11X001 – Introduction à la programmation des algorithmes

2.6 Fonctions pré-existantes

François Fleuret

https://fleuret.org/francois

13 Octobre, 2020

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

Énumérations avec valeurs associées

Énumérations avec valeurs associées

Nous avons vu comment construire des types sous la forme d'énumérations de valeurs possibles:

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

Énumérations avec valeurs associées

Nous avons vu comment construire des types sous la forme d'énumérations de valeurs possibles:

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

Ces différentes valeurs peuvent être paramétrées par un tuple

```
enum ExtendedInt {
  case minusInf, plusInf
  case value(Int)
}
```

Énumérations avec valeurs associées (cont.)

Dans de nombreux cas, on voudrait traiter les différentes valeurs séparément, *e.g.*

```
func add(x: ExtendedInt, k: Int) -> ExtendedInt {
  if x == ExtendedInt.minusInf {
    return ExtendedInt.minusInf
  } else if x == ExtendedInt.plusInf {
    return ExtendedInt.plusInf
  } else {
    return ExtendedInt.value(v + k)
  }
}
```

mais Swift refuse des comparaisons sur des types définis de cette manière

```
error: binary operator '==' cannot be applied to two 'ExtendedInt' operands
```

Énumérations avec valeurs associées (cont.)

Swift propose la construction `switch` pour traiter les différents cas. Par exemple

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

func string( _ t: TrafficLight) -> String {
  switch t {
  case .green:
    return "Green"
  case .yellow:
    return "Yellow"
  case .red:
    return "Red"
  }
}

let redLight = TrafficLight.red

print(string(redLight))
```

affiche `Red`.

Énumérations avec valeurs associées (cont.)

La construction `switch / case` permet également de récupérer les valeurs associées avec `let`.

```
enum ExtendedInt {
  case minusInf, plusInf
  case value(Int)
}

func add(x: ExtendedInt, k: Int) -> ExtendedInt {
  switch x {
  case ExtendedInt.minusInf:
    return ExtendedInt.minusInf
  case ExtendedInt.plusInf:
    return ExtendedInt.plusInf
  case ExtendedInt.value(let v):
    return ExtendedInt.value(v + k)
  }
}

print(add(x: ExtendedInt.minusInf, k: 5))
print(add(x: ExtendedInt.value(11), k: 5))
```

affiche

```
minusInf
value(16)
```

Énumérations avec valeurs associées (cont.)

Tous les cas doivent être traités, ou bien Swift provoque une erreur de compilation:

```
func add(x: ExtendedInt, k: Int) -> ExtendedInt {
  switch x {
  case ExtendedInt.minusInf:
    return ExtendedInt.minusInf
  case ExtendedInt.value(let v):
    return ExtendedInt.value(v + k)
  }
}
```

```
acces-enum.swift:17:3: error: switch must be exhaustive switch x {
```

Énumérations avec valeurs associées (cont.)

La syntaxe permet de regrouper plusieurs cas en un seul:

```
func add(x: ExtendedInt, k: Int) -> ExtendedInt {
  switch x {
  case ExtendedInt.minusInf, ExtendedInt.plusInf:
    return x
  case ExtendedInt.value(let v):
    return ExtendedInt.value(v + k)
  }
}
```

Gestion des erreurs

Gestion des erreurs

Il arrive souvent que l'on veuille gérer des anomalies exceptionnelles sans modifier en profondeur le programme qui traite le fonctionnement normal.

Une manière brutale de procéder consiste à utiliser

```
assert(condition, message)
```

qui termine le programme avec le message si la condition n'est pas vraie, et indique au système d'exploitation que le programme a échoué.

Gestion des erreurs (cont.)

```
struct Customer {
  let name: String
  let credit: UInt
}

func pay(customer: Customer, amount: UInt) -> Customer {
  assert(customer.credit >= amount, customer.name + " cannot pay")
  return Customer(name: customer.name, credit: customer.credit - amount)
}

let bob1 = Customer(name: "Bob", credit: 125)
let bob2 = pay(customer: bob1, amount: 100)
let bob3 = pay(customer: bob2, amount: 100)
```

```
Assertion failed: Bob cannot pay: file err1.swift, line 8
```

Gestion des erreurs (cont.)

Une alternative en Swift serait d'utiliser les types optionnels et de retourner `nil` quand la fonction rencontre une erreur.

```
func pay(customer: Customer, amount: UInt) -> Customer? {
  if customer.credit >= amount {
    return Customer(name: customer.name, credit: customer.credit - amount)
  } else {
    print(customer.name + " cannot pay")
    return nil
  }
}

let bob1 = Customer(name: "Bob", credit: 125)
let bob2 = pay(customer: bob1, amount: 100)
let bob3 = pay(customer: bob2!, amount: 100)


Bob cannot pay
```

Gestion des erreurs (cont.)

La manière la plus élégante consiste à générer des **erreurs**, qui sont en Swift très proches de ce que d'autres langages appellent des exceptions.

Lorsqu'une fonction produit un erreur, elle s'interrompt, et l'exécution du programme continue dans la fonction appelante.

Cette dernière peut alors traiter l'erreur ou bien elle-même s'interrompre de la même manière.

Gestion des erreurs (cont.)

- Une fonction qui peut renvoyer une erreur doit indiquer `throws` avant le type de retour
- Une erreur peut être générée n'importe quand dans le corps de la fonction avec `throw`.
- Une telle fonction doit être invoquée avec `try` dans une structure `do / catch`.
- Un type d'erreur doit "implémenter le protocole" `Error`.

Gestion des erreurs (cont.)

```
enum TransactionError: Error {
  case negativeAmount
  case negativeCredit
}

func pay(customer: Customer, amount: UInt) throws -> Customer {
  if customer.credit >= amount {
    return Customer(name: customer.name, credit: customer.credit - amount)
  } else {
    throw TransactionError.negativeCredit
  }
}
```

Gestion des erreurs (cont.)

```
do {
  let bob1 = Customer(name: "Bob", credit: 125)
  let bob2 = try pay(customer: bob1, amount: 100)
  let bob3 = try pay(customer: bob2, amount: 100)
  print(bob3)
} catch {
  print("An error occured")
}
```

Gestion des erreurs (cont.)

La clause `catch` peut être restreinte à certaines erreurs.

```
do {
  let bob1 = Customer(name: "Bob", credit: 125)
  let bob2 = try pay(customer: bob1, amount: 100)
  let bob3 = try pay(customer: bob2, amount: 100)
  print(bob3)
} catch TransactionError.negativeCredit {
  print("Someone cannot pay")
}
```

Associer des valeurs aux erreurs permet de fournir des informations sur l'anomalie qui les a provoquées.

```
enum TransactionError: Error {
  case negativeAmount
  case negativeCredit(name: String)
}

func pay(customer: Customer, amount: UInt) throws -> Customer {
  if customer.credit >= amount {
    return Customer(name: customer.name, credit: customer.credit - amount)
  } else {
    throw TransactionError.negativeCredit(name: customer.name)
  }
}

do {
  let bob1 = Customer(name: "Bob", credit: 125)
  let bob2 = try pay(customer: bob1, amount: 100)
  let bob3 = try pay(customer: bob2, amount: 100)
  print(bob3)
} catch TransactionError.negativeCredit(let name) {
  print(name + " cannot pay")
}
```

Gestion des erreurs (cont.)

Une fonction peut se contenter de faire suivre les erreurs provoquées par les fonctions qu'elle-même appelle.

```
func payWithTip(customer: Customer, amount: UInt) throws -> Customer {
  return try pay(customer: customer, amount: (amount * 115) / 100)
}

do {
  let sylviane1 = Customer(name: "Sylviane", credit: 110)
  let sylviane2 = try payWithTip(customer: sylviane1, amount: 100)
  print(sylviane2)
} catch TransactionError.negativeCredit(let name) {
  print(name + " cannot pay")
}
```

Manipulations de tableaux et de chaînes de caractères

Swift offre un grand nombre d'opérations déjà programmées pour manipuler des tableaux, en particulier

- `min` / `max`
- `sorted`
- Plages d'indices.

Les opérations `min`, `max` et `sorted` sont applicable à tout tableaux d'éléments comparables.

```
  1> let v = [ 33, 15, 1007, 12, 21, 29 ]
v: [Int] = 6 values {
  [0] = 33
  [1] = 15
  [2] = 1007
  [3] = 12
  [4] = 21
  [5] = 29
}
  2> v.min()
$R0: Int? = 12
  3> v.max()
$R1: Int? = 1007
  4> v.sorted()
$R2: [Int] = 6 values {
  [0] = 12
  [1] = 15
  [2] = 21
  [3] = 29
  [4] = 33
  [5] = 1007
}
```

Manipulations de tableaux et de chaînes de caractères (cont.)

```
  1> [ "ceci", "est", "un", "tableau", "de", "mots" ]
$R0: [String] = 6 values {
  [0] = "ceci"
  [1] = "est"
  [2] = "un"
  [3] = "tableau"
  [4] = "de"
  [5] = "mots"
}
  2> [ "ceci", "est", "un", "tableau", "de", "mots" ].sorted()
$R1: [String] = 6 values {
  [0] = "ceci"
  [1] = "de"
  [2] = "est"
  [3] = "mots"
  [4] = "tableau"
  [5] = "un"
}
```

Manipulations de tableaux et de chaînes de caractères (cont.)

```
  1> [ true, false ].sorted()
error: repl.swift:1:17: error: referencing instance method 'sorted()' on
'Sequence' requires that 'Bool' conform to 'Comparable'
[ true, false ].sorted()
                ^
```

Les opérations `min`, `max` et `sorted` acceptent un argument `by:` qui spécifie une fonction de comparaison.

```
  1> let v = [ 33, 15, 1007, 12, 21, 29 ]
v: [Int] = 6 values {
  [0] = 33
  [1] = 15
  [2] = 1007
  [3] = 12
  [4] = 21
  [5] = 29
}
  2> v.min(by: { a, b in a%10 < b%10 })
$R0: Int? = 21
  3> v.max(by: { a, b in a%10 < b%10 })
$R1: Int? = 29
  4> v.sorted(by: { a, b in a%10 < b%10 })
$R2: [Int] = 6 values {
  [0] = 21
  [1] = 12
  [2] = 33
  [3] = 15
  [4] = 1007
  [5] = 29
}
```

Swift permet de spécifier des plages d'indices pour la manipulation des tableaux à l'aide des opérateurs `...` et `..<`.

Par exemple:

```
let a = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
print(a[3])
print(a[3...7])
print(a[...4])
print(a[4...])
print(a[3..<7])
```

affiche

```
3
[3, 4, 5, 6, 7]
[0, 1, 2, 3, 4]
[4, 5, 6, 7, 8, 9]
[3, 4, 5, 6]
```

Une chaîne de caractères peut être convertie en type numérique. La valeur obtenue est optionnelle pour traiter les cas où la chaîne ne peut pas être interprétée comme un nombre.

```
  1> Int(123)
$R0: Int = 123
  2> Int("123")
$R1: Int? = 123
  3> Int("le petit chat dort")
$R2: Int? = nil
  4> Int("1e3")
$R3: Int? = nil
  5> Double("123")
$R4: Double? = 123
  6> Double("1e4")
$R5: Double? = 10000
```

Manipulations de tableaux et de chaînes de caractères (cont.)

Le module `Foundation` fournit un grand nombre de fonctionnalités de base pour de opérations fréquentes. En particulier:

- Collections (dictionnaires, ensembles).
- Manipulations de tableaux et de chaînes de caractères.
- Dates et temps.
- Accès aux fichiers et connexions à Internet.

Manipulations de tableaux et de chaînes de caractères (cont.)

```
import Foundation

print("Université de Genève".lowercased())
print("Université de Genève".uppercased())
```

affiche

```
université de genève
UNIVERSITÉ DE GENÈVE
```

Manipulations de tableaux et de chaînes de caractères (cont.)

```
import Foundation

let a = "Le petit chat dort"
print(a)
print(a.replacingOccurrences(of: "chat", with: "chien"))
print("Cette phrase contient les mots chien et cheval".contains("chien"))
```

affiche

```
Le petit chat dort
Le petit chien dort
true
```

Manipulations de tableaux et de chaînes de caractères (cont.)

Il arrive que l'on veuille manipuler les éléments d'une chaîne de caractères comme des éléments d'un tableau. La fonction `components` fait cette conversion.

```
import Foundation

print("Ceci est une suite de mots".components(separatedBy: " "))
```

affiche

```
["Ceci", "est", "une", "suite", "de", "mots"]
```

Fonctions d'entrées / sorties

La fonction `print` accepte deux arguments qui permettent de moduler comment se fait l'affichage:

- `separator: String` spécifie quelle chaîne de caractères doit être ajoutée entre les grandeurs affichées. C'est un espace par défaut.
- `terminator: String` spécifie quelle chaîne doit être affichée pour terminer une ligne. c'est par défaut un retour à la ligne.

Fonctions d'entrées / sorties (cont.)

Avec

```
let nbApples = 17

print("Nous avons", nbApples, "pommes")
print("C'est bien suffisant")
```

affiche

```
Nous avons 17 pommes
C'est bien suffisant
```

En revanche

```
print("Nous avons", nbApples, "pommes", separator: ", ")
print("C'est bien suffisant")
```

affiche

```
Nous avons, 17, pommes
C'est bien suffisant
```

En revanche

```
print("Nous avons", nbApples, "pommes", separator: ", ")
print("C'est bien suffisant")
```

affiche

```
Nous avons, 17, pommes
C'est bien suffisant
```

Et

```
print("Nous avons", nbApples, "pommes", terminator: "! ")
print("C'est bien suffisant")
```

affiche

```
Nous avons 17 pommes! C'est bien suffisant
```

Fonctions d'entrées / sorties (cont.)

La fonction `readLine` permet de saisir une chaîne de caractères au clavier. Elle renvoie une `String?` pour traiter le cas où l'utilisateur interrompt la saisie sans rien entrer.

```
let str = readLine()
print(str ?? "N.A.")
```

La lecture d'un fichier peut se faire simplement en créant une chaîne de caractères.

```
import Foundation

let fileName = "toto.txt"

do {
  let text = try String(contentsOfFile: fileName)
  print(text)
} catch {
  print("Impossible d'ouvrir " + fileName)
}
```

affiche le contenu du fichier `toto.txt` s'il existe, ou bien un message d'erreur.

On peut accéder de manière similaire à des fichiers situés sur des sites web.

```
import FoundationNetworking

let webPage = "https://api.myip.com"

do {
  let url = URL(string: webPage)
  let content = try String(contentsOf: url!)
  print(content)
} catch {
  print("Impossible d'accéder à " + webPage)
}
```

affiche

```
{"ip":"129.194.183.19","country":"Switzerland","cc":"CH"}
```

Exemple: Longueurs des mots d'un texte du web

Longueurs des mots d'un texte du web

Nous pouvons combiner ces éléments pour compter le nombre de mots des différentes longueurs dans la version anglaise de "La République" de Platon.

- Downloader le fichier depuis le web.
- Convertir le texte en liste de mots en minuscules.
- Compter le nombre de mots de chaque longueur.

Longueurs des mots d'un texte du web (cont.)

```
func nbWordsOfLength(_ w: [String]) -> [(Int, Int)] {
  if w == [] {
    return []
  } else {
    let lenMax = w.map({ w in w.count }).max()!

    func nbWordsOfLength(len: Int, r: [(Int, Int)]) -> [(Int, Int)] {
      if len <= lenMax {
        let n = w.filter({ w in w.count == len}).count
        return nbWordsOfLength(len: len+1, r: r + [(len, n)])
      } else {
        return r
      }
    }

    return nbWordsOfLength(len: 1, r: [])
  }
}
```

Longueurs des mots d'un texte du web (cont.)

```
func allLetter(w:String) -> Bool {
  if w == "" {
    return true
  } else {
    return
      w.min()! >= Character("a") && w.max()! <= Character("z")
  }
}
```

Longueurs des mots d'un texte du web (cont.)

```
import Foundation

import FoundationNetworking

let documentAddress = "http://classics.mit.edu/Plato/republic.mb.txt"

do {
  let url = URL(string: documentAddress)
  let content = try String(contentsOf: url!)
  let allWords = content.lowercased().components(separatedBy: " ")
  let words = allWords.filter(allLetter)
  print(words.count, nbWordsOfLength(words))
} catch {
  print("Impossible d'accéder à " + documentAddress)
}
```

affiche

```
90288 [(1, 3136), (2, 21642), (3, 23135), (4, 16084), (5, 9000), (6, 5329),
 (7, 4634), (8, 2731), (9, 2309), (10, 1273), (11, 524), (12, 320),
 (13, 126), (14, 29), (15, 10), (16, 2), (17, 0), (18, 1)]
```

FIN