

# 11X001 – Introduction à la programmation des algorithmes

## 2.1. Types

François Fleuret

<https://fleuret.org/francois>

21 Septembre, 2020

*Le contenu de ce document a été en grande partie  
repris du cours de Jean-Luc Falcone.*



**UNIVERSITÉ  
DE GENÈVE**

FACULTY OF SCIENCE

1. Introduction.
2. Programmation fonctionnelle.
  - 2.1 Types de données.
3. Programmation impérative.
4. Algorithmique.

# Théorie des Types

- Théorie mathématique.
- Inventée au début du 20ème siècle.
- Évite le **paradoxe de Russell**.
- Divise les objets en **termes** et **types**.

Les termes sont des combinaisons de symboles qui respectent des règles d'écriture.

- 3
- $3 + 2$
- $\frac{3}{2}$
- $\{1, 2, 4\}$

Les types sont des collections de termes.

- $3$  : nat
- $3 + 2$  : nat
- $\frac{3}{2}$  : rational
- $\{1, 2, 4\}$  : natSet

Les types sont des collections de termes.

- $3 : \text{nat}$
- $3 + 2 : \text{nat}$
- $\frac{3}{2} : \text{rational}$
- $\{1, 2, 4\} : \text{natSet}$
  
- $\sqrt{2} : \text{real}$
- $\sqrt{\cdot} : \text{real} \mapsto \text{real}$

Dans tous les langages:

- Toutes les données ont un type associé.
- Le type définit et restreint les opérations possibles sur les données.

Par exemple

- $2 + 1 = 3$
- $\sqrt{144} = 12$
- 'bon' + 'jour' = 'bonjour'
- 'bon' + 144 = ???
- $\sqrt{\text{'bonjour'}}$  = ???



Dans la mémoire d'un ordinateur:

- Toutes les données sont une suite de zéros et de uns (bits).
- Ce choix de représentation de l'information découle de la facilité de représenter des valeurs oui/non, présence/absence dans un objet physique.

Dans la mémoire d'un ordinateur:

- Toutes les données sont une suite de zéros et de uns (bits).
- Ce choix de représentation de l'information découle de la facilité de représenter des valeurs oui/non, présence/absence dans un objet physique.
- Ces bits sont organisés par groupes de 8 (octets, ou *bytes*).
- Un octet peut être interprété de plusieurs manières, en particulier comme une écriture en base 2 d'une valeur entière entre 0 et 255.

Dans la mémoire d'un ordinateur:

- Toutes les données sont une suite de zéros et de uns (bits).
- Ce choix de représentation de l'information découle de la facilité de représenter des valeurs oui/non, présence/absence dans un objet physique.
- Ces bits sont organisés par groupes de 8 (octets, ou *bytes*).
- Un octet peut être interprété de plusieurs manières, en particulier comme une écriture en base 2 d'une valeur entière entre 0 et 255.

00000000	0	00000001	1
00000001	1	00000010	2
00000010	2	00000011	3
00000100	4	00000100	4
00001000	8	00000101	5
00010000	16	00000110	6
00100000	32	00000111	7
01000000	64	10000001	129
10000000	128	11111111	255

- Un octet peut également être interprété comme un caractère (chiffre, lettre, symbole) selon une convention spécifique.
- Un groupe d'octets peut être interprété comme une valeur numérique entière sur une plus grande plage de valeurs, ou à virgule, ou comme une suite de caractères.
- Les types spécifient cette interprétation de plus haut niveau.

Par exemple en langage C, qui permet des manipulations de bas niveau:

```
char *s = "Hihi";
int *x = s;
printf("%d %d\n", *x, 72 + 256 * (105 + 256 * (104 + 256 * (105))));
```

1768450376 1768450376

### Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000100	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l

Au niveau des langages on distingue:

- Typage statique vs. typage dynamique.
- Typage fort vs. typage faible.

### Typage statique

- Un type est associé à chaque expression du programme **avant son exécution**.
- Propre des programmes compilés.
- Les types doivent être déclarés ou inférés.
- Par exemple: Java, C, Swift, Pascal, Haskell, Typescript.

```
int x = 3;
```

```
size_t strlen(const char *s);
```

### Typage dynamique

- Un type est associé à chaque expression du programme **pendant son exécution**.
- Propre des programmes interprétés.
- Pas de déclaration de types.
- Par exemple: Python, Scheme, Javascript, Bash.

```
a = "toto"  
a = ma_fonction_bizarre()
```



### Typage fort

- Le type d'une expression est fixé durant l'exécution et doit être **explicitement** modifié.
- Par exemple: Python, Haskell, Swift, OCaml.

```
>>> a = '3'  
>>> b = 5  
>>> a+b  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: can only concatenate str (not "int") to str  
>>> a+str(b)  
'35'
```

### Typage faible

- Le type d'une expression peut changer **implicitement** durant l'exécution.
- Par exemple: Java, C, Perl, PHP.

```
public class Main {  
    public static void main(String[] args) {  
        int a = 3;  
        String s = "4";  
        System.out.println(a + s);  
    }  
}
```

affiche 34 (!)

Plus rigide:

- force prendre les décisions avant,
- plus difficile de changer après (sauf bonne encapsulation).

Plus sûr:

- possibilité d'analyse et de raisonnement,
- vérification statique à la compilation,
- permet d'exprimer des contraintes au niveaux des types.

Au programme de ce cours:

- Types primitifs prédéfinis.
- Optionnels.
- Tableaux.
- Types dérivés définis par l'utilisateur (tuples, structures et énumérations).

En dehors du programme:

- Constructions orientées objets (classes et protocoles).
- Autres structures de données.

## Types primitifs en Swift

### Avertissement

- Pendant ce cours nous ne considérerons qu'un sous ensemble de la syntaxe et des fonctionnalités de Swift.
- Pour réussir l'examen, il est nécessaire de vous limiter à ce sous-ensemble.

On représente les valeurs logiques (dites Booléennes) par le type `Bool`. % Ce type ne peut prendre que deux valeurs: %

- `true`,
- `false`.

On représente les valeurs logiques (dites Booléennes) par le type `Bool`. % Ce type ne peut prendre que deux valeurs: %

- `true`,
- `false`.

Il est utile pour représenter:

- conditions,
- préférences utilisateur,
- validations.

et permet le raisonnement logique.



Plusieurs types représentent les nombres entiers:

Type	Nb. de val.	min	max
Int8	$2^8$	-128	127
UInt8	$2^8$	0	255
Int16	$2^{16}$	-32768	32767
UInt16	$2^{16}$	0	65535
Int32	$2^{32}$	-2 147 483 648	2 147 483 647
UInt32	$2^{32}$	0	4 294 967 295
Int64	$2^{64}$	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
UInt64	$2^{64}$	0	18 446 744 073 709 551 615

- Le nombre représente le nombre de bits utilisés pour coder la valeur. Par exemple: `Int32` signifie “codé sur 32 bits”.
- La présence de `U` indique que l'entier est non signé:
  - positif ou nul,
  - on dispose d'un bit de plus, non-utilisé pour coder le signe.
- On peut aussi utiliser `Int` et `UInt` dont la taille dépend de la plateforme:
  - sur une plateforme 32 bits: `Int32` et `UInt32`,
  - sur une plateforme 64 bits: `Int64` et `UInt64`.

Les nombres décimaux sont représentés par les types à virgule flottante:

- **Float**: 6 chiffres significatifs (32 bits),
- **Double**: 15 chiffres significatifs (64 bits).

Exemple de valeurs:

- 12.5
- -0.02
- 1.23e12
- 0.145e-2

On peut imaginer de nombreuses manière de représenter un nombre à virgule. Swift suit la norme IEEE 754.

**Attention:** il ne s'agit pas de nombres réels!

```
1> 0.5 - (0.25 + 0.25)
```

```
$R0: Double = 0
```

```
2> 0.3 - (0.1 + 0.2)
```

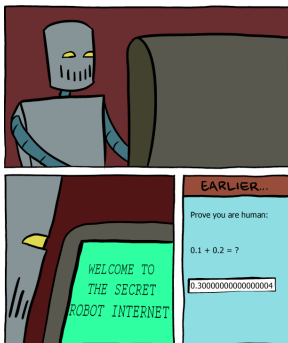
```
$R1: Double = -5.5511151231257827E-17
```

```
3> 0.3 - 0.1 - 0.2
```

```
$R2: Double = -2.7755575615628914E-17
```

**Attention:** il ne s'agit pas de nombres réels!

```
1> 0.5 - (0.25 + 0.25)
$R0: Double = 0
2> 0.3 - (0.1 + 0.2)
$R1: Double = -5.5511151231257827E-17
3> 0.3 - 0.1 - 0.2
$R2: Double = -2.7755575615628914E-17
```



Le type `String` représente du texte **arbitraire** %

- Pas de limitation de taille (mémoire)
- Tous les caractères unicodes: Chiffres, alphabet latin, cyrillique, idéogramme chinois, emojis, ...

% Le type `Character` représente un caractère dans une `String`, sa taille en bits dépend du caractère

**Attention:** Ne jamais utiliser ce type pour autre chose que du texte !

## Question !

Quel type devrait-on utiliser pour représenter le montant d'un transfert d'argent de compte à compte ?

## Types dérivés en Swift

- Un tableau représente une liste de valeurs du même type
  - Sa longueur est arbitraire
  - Un tableau peut être vide
- 
- On note le type d'un tableau: `[T]` où `T` est le type des éléments.
  - Par exemple:
    - `[Double]`: Tableau de nombres à virgule flottantes
    - `[Bool]`: Tableau de valeurs logiques



## Questions !

Que représente le type `[[Int]]` ?

## Questions !

Que représente le type `[[Int]]` ?

On veut représenter un point dans l'espace par le type `[Double]`. Quel est le problème potentiel ?

## Questions !

Que représente le type `[[Int]]` ?

On veut représenter un point dans l'espace par le type `[Double]`. Quel est le problème potentiel ?

On veut représenter les prénoms de personnes par le type `[String]`. Quel est le problème potentiel ?

On peut représenter des valeurs optionnelles par `T?` où `T` est le type de la valeur qui peut être absente.

- Par exemple:
  - Réponse à un sondage (Oui, Non, Sans avis): `Bool?`
  - L'utilisateur peut choisir de donner son adresse email: `String?`
  - On recherche le montant à partir d'un numéro de compte: `Int?`

On peut définir de nouveaux types en agrégeant plusieurs types existants.

Par exemple: couleur

- Modèle additif RGB: Intensité rouge, intensité verte, intensité bleue.

On peut définir de nouveaux types en agrégeant plusieurs types existants.

Par exemple: couleur

- Modèle additif RGB: Intensité rouge, intensité verte, intensité bleue.
- L'intensité est codée de 0 à 255: `UInt8`.

On peut définir de nouveaux types en agrégeant plusieurs types existants.

Par exemple: couleur

- Modèle additif RGB: Intensité rouge, intensité verte, intensité bleue.
- L'intensité est codée de 0 à 255: `UInt8`.
- On pourrait donc représenter une couleur par trois `UInt8`.

On peut définir de nouveaux types en agrégeant plusieurs types existants.

Par exemple: couleur

- Modèle additif RGB: Intensité rouge, intensité verte, intensité bleue.
- L'intensité est codée de 0 à 255: `UInt8`.
- On pourrait donc représenter une couleur par trois `UInt8`.

Question: Pourquoi pas un tableau ?



En Swift, on peut utiliser les types `struct`, `tuple` ou `class`.

Nous ne verrons et n'utiliserons dans ce cours que les deux premiers.

Par exemple: couleur

```
struct Color {  
    let red: UInt8  
    let green: UInt8  
    let blue: UInt8  
}
```

Déclaration de structure: `struct Nom { }`

- `struct` est un mot réservé.
- Le nom de la structure est un `identifiant`.
- Les identifiants sont arbitraires, mais:
  - ne peuvent pas commencer par un chiffre,
  - ne peuvent pas contenir de ponctuation sauf underscore,
  - ne peuvent pas contenir d'espace,
  - débute par convention par une majuscule,
  - doit être unique,
  - la casse compte: `Hello`  $\neq$  `hello`.

## Propriétés (alias membres ou champs)

- La structure peut contenir des propriétés (aussi connues comme champs ou membres).
- Les membres peuvent être de n'importe quel type (non récursif).
- Une propriété se déclare entre les accolades.
- Forme: `let nom: T`
  - `let` est un mot réservé
  - `nom` est un identifiant, par convention commence par une minuscule
  - `T` est le type de la propriété

## Exemple

```
struct Color {  
  let red: UInt8  
  let green: UInt8  
  let blue: UInt8  
}  
  
struct Style {  
  let foreground: Color  
  let background: Color  
}
```

## Exercice: Rectangle

- Représenter un rectangle dans le plan
- Considérer les éléments suivant:
  - Hauteur et largeur, nombre à virgule.
  - Couleur de remplissage, peut-être absente.

## Solution: Rectangle

```
struct Rectangle {  
  let height: Double  
  let width: Double  
  let fillColor: Color?  
}
```

- Représenter un utilisateur défini par:
  - nom d'utilisateur,
  - mot de passe,
  - adresse email, facultative.
- L'adresse email peut avoir été vérifiée ou non

## Solution médiocre: Utilisateur

```
struct User {  
  let username: String  
  let password: String  
  let email: String?  
  let verifiedEmail: Bool  
}
```

Quel est le problème avec cette solution ?



## Meilleure solution: Utilisateur

```
struct Email {  
  let address: String  
  let verified: Boolean  
}  
  
struct User {  
  let username: String  
  let password: String  
  let email: Email?  
}
```

Les tuples sont des “structures anonymes”.

Ils sont définis uniquement par l'ordre des types qui les composent:

```
(UInt8, UInt8, UInt8)
(String, Boolean)
(Int)
(Int, (Double, Double))
()
```

## Question: Feu de circulation

Comment représenter l'état d'un feu de circulation ? (vert, orange, rouge)

Un type ayant un nombre fini de valeurs par être défini avec une **énumération**:

```
enum TrafficLight {  
    case green  
    case yellow  
    case red  
}
```

## Énumérations: Déclaration

Déclaration d'une énumération: `enum Nom { }`

- `enum` est un mot réservé
- Le nom de l'énumération est un **identifiant**

Une/des valeur(s) de l'énumération: `case nom %`

- `case` est un mot réservé.
- `nom` est un identifiant.
- On peut mettre plusieurs valeurs dans une seule déclaration `case`.

## Énumérations: Exemples

```
enum Suit {  
    case hearts, diamonds, spades, clubs  
}
```

```
enum Door {  
    case open  
    case closed  
}
```

```
enum Language {  
    case german, french, italian, other  
}
```

On peut associer un **Tuple** à une ou plusieurs valeurs de l'énumération:

```
enum Language {  
  case german, french, italian  
  case other(String)  
}
```

```
enum Host {  
  case ipv4(UInt8, UInt8, UInt8, UInt8)  
  case ipv6(UInt16, UInt16, UInt16, UInt16, UInt16, UInt16, UInt16, UInt16)  
  case name(String)  
}
```

## Exemple final: Uniform resources indirection (URI)

Spécifications (les crochets désignent un élément optionnel):

```
URI = scheme:[//authority]path[?query][#fragment]  
authority = [userinfo@]host[:port]
```

Par exemple:

- `https://www.unige.ch/sciences/fr/`
- `file:///tmp/downloads/machin.txt`
- `mailto:machin.chose@truc.org`
- `http://127.0.0.1:8080/`



## Exemple final: URI (Solution 1)

```
URI = scheme:[//authority]path[?query][#fragment]  
authority = [userinfo@]host[:port]
```

```
enum Scheme {  
  case http, https, ftp, file, mailto, irc, ssh  
}
```

```
struct Authority {  
  let userinfo: String?  
  let host: Host  
  let port: UInt16?  
}
```

## Exemple final: Uniform resources indirectio (URI)

```
URI = scheme:[//authority]path[?query] [#fragment]  
authority = [userinfo@]host[:port]
```

```
struct URI {  
  let scheme: Scheme  
  let authority: Authority?  
  let path: String  
  let query: String?  
  let fragment: String?  
}
```

FIN