# 11X001 – Introduction à la programmation des algorithmes

## 3.2 Boucles

François Fleuret

https://fleuret.org/francois

20 Octobre, 2020

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

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

# Les boucles

# Boucles: structures de contrôle

- Les boucles sont une des constructions principales de la programmation impérative.
- L'utilisation d'une boucle permet de répéter un fragment de code.
- Une condition contrôle le nombre de répétitions.
- En général, pour que cette répétition soit utile, il faut que l'état courant puisse changer d'une itération à l'autre.

# Boucles `while`

- La boucle la plus "classique" est la boucle `while`
- Elle répète un bloc de déclarations, tant qu'une expression booléenne est vraie (la condition)
- Syntaxe:

```
while CONDITION {
  DECLARATION 1
  DECLARATION 2
  DECLARATION 3
  ...
}
```

# Exemple 1: Boucle `while`

```
var a = 1

while a <= 1000 {
  print(a)
  a = a * 2
}
```

# Exemple 1: Boucle `while`

```
var a = 1

while a <= 1000 {
  print(a)
  a = a * 2
}
```

affiche

```
1
2
4
8
16
32
64
128
256
512
```

# Exemple 2: Boucle `while`

```
var n = 0
var sum = 0
while n < 4 {
  sum += n
  n += 1
}

print(sum)
```

## Exemple 2: Boucle `while`

```
var n = 0
var sum = 0
while n < 4 {
  sum += n
  n += 1
}

print(sum)
```

affiche

```
6
```

qui correspond bien à $0 + 1 + 2 + 3$.

# Boucles: Modèle d'exécution

On utilisera le modèle d'exécution par environnement utilisé pour la programmation impérative.

Il faut penser à mettre à jour l'environnement à chaque itération de la boucle.

# Boucles: Modèle d'exécution (exemple)

```
var n = 0
var sum = 0

while n < 4 {
  sum += n
  n += 1
}

print(sum)
```

# Boucles: Modèle d'exécution (exemple)

```
var n = 0
var sum = 0

while n < 4 {   // {n=0, sum=0}  [ première itération ]
  sum += n      // {n=0, sum=0}
  n += 1        // {n=1, sum=0}
}

print(sum)
```

# Boucles: Modèle d'exécution (exemple)

```
var n = 0
var sum = 0

while n < 4 {   // {n=1, sum=0}  [ seconde itération ]
  sum += n      // {n=1, sum=1}
  n += 1        // {n=2, sum=1}
}

print(sum)
```

# Boucles: Modèle d'exécution (exemple)

```
var n = 0
var sum = 0

while n < 4 {   // {n=2, sum=1}  [ troisième itération ]
  sum += n      // {n=2, sum=3}
  n += 1        // {n=3, sum=3}
}

print(sum)
```

# Boucles: Modèle d'exécution (exemple)

```
var n = 0
var sum = 0

while n < 4 {   // {n=3, sum=3}  [ quatrième itération ]
  sum += n      // {n=3, sum=6}
  n += 1        // {n=4, sum=6}
}

print(sum)
```

# Boucles: Modèle d'exécution (exemple)

```
var n = 0
var sum = 0

while n < 4 {   // {n=4, sum=6}
  sum += n
  n += 1
}

print(sum)      // {n=4, sum=6}
```

# Boucles imbriquées

Ces structures peuvent évidemment être imbriquées les unes dans les autres.

```
var a, b: Int

a = 1

while a <= 5 {
  b = 1
  while b <= a {
    print(a * b, terminator: " ")
    b += 1
  }
  print()
  a += 1
}
```

affiche

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
```

# Fonction factorielle: Boucle vs. récursion

```
func factorial(n: UInt) -> UInt{
  var prod = 1
  var i = n
  while i > 1 {
    prod *= i
    i -= 1
  }
  return prod
}
```

# Fonction factorielle: Boucle vs. récursion

```
func factorial(n: UInt) -> UInt{
  var prod = 1
  var i = n
  while i > 1 {
    prod *= i
    i -= 1
  }
  return prod
}
```

```
func factorial(n: UInt) -> UInt{
  if n > 1 {
    return n * factorial(n-1)
  } else {
    return 1
  }
}
```

# Fonction factorielle: Boucle vs récursion terminale

```
func factorial(n: UInt) -> UInt{
  var prod = 1
  var i = n
  while i > 1 {
    prod *= i
    i -= 1
  }
  return prod
}
```

```
func factorial(n: UInt) -> UInt{
  func factorial( _ i: UInt, _ prod: UInt) -> UInt {
    if i > 1 {
       return factorial(i - 1, prod * i)
    } else {
       return prod
    }
  }
  return factorial(n, 1)
}
```

# Exemple: Calculer un montant total

```
struct Item {
  let description: String
  let amount: UInt
  let price: UInt
}

func grandTotal( items: [Item]) -> UInt {
  var i = 0
  var result = 0
  while i < items.count {
    result += items[i].price * items[i].amount
    i += 1
  }
  return result
}
```

# Boucles vs. fonctions d'ordre supérieur

```
func grandTotal( items: [Item]) -> UInt {
  var i = 0
  var result = 0
  while i < items.count {
    result += items[i].price * items[i].amount
    i += 1
  }
  return result
}

func grandTotal2( items: [Item]) -> UInt {
  return items.map( {it in it.price*it.amount} )
              .reduce( 0, {i, j in i+j} )
}
```

# Exemple: Tirer au sort deux nombres différents

Version fonctionnelle récursive terminale:

```
func random( max: Int ) -> Int {
  return Int.random(in: 1...max ) // Uniforme dans { 1, 2, ..., max }
}

func randomPair( max: Int ) -> (Int, Int) {
  let x = random(max: max)
  let y = random(max: max)
  if x == y {
    return randomPair (max: max)
  } else {
    return (x, y)
  }
}
```

# Exemple: Tirer au sort deux nombres différents (cont.)

Version utilisant une boucle `while`:

```
func random( max: Int ) -> Int {
  return Int.random(in: 1...max ) // Uniforme dans { 1, 2, ..., max }
}

func randomPair( max: Int ) -> (Int, Int) {
  var x = random(max: max)
  var y = random(max: max)
  while x == y {
    x = random(max: max)
    y = random(max: max)
  }
  return (x, y)
}
```

## Boucles `repeat ... while`

Une boucle `repeat ... while` assure au moins une exécution du bloc de déclarations.

```
repeat {
  DECLARATION 1
  DECLARATION 2
  DECLARATION 3
  ...
} while CONDITION
```

# Boucles `repeat ... while` (cont.)

```
var k = 100
while k <= 10 {  // Faux dès le début, le bloc de
  print("k", k)  // déclarations n'est jamais exécuté
  k += 1
}
```

# Boucles `repeat ... while` (cont.)

```
var k = 100
while k <= 10 {  // Faux dès le début, le bloc de
  print("k", k)  // déclarations n'est jamais exécuté
  k += 1
}

var l = 100
repeat {
  print("l", l)
  l += 1
} while l <= 10  // Faux dès le début, après avoir exécuté
                 // une fois le bloc de déclarations
```

affiche

```
l 100
```

# Exemple: Tirer au sort deux nombres différents (cont.)

Tirage de deux nombres aléatoires différents avec une boucle `repeat ... while`:

```
func randomPair( max: Int ) -> (Int, Int) {
  var x: Int, y: Int
  repeat {
    x = random(max: max)
    y = random(max: max)
  } while x == y
  return (x, y)
}
```

# Que fait ce code

```
func maFonction( _ n: Int ) -> Int {
  var i = 0
  var sum = 0
  while i < n {
    sum += i
  }
  return sum
}
```

## Boucles infinies

Si le programme semble ne pas s'arrêter, il peut se trouver pris dans une boucle infinie... (cause fréquente de bugs)

Ce fonctionnement est parfois souhaitable. Par exemple:

- Application graphique;
- Serveur web

# Boucle "infinie"

```
func maFonction( _ n: Int ) -> Int {
  var i = 0
  var sum = 0
  while true {
    sum += i
    i += 1
    if i == n {
      return sum
    }
  }
}
```

# Boucles sur des collections

# Boucles `for ... in`

Affiche les jours de la liste, un par ligne.

```
let days = [ "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi" ]

var k = 0
while k < days.count {
    print(days[k])
    k += 1
}
```

## Boucles `for ... in`

Swift permet de construire directement une boucle sur un tableau (ou plus généralement une collection) avec le mot-clé `in`.

```
for IDENTIFIANT in COLLECTION {
  DECLARATION_1
  DECLARATION_2
  ...
}
```

# Exemple (1)

Affiche les jours de la liste, un par ligne.

```
let days = [ "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi" ]
for day in days {
    print(day)
}
```

# Exemple (3)

```
func grandTotal( items: [Item]) -> UInt {
  var result = 0
  for item in items {
    result += item.price * item.amount
  }
  return result
}
```

## Fonction zip

Un outil très utile pour faire des boucles sur des tuples est la fonction `zip` qui crée une collection itérable avec `in`.

```
let numbers = [ 1, 2, 3, 4, 5, 6, 7 ]
let names = [ "Prof", "Atchoum", "Dormeur", "Grincheux",
              "Joyeux", "Timide", "Simplet" ]

for n in numbers {
  print(n)
}
```

affiche

```
1
2
3
4
5
6
7
```

## Fonction zip

Un outil très utile pour faire des boucles sur des tuples est la fonction `zip` qui crée une collection itérable avec `in`.

```
let numbers = [ 1, 2, 3, 4, 5, 6, 7 ]
let names = [ "Prof", "Atchoum", "Dormeur", "Grincheux",
              "Joyeux", "Timide", "Simplet" ]

for n in names {
  print(n)
}
```

affiche

```
Prof
Atchoum
Dormeur
Grincheux
Joyeux
Timide
Simplet
```

## Fonction zip

Un outil très utile pour faire des boucles sur des tuples est la fonction `zip` qui crée une collection itérable avec `in`.

```
let numbers = [ 1, 2, 3, 4, 5, 6, 7 ]
let names = [ "Prof", "Atchoum", "Dormeur", "Grincheux",
              "Joyeux", "Timide", "Simplet" ]

for n in zip(numbers, names) {
  print(n)
}
```

affiche

```
(1, "Prof")
(2, "Atchoum")
(3, "Dormeur")
(4, "Grincheux")
(5, "Joyeux")
(6, "Timide")
(7, "Simplet")
```

## Intervalles

Les intervalles que nous avons vu pour sélectionner une partie d'un tableau peuvent également être utilisés pour programmer des boucles sur des séquences d'entiers:

- Intervalle fermé: `4...10` tous les nombres de 4 à 10 inclus
- Intervalle semi-ouvert: `4..<10` tous les nombre de 4 (inclus) à 10 (exclus).

Un intervalle est une collection qui peut se parcourir avec une boucle `for`.

# Intervalles (cont.)

```
for k in [ 11, -2, 5 ] {
  print("Première boucle", k)
}

for k in 1...3 {
  print("Deuxième boucle", k)
}

for k in 1..<5 {
  print("Troisième boucle", k)
}
```

```
Première boucle 11
Première boucle -2
Première boucle 5
Deuxième boucle 1
Deuxième boucle 2
Deuxième boucle 3
Troisième boucle 1
Troisième boucle 2
Troisième boucle 3
Troisième boucle 4
```

# Exemple

Que fait ce programme ?

```
for a in 1...12 {
  for b in 1...12 {
    print(a, "*", b, "=", a * b)
  }
}
```

# Fonction factorielle

```
func factorial(n: UInt) -> UInt {
  var prod: UInt = 1
  for i in 2...n {
    prod *= i
  }
  return prod
}
```

# Ignorer la variable de boucle

```
func maFonction( msg: String ) {
  for i in 1...3 {
    print( msg )
  }
}

maFonction(msg: "toto")
```

# Ignorer la variable de boucle

```
func maFonction( msg: String ) {
  for i in 1...3 {
    print( msg )
  }
}

maFonction(msg: "toto")
```

affiche

```
toto
toto
toto
ignored_exo.swift:2:7: warning: immutable value 'i' was never used; consider replacing
  for i in 1...3 {
      ^
      _
```

## Ignorer la variable de boucle

Un underscore (_) permet d'ignorer l'élément parcouru par l'itération.

```
func maFonction( msg: String ) {
  for _ in 1...3 {
    print( msg )
  }
}
```

# Produit scalaire: boucle vs. fonctions d'ordre supérieur

```
func dot(x: [Double], y: [Double]) -> Double{
  let n = min( x.count, y.count )
  var sum = 0.0
  for i in 0..<n {
    sum += x[i]*y[i]
  }
  return sum
}
```

# Produit scalaire: boucle vs. fonctions d'ordre supérieur

```
func dot(x: [Double], y: [Double]) -> Double{
  let n = min( x.count, y.count )
  var sum = 0.0
  for i in 0..<n {
    sum += x[i]*y[i]
  }
  return sum
}

func dot2( x: [Double], y: [Double]) -> Double {
  return zip(x, y) // Groupe les éléments par pair
         .map({i, j in i*j})
         .reduce(0, {i, j in i+j})
}
```

## Crible d'Érathostène

On peut caractériser un nombre premier par

$$\forall n \in \mathbb{N},\ \big(n \text{ est premier}\big) \iff \big(n > 1 \text{ et } \ \forall k \in \mathbb{N}, \big(k > 1 \text{ et } \ k^2 \leq n\big) \Rightarrow n \notin k\mathbb{N}\big)$$

# Crible d'Érathostène (cont.)

```
func isPrime(_ n: Int) -> Bool {
  if n <= 1 { return false }
  var k: Int = 2
  while k * k <= n {
    if n%k == 0 {
      return false
    }
    k += 1
  }
  return true
}
```

# Crible d'Érathostène (cont.)

```
func isPrime(_ n: Int) -> Bool {
  if n <= 1 { return false }
  var k: Int = 2
  while k * k <= n {
    if n%k == 0 {
      return false
    }
    k += 1
  }
  return true
}

for n in 1...100 {
  if isPrime(n) {
    print(n, terminator: ", ")
  }
}

print("...")
```

affiche

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, ...
```

# Portée lexicale

# Portée Lexicale: environnement d'exécution

```
let a = 100  // Global
let b = 20   // Global
let c = 3    // Global

func chose(x: Int) -> Int { // chose globale, mais x local
  let b = -x                // b local, cache b global
  return b*c
}

let w = chose(x: a) // w global
```

# Portée Lexicale: environnement d'exécution (cont)

Règle d'évaluation:

- Chaque bloc de code définit un nouvel environnement
- L'environnement local est cependant chaîné à l'environnement parent.
- Si une déclaration n'existe pas dans l'environnement en cours, on remonte au parent (puis au grand-parent, etc.)
- L'environnement local est détruit à la fin du bloc de code.

# Exemple: Portée Lexicale

On note `X --> Y` “l’environnement `X` a pour parent l’environnement `Y`”.

```
var x = 10
var y = 5
if x > y {
  var x = 1
  y += 1
  let z = x + y
  x += 1
  print(x)
  print(y)
  print(z)
}
x += 1
print(x)
print(y)
```

# Exemple: Portée Lexicale

On note `X --> Y` “l’environnement `X` a pour parent l’environnement `Y`”.

```
var x = 10       // {x=10}
var y = 5        // {x=10, y=5}
if x > y {
  var x = 1
  y += 1
  let z = x + y
  x += 1
  print(x)
  print(y)
  print(z)
}
x += 1
print(x)
print(y)
```

# Exemple: Portée Lexicale

On note `X --> Y` “l’environnement `X` a pour parent l’environnement `Y`”.

```
var x = 10       // {x=10}
var y = 5        // {x=10, y=5}
if x > y {       // {} --> {x=10, y=5}
  var x = 1      // {x=1} --> {x=10, y=5}
  y += 1         // {x=1} --> {x=10, y=6}
  let z = x + y // {x=1, z=7} --> {x=10, y=6}
  x += 1         // {x=2, z=7} --> {x=10, y=6}
  print(x)       // Affiche 2
  print(y)       // Affiche 6
  print(z)       // Affiche 7
}
x += 1
print(x)
print(y)
```

# Exemple: Portée Lexicale

On note `X --> Y` “l’environnement `X` a pour parent l’environnement `Y`”.

```
var x = 10       // {x=10}
var y = 5        // {x=10, y=5}
if x > y {       // {} --> {x=10, y=5}
  var x = 1      // {x=1} --> {x=10, y=5}
  y += 1         // {x=1} --> {x=10, y=6}
  let z = x + y // {x=1, z=7} --> {x=10, y=6}
  x += 1         // {x=2, z=7} --> {x=10, y=6}
  print(x)       // Affiche 2
  print(y)       // Affiche 6
  print(z)       // Affiche 7
}                // {x=10, y=6}
x += 1           // {x=11, y=6}
print(x)         // Affiche 11
print(y)         // Affiche 6
```

# Exemples / exercices

# Exemple / exercice (1)

```
var a = 3
var b = 1

while a < 5 {
  var b = 1
  print(a, b)
  a = a + 1
  b = b + 1
}
```

# Exemple / exercice (1)

```
var a = 3
var b = 1

while a < 5 {
  var b = 1
  print(a, b)
  a = a + 1
  b = b + 1
}
```

affiche

```
3 1
4 1
```

# Exemple / exercice (2)

```
var a = 3

if a > 0 {
  var b = 1
  while a < 5 {
    print(a, b)
    a = a + 1
    b = b + 1
  }
}

print(a, b)
```

# Exemple / exercice (2)

```
var a = 3

if a > 0 {
  var b = 1
  while a < 5 {
    print(a, b)
    a = a + 1
    b = b + 1
  }
}

print(a, b)
```

affiche

```
env2_exo.swift:12:10: error: use of unresolved identifier 'b'
print(a, b)
         ^
```

# Exemple / exercice (3)

```
for k in -5..<5 {
  for l in -1...1 {
    print(k + l, terminator: " ")
  }
}
```

# Exemple / exercice (3)

```
for k in -5..<5 {
  for l in -1...1 {
    print(k + l, terminator: " ")
  }
}
```

affiche

```
-6 -5 -4 -5 -4 -3 -4 -3 -2 -3 -2 -1 -2 -1 0 -1 0 1 0 1 2 1 2 3 2 3 4 3 4 5
```

# Exemple / exercice (3)

```
let s = 7

for r in 1...s {
  for c in 1...s {
    var a = "."
    if r == 1 ||  r == s || c == 1 || c == s {
      a = "X"
    }
    print(a, terminator: "")
  }
  print()
}
```

# Exemple / exercice (3)

```
let s = 7

for r in 1...s {
  for c in 1...s {
    var a = "."
    if r == 1 ||  r == s || c == 1 || c == s {
      a = "X"
    }
    print(a, terminator: "")
  }
  print()
}
```

affiche

```
XXXXXXX
X.....X
X.....X
X.....X
X.....X
X.....X
XXXXXXX
```

# Exemple / exercice (3)

```
let t = 3
let s = t * 4

for r in 0..<s {
  for c in 0..<s {
    var a = "."
    if (r/t + c/t) % 2==0 {
      a = "X"
    }
    print(a, terminator: "")
  }
  print()
}
```

# Exemple / exercice (3)

```
let t = 3
let s = t * 4

for r in 0..<s {
  for c in 0..<s {
    var a = "."
    if (r/t + c/t) % 2==0 {
      a = "X"
    }
    print(a, terminator: "")
  }
  print()
}
```

affiche

```
XXX...XXX...
XXX...XXX...
XXX...XXX...
...XXX...XXX
...XXX...XXX
...XXX...XXX
XXX...XXX...
XXX...XXX...
XXX...XXX...
...XXX...XXX
...XXX...XXX
...XXX...XXX
```

# Exemple / exercice (3)

```
var x = 0.0
var vx = 3.0

for _ in 0..<12 {
  for _ in 0...UInt(x) {
    print(" ", terminator: "")
  }
  print("O")
  x = x + vx
  vx = vx - 0.6
}
```

# Exemple / exercice (3)

```
var x = 0.0
var vx = 3.0

for _ in 0..<12 {
  for _ in 0...UInt(x) {
    print(" ", terminator: "")
  }
  print("O")
  x = x + vx
  vx = vx - 0.6
}
```

affiche

```
 O
    O
      O
        O
         O
          O
          O
         O
        O
      O
    O
 O
```

FIN