquer en appuyant sur la touche option sur une constante, une variable ou même un type (essayez de cliquer option sur le type String) permet d'obtenir une documentation rapide (un aperçu) sur l'élément. Nous verrons comment exploiter pleinement

cela à votre profit pour bien documenter votre code.

2.1.4. La concaténation

Vous pouvez faire plus que de créer simplement des chaînes de caractères. Il peut arriver que vous ayez besoin de combiner plusieurs chaînes de caractères pour en faire une nouvelle. Il s'agit de la **concaténation**. En Swift, le recours à l'opérateur arithmétique de l'addition (+) permet de le faire. C'est comme si vous ajoutez une chaîne de caractères à une autre pour en obtenir une nouvelle.

```
var salutations = "Bonjour, " + "je m'appelle "
let prenom = "Igor"
salutations += prenom   // Bonjour, je m'appelle Igor
```

Nous avons besoin de déclarer `salutations` comme une variable et non une constante car sa valeur devra être modifiée. Avec le signe `+` nous assemblons deux chaînes de caractères. Le prénom `"Igor"` est affecté comme valeur à la constante `prenom`. Maintenant, pour modifier le message de `salutations`, l'opérateur composé `+=` est utilisé. Cet opérateur additionne, dans un premier temps la valeur de `prenom` (`Igor`) et de `salutations` (`Bonjour, je m'appelle`) puis, dans un second temps, affecte le résultat (`Bonjour, je m'appelle Igor`) à `salutations` en lieu et place de sa valeur précédente.

Comme vous vous en doutez certainement, il est également possible d'additionner un caractère à une chaîne de caractères. Essayons de terminer le message de `salutations` par un point d'exclamation ! :

```
let exclamation = "!"
salutations += exclamation  // Bonjour, je m'appelle Igor !
```

Rappelez-vous que tel que défini, Swift va traiter `exclamation` comme une chaîne de caractères composée d'un seul élément. Si nous voulons que Swift traite notre point d'exclamation comme un caractère à part entière (donc un type `Character`), il faudrait procéder comme suit :

```
let pointDExclamation: Character = "!"
salutations += String(pointDExclamation)  // Bonjour, je m'appelle Igor !
```

Ici, nous déclarons de façon explicite que `pointDExclamation` est de type `Character`. Pour pourvoir le concatener avec la valeur précédente de `salutations`, il est nécessaire de convertir son type `Character` en type `String`.

2.1.5. La conversion de type

Parfois, vous aurez des informations dans un format donné qu'il faut convertir dans un autre format afin de pouvoir vous en servir. C'est typiquement le cas avec notre `pointDExclamation`. Si on avait procédé comme suit (vous pouvez l'essayer !) :

```
let pointDExclamation: Character = "!"
salutations += pointDExclamation  // ERREUR !
```

Le compilateur se serait plaint du fait de l'inadéquation entre le type de `salutations` (qui est `String`) et celui de `pointDExclamation` (qui est `Character`) en affichant le message « *Cannot convert value of type 'Character' to expected argument type 'String'* ».

Certains langages de programmation ne sont pas aussi stricts que Swift et font ce genre de conversion entre types de façon implicite et silencieuse. L'expérience a prouvé que ce genre de laxisme est source de bugs et de manque de performance. Aussi, Swift ne l'autorise pas.

Rappelez-vous, les ordinateurs sont des automates qui n'exécutent que les ordres du programmeur. En Swift, cela comprend aussi les conversions d'un type à un autre. Retenez donc que **si vous voulez convertir un type de donné en un autre, il faut le faire de manière explicite**. Comme on souhaite que notre `pointDExclamation` soit traité comme une chaîne de caractères, il a fallu faire la conversion de manière explicite : `String(pointDExclamation)`.

```
let pointDExclamation: Character = "!"
salutations += String(pointDExclamation)  // Bonjour, je m'appelle Igor !
```

Dans la suite du chapitre, nous verrons d'autres situations de conversion explicite de types avec les nombres. Mais en attendant, continuons de découvrir d'autres subtilités relatives aux chaînes de caractères.

2.1.6. Interpolation de chaînes

Vous pouvez également construire une chaîne de caractères à partir de plusieurs autres en usant de l'**interpolation de chaînes** (**String interpolation** en anglais), qui est une syntaxe spéciale de Swift vous permettant de faire, en général, de la concaténation de façon plus lisible que précédemment. Autrement dit, ***l'interpolation de chaînes est une manière de produire une nouvelle chaîne de caractères à partir d'un assemblage de constantes, variables, littérales et expressions en intégrant leurs valeurs à une littérale de chaînes de caractères***. Essayons par exemple d'automatiser un message de présentation en fonction du prénom et du pays d'une personne. Dans ce cas, si la personne change, donc changement de prénom et/ou du pays, le message de présentation change également.

```
var personne = "Igor"
var pays = "France"
var presentation = "Bonjour, je m'appelle \(personne) de \(pays)"
```

Vous êtes probablement d'accord pour dire que le message de présentation est beaucoup plus lisible et on comprend tout de suite le rôle et l'intention de chaque partie du code. Il s'agit en effet d'une extension de la syntaxe des chaînes de caractères (*String literal*) où vous avez la possibilité de remplacer certaines parties de la chaîne par d'autres valeurs. Pour ce faire, il suffit de mettre entre parenthèses précédées d'un ***slash arrière*** "\" la valeur à insérer.

Cette syntaxe fonctionne de la même manière lorsque vous désirez construire une chaîne de caractères à partir d'autres types de données comme les nombres par exemple. Admettons que vous veuillez compléter votre message de présentation par l'année en cours.

```
var personne = "Igor"
var pays = "France"
let annee = 2021
var presentation = "Bonjour, je m'appelle \(personne) de \(pays) et nous sommes en \
(annee)"
```

Vous pouvez expérimenter en modifiant les valeurs de `personne`, `pays` et `annee` pour voir...

2.1.7. Les chaînes de caractères multi-lignes

Swift offre également un moyen très élégant d'exprimer une chaîne de caractères disposés en plusieurs lignes. Cette syntaxe peut s'avérer très utile lorsque vous aurez besoin d'inclure un long texte dans votre code par exemple en procédant comme suit :

```
let longueChaine = """
Vous pouvez afficher une
longue chaîne de caractères
disoposés en plusieurs
lignes en faisant ceci
"""

print(longueChaine)
```

Le triple guillemet signifie qu'il s'agit d'une chaîne de caractères disposés sur plusieurs lignes. Dans la pratique, la première et la dernière ligne, donc celles qui comportent les triples guillemets de début et de fin délimitant la chaîne de caractères, ne font pas partie de cette dernière.

Il est très pratique de voir les résultats de ce que fait votre code. Swift permet d'accomplir cela par le biais de la commande **print** implémentée sous la forme de fonctions (notez que j'ai bien mis le mot fonctions au pluriel) `print()`. C'est donc cette commande qui est utilisée pour afficher notre `longueChaine`. Lorsque vous exécutez le code (en cliquant sur le bouton d'exécution situé sur la dernière ligne ou bien en appuyant sur les touches SHIFT "⇧" c'est-à-dire Majuscule et RETOUR "⏎" après avoir positionné après la fonction `print(longueChaine)` sur la même ligne ou en-dessous), le résultat s'affichera dans la console comme illustré par la figure 2.2. À leur niveau le plus fondamental, s'exécutent des calculs mathématiques simples.

Pour vous entraîner (Essayez avant de regarder la correction ci-dessous)

1. Créez une constante de type String appelée `monPrenom` et affectez-lui votre prénom. Créez ensuite, une autre constante du même type appelée `monNom` et affectez-lui votre nom.

2. Créez une constante de type String appelée `monNomEtPrenom` que vous allez initialiser en lui affectant par addition les valeurs de `monPrenom` et `monNom` et les séparant par un espace.

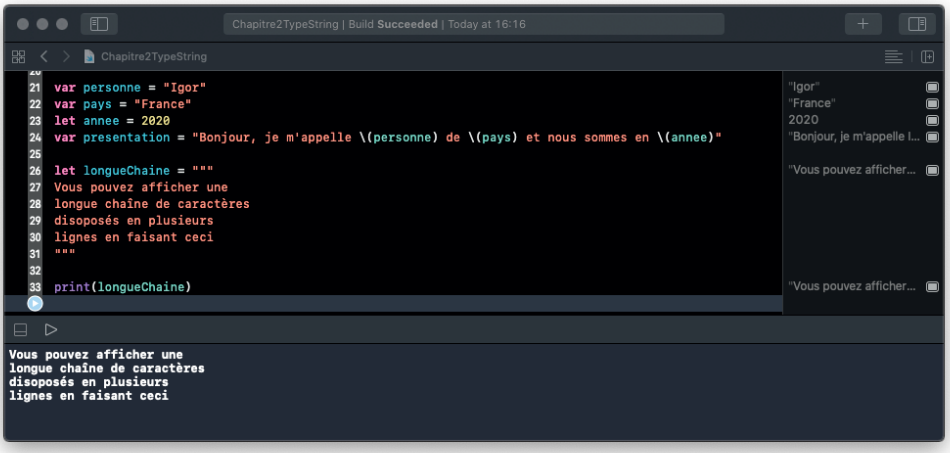

Figure 2.2 *: Illustration des* String *multi-ligne et la commande* print()

3. En utilisant la syntaxe d'interpolation de chaînes, créez une constante de type String nommée meVoici qui utilise la constante monNomEtPrenom pour vous présenter. Par exemple si c'était moi, le résultat donnerait "Bonjour, je m'appelle Igor Agbossou".

Astuce : pour vous faciliter la saisie de la syntaxe d'interpolation de chaines, placez d'abord les parenthèses précédées du caractère *slash* à l'intérieur des guillemets avant de placer le curseur de votre souris entre les parenthèses pour y placer monNomEtPrenom, comme suit : "Bonjour, je m'appelle \()".

```
let monPrenom = "Igor"  // consigne 1
let monNom = "Agbossou" // consigne 1
let monNomEtPrenom = monPrenom + " " + monNom   // consigne 2
let meVoici = "Bonjour, je m'appelle \(monNomEtPrenom)"     // consigne 3
```

2.2. Les types de données numériques

Swift propose tous les types de données numériques standardisés pour stocker et faire toutes sortes d'opérations scientifiques, techniques, comptables, etc. à savoir les entiers naturels (ensemble **N**), les entiers relatifs (ensemble **Z**), les nombres décimaux (ensembles **Q** et **R**) et les nombres complexes (ensemble **C**) avec les relations d'inclusion illustrées par la figure 2.3. Il existe d'autres ensembles mathématiques comme les Quaternions et les Octonions que nous n'abordons pas dans ce livre. Commençons par explorer les nombres entiers.

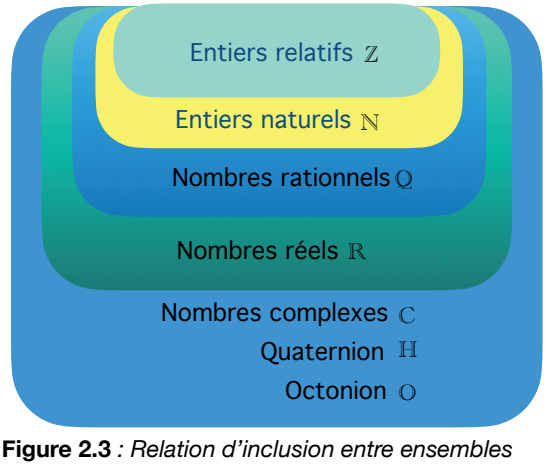

Figure 2.3 *: Relation d'inclusion entre ensembles*

2.2.1. Les types associés aux nombres entiers

Comme vous le savez et comme illustrés par la figure 2.3, les nombres entiers se divisent en deux catégories. Il y a les entiers naturels qui vont de 0 à l'infini positif (+∞) et les entiers relatifs qui vont de l'infini négatif (-∞) à l'infini positif (+∞) en passant par 0. En Swift ainsi que plusieurs autres langages de programmation, les entiers (**Integer** en anglais) sont de types **Int** pour les nombres relatifs et **UInt** (**unsigned integer** c'est-à-dire entier non signé) exclusivement pour les entiers naturels. Vous imaginez que la plage de variation de tous ces entiers est très large et que pour des questions de performance, il ne serait pas judicieux coder un nombre à trois chiffres de la même façon qu'un nombre à cinq chiffres ou un nombre à 10 chiffres ainsi de suite. De plus, il faut tenir compte du fait que, pour la partie négative des entiers relatifs (donc inférieur à 0), chaque nombre est mathématiquement affecté du signe **-** pour matérialiser la nature négative du nombre concerné.

La prise en compte de l'ensemble de toutes ces exigences par le langage Swift aboutit à 10 catégories différentes de types entiers négatifs et positifs dont les caractéristiques sont résumées dans le tableau ci-après :

Ensemble mathématique	Type Swift	Octets pour le codage	Valeur minimale	Valeur maximale
ℤ	Int8	1 Octet	-127	127
	Int16	2 Octets	-32 768	32 767
	Int32	4 Octets	-2 147 483 648	2 147 483 647
	Int64	8 Octets	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
	Int	4 Octets	-2 147 483 648	2 147 483 647
		8 Octets	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
ℕ	UInt8	1 Octet	0	255
	UInt16	2 Octets	0	65 535
	UInt32	4 Octets	0	4 294 967 295
	UInt64	8 Octets	0	18 446 744 073 709 551 615
	UInt	8 Octets	0	18 446 744 073 709 551 615

Vous en conviendrez qu'il y a de quoi stocker tous les résultats de calculs possibles et imaginables. La connaissance des possibilités offertes par chaque type d'entier combinée à une bonne maitrise du domaine d'activité pour laquelle vous développez une application est un gage d'efficacité et de performance. Remarquez que les types `Int64` et `Int` respectivement `UInt64` et `UInt` présentent les mêmes caractéristiques. En effet, `Int` et `UInt` sont respectivement des *alias* (nom d'emprunt) de `Int64` et `UInt64`. On comprend que la logique de nommage de chaque variante `Integer` est de préfixer `Int` (et `UInt`) au nombre de bits ayant servi à son encodage. Swift disposant d'un mécanisme d'affectation d'un nom d'emprunt à une