Introduction à la Programmation Objet : Structures de contrôle

Laboratoire d'Intelligence Artificielle
Faculté I&C

J. Sam

Support MOOC

Vidéo et Quiz de :

https://www.coursera.org/learn/initiation-programmation-java/home/week/2

https://www.coursera.org/learn/initiation-programmation-java/home/week/3

- ☞ Semaine 2 et 3
- ☞ Notes de cours et BOOC, semaine 2
- ☞ Notes de cours et BOOC, semaine3

Pendant l'heure de cours

- Petit rappel des points importants
- Fiches résumé : opérateurs et structures de contrôle
- Approfondissements :
  - Évaluation « paresseuse » des expressions (slide 11 chez vous)
  - Du bon usage des booléens (12)
  - Opérateur ternaire (23)
  - Choix multiples (26)
  - Sauts : `break` et `continue` (44)
- Étude de cas (4)
- Une dernière question ... (à découvrir en cours)

Etude de cas

Comment calculer l'expression suivante sans produire d'erreur (par exemple sans « `Nan` », « *Not a number* » ou sans valeurs infinies) ?

$$f(x) = \frac{\sqrt{20 + 7x - x^2}\,\log\left(\frac{1}{x+5}\right)}{\frac{x}{10} - \sqrt{\log\left(x^3 - 3x + 7\right) - \frac{x^2}{5}}}$$

J. Sam

Structures de contrôle

À ce stade du cours, un programme se réduit à une simple **séquence d'instructions** (simples).

... Comment décrire aisément des algorithmes plus complexes ?

☞ **structures de contrôle**

- permettent la représentation d'enchaînements plus complexes
- servent à modifier l'ordre linéaire d'exécution d'un programme

Les structures de contrôle font exécuter à la machine des tâches :

- de façon répétitive,
- ou en fonction de certaines conditions
- . . . ou les deux

Exemple (connu)

```
import java.util.Scanner;

class Degre2 {
  public static void main (String[] args) {

    Scanner keyb = new Scanner(System.in);
    double b=0.0;
    double c=0.0;
    double delta=0.0;

    b = keyb.nextDouble();
    c = keyb.nextDouble();
    delta = b*b - 4.0*c;
      if (delta < 0.0) {
      System.out.println( "Pas de solutions réelles");
       } else if (delta == 0.0) {
      System.out.println ("Une solution unique : " + -b/2.0);
    } else {
      System.out.println("2 solutions: " + (-b-Math.sqrt(delta))
           +  " et " + (-b+Math.sqrt(delta))/2.0);
    }
  }
}
```

données
traitements
structures de contrôle

Opérateurs de comparaison

Les opérateurs de comparaison (relationnels) sont :

`==`	égalité
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Leur résultat est un **booléen** (`true` ou `false`) (expression logique)

Exemples (expressions logiques avec opérateur de comparaison) :

```
x >= y
x != (z + 2)
(x + 4) - z == 5
b = (x == 5);
```

ATTENTION PIÈGE !

Ne pas confondre l'opérateur de test d'égalité `==` et l'opérateur d'affectation `=`

- `x = 3` : **affecte** la valeur 3 à la variable `x`
  (et donc modifie cette dernière)

- `x == 3` : **teste** la valeur de la variable `x`, renvoie `true` si elle vaut 3 et `false` sinon
  (et donc ne modifie pas la valeur de `x`)

Opérateurs logiques

On peut combiner des expressions logiques au moyen d'opérateurs logiques :

`&&`	"et" logique
`||`	ou
$\wedge$	ou exclusif
`!`	négation

*(Remarque : cet opérateur n'a qu'**un seul** opérande)*

Exemples :

- Expression logique utilisant des opérateurs logiques :

```
((z != 0) && (2*(x-y)/z < 3))
```

- Code utilisant des opérateurs logiques :

```
boolean unTest=true;
//...
boolean unAutreTest =
        (x >= 0) || ((x*y > 0) && !unTest);
```

Opérateurs logiques (2)

Les opérateurs logiques `&&`, `||` et `!` sont définis par les tables de vérité usuelles :

`x`	`y`	`!x`	`x && y`	`x || y`	`x ∧ y`
`true`	`true`	`false`	`true`	`true`	`false`
`true`	`false`	`false`	`false`	`true`	`true`
`false`	`true`	`true`	`false`	`true`	`true`
`false`	`false`	`true`	`false`	`false`	`false`

Évaluation paresseuse

Les opérateurs logiques `&&` et `||` effectuent une **évaluation paresseuse** (“*lazy evaluation*”) de leur arguments :

l’évaluation des arguments se fait de la gauche vers la droite et seuls les arguments strictement nécessaires à la détermination de la valeur logique sont évalués.

Ainsi, dans `X1 && X2 && ... && Xn`, les arguments `Xi` ne sont évalués que *jusqu’au 1er argument faux* (s’il existe, auquel cas l’expression est fausse, sinon l’expression est vraie) ;

Exemple : dans `(i != 0) && (3/i < 25)` le second terme ne sera effectivement évalué uniquement si `i` est non nul. La division par `i` ne sera donc jamais erronée.
Et dans `X1 || X2 || ... || Xn`, les arguments ne sont évalués que *jusqu’au 1er argument vrai* (s’il existe, auquel cas l’expression est vraie, sinon l’expression est fausse).

Exemple : dans `(i == 0) || (3/i < 25)` le second terme ne sera effectivement évalué uniquement si `i` est non nul.

Variables Booléennes

Une **variable booléenne** représente une **condition**

☞ Inutile de la comparer explicitement à `true` ou `false` !

Correct :

```
if (unTest)
if (!unTest)
return unTest;
```

Non recommandé :

```
if (unTest == true)
if (unTest != true)
if (unTest == false)
if (unTest != false)
if (unTest) return true;
   else return false;
```

Opérateurs

Operateurs arithmétiques

`*`	multiplication	
`/`	division	
`%`	modulo	
`+`	addition	
`-`	soustraction	
`++`	incrémentation	*(1 opérande)*
`--`	décrémentation	*(1 opérande)*

Operateurs de comparaison

`==`	teste l'égalité logique
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Operateurs logiques

`&&`	"et" logique	
`||`	ou	
`^`	ou exclusif	
`!`	négation	*(1 opérande)*

Priorités (par ordre décroissant, tous les opérateurs d'un même groupe sont de priorité égale) : `!` `++` `-`, `*` `/` `%`, `+` `-`, `<` `<=` `>` `>=`, `==` `!=`, `^` `&&`, `||`

Notation abrégée : `x = x <op> y`, où `<op>` est un opérateur arithmétique, peut aussi s'écrire : `x <op>= y` (exemple : `x += y`)

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

Si $\Delta = 0$
  $x \leftarrow -\frac{b}{2}$
Sinon
  $x \leftarrow \frac{-b-\sqrt{\Delta}}{2}$, $y \leftarrow \frac{-b+\sqrt{\Delta}}{2}$
fin du si

les boucles conditionnelles : *tant que ...*

Tant que réponse non valide
  poser la question
fin de "tant que"

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

$$x = \sum_{i=1}^{5} \frac{1}{i^2}$$

$x \leftarrow 0$
Pour i de 1 à 5
  $x \leftarrow x + \frac{1}{i^2}$
fin du pour

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

les boucles conditionnelles : *tant que ...*

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

Note : on peut toujours (évidemment !) faire des itérations en utilisant des boucles :

$x \leftarrow 0$
$i \leftarrow 1$
Tant que $i \leq 5$
    $x \leftarrow x + \frac{1}{i^2}$
    $i \leftarrow i + 1$
fin de "tant que"

mais conceptuellement (et syntaxiquement aussi dans certains langages) il y a une différence.

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

les boucles conditionnelles : *tant que ...*

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

Les définitions de ces diverses structures de contrôle reposent sur les notions de **condition** et de **bloc** d'instructions.

Une **condition** est une *expression logique* telle que définie au cours précédent.

Instructions composées : les blocs

En Java, les instructions peuvent être **regroupées en blocs**.

Les blocs sont identifiés par des **délimiteurs explicites** de début et de fin de bloc : { }
Exemple de bloc :

```
{
    Scanner keyb = new Scanner(System.in);
    int i ;
    double x ;
    System.out.println("Valeurs pour i et x : ") ;
    i = keyb.nextInt();
    x = keyb.nextDouble();
    System.out.println("Vous avez saisi : i=" + i
                       + ", x=" + x);
}
```


J. Sam

Notion de portée : les blocs

Bloc = séquence d'instructions comprises entre { et }

- Les blocs en Java sont autonomes.
- Ils peuvent contenir leurs propres déclarations/initialisations de variables

Exemple :

```
if (x != 0.0) {
        double y = 0.0; // variable locale
        y = 5.0 / x;
        ...
}
// ici on ne peut plus utiliser y
```

Notion de portée : local/global

1. les variables déclarées à l'intérieur d'un bloc sont appelées **variables locales** (au bloc).
   ☞ ne sont accessibles qu'à l'intérieur du bloc.
2. les variables déclarées en dehors de `main` seront de portée **globales** (à la classe).
   ☞ sont accessibles dans toute la classe.

Note : pour les variables globales, on distingue en Java des variables de classes et des variables d'instances (prochains cours)

Portée : règles

Il ne peut y avoir en Java d'ambiguité sur les noms de variables : on ne peut pas redéclarer localement une variable déjà déclarée plus globalement.

Dans un sous-bloc (bloc imbriqué dans un autre), les références à des variables d'un bloc englobant sont autorisées.

Branchement conditionnel

Le branchement conditionnel permet d'exécuter des traitements selon certaines conditions.

Syntaxe générale :

```
if (condition)
        Instructions1
else
    Instructions2
```

La *condition* est une expression logique. Elle est tout d'abord évaluée puis, si le résultat de l'évaluation est vrai alors la séquence *Instructions1* est exécutée, sinon la séquence *Instructions2* est exécutée.

Instructions1 et *Instructions2* sont :

- soit une **instruction élémentaire**,
- soit un **bloc d'instructions**.

Branchement conditionnel – Exemples

Avec **instructions simples** :

```
if (x != 0.0)
    System.out.println (1.0/x) ;
else
    System.out.println ("erreur : x est nul.") ;
```

Avec **blocs** :

```
if (x > y) {
   min = y;
   max = x;
}
else {
   min = x;
   max = y;
}
```


J. Sam

Branchement conditionnel – Blocs

Conseil : **utilisez toujours la syntaxe avec des blocs**, même si vous n'avez qu'une seule instruction.

- évite les ambiguités
- plus facile pour la maintenance (si l'on doit ajouter des instructions).
  - Dans les slides, on se passera des fois de la syntaxe avec des blocs pour économiser en place...

```
if (x != 0.0) {
System.out.println (1.0/x);
}
else {
System.out.println ("erreur : x est nul.") ;
}
```

☞ En cas de modification ultérieure (ajout d'une instruction), le bloc est déjà présent

J. Sam

Branchement conditionnel – Blocs (2)

Notez également que le second bloc (`else`) est optionnel.

Exemple :

```
if (x < 0.0) {
        x = -x;
}
y = Math.sqrt(x);
```

(calcule $y = \sqrt{|x|}$)

Branchement conditionnel – opérateur ?

Il existe un opérateur ternaire, `?`, permettant d'exprimer de façon consise l'évaluation d'une expression dépendant d'un branchement conditionnel simple :

Syntaxe générale :

```
(expression logique) ? ExpressionVrai : ExpressionFaux;
```

où `ExpressionVrai` est l'expression dont l'évaluation sera retournée si l'expression logique retourne vrai et `ExpressionFaux` est l'expression dont l'évaluation sera retournée sinon.

Exemple d'utilisation :

```
// Affiche la plus grande des deux valeurs
System.out.println((x > y) ? x : y);
```


J. Sam

Branchement conditionnel – opérateur ?

Au lieu d'écrire ceci :

```
if (number % 2 == 0){
        System.out.println("Nombre pair");
}
else {
        System.out.println("Nombre impair");
}
```

On peut écrire :

```
System.out.println((number % 2 == 0) ?
                    "Nombre pair" : "Nombre impair");
```

Branchement conditionnel – Suite

On peut également enchaîner plusieurs conditions :

```
if (condition1)
    Instructions1
 else if (condition2)
    Instructions2
 ...
 else if (conditionN)
    InstructionsN
 else
    InstructionsN+1
```

- *condition1* est tout d'abord évaluée puis, si le résultat de l'évaluation est vrai alors la séquence *Instructions1* est exécutée, sinon *condition2* est évaluée, et ainsi de suite ...
- Si aucune des conditions n'est vérifiée, la séquence *InstructionsN+1* est exécutée.

Choix multiples (la version traditionnelle)

On peut écrire de façon plus claire l'enchaînement de plusieurs conditions dans le cas où l'on teste différentes valeurs d'une expression

Avec **`if ..else`**

```
if (i == 1)
   Instructions1
else if (i == 12)
   Instructions2
else if ...
else
   InstructionsN+1
```

Avec **`switch`**

```
switch (i)
{
   case 1:
      Instructions1
      break;
   case 12:
      Instructions2
      break;
   case ...
   default:
      InstructionsN+1
}
```

☞ chaque `case` correspond à une constante **`int`** (ou équivalent) ou **`char`** (**`String`** aussi depuis Java 7)

`InstructionsJ` : instruction élémentaire ou bloc d'instructions

`instructionsJ` : instruction élémentaire

To `break` or not to `break` ...

Attention Si l'on ne met pas de `break`, l'exécution ne passe pas à la fin du `switch`, mais continue avec les instructions du `case` suivant :

```
switch (a+b) {
    case 0: instruction1; // execution uniquement
        break;            // quand (a+b) vaut 0
    case 2:
    case 3: instruction2; // quand (a+b) vaut 2 ou 3
    case 4:
    case 8: instruction3; // quand (a+b) vaut 2, 3, 4
        break;            //  ou 8
    default: instruction4; // dans tous les autres cas
}
```

`switch` : un exemple

Soit l'enchaînement de conditions suivant :

```
System.out.print("Entrez un entier: ");

int a = keyb.nextInt();

if (a == 0)
    System.out.println("To break");
else
    if (a == 1)
        System.out.println("or not");
    else
        if (a == 2)
            System.out.println("to break");
        else
            System.out.println("that is the question");
```

Exercice : essayons de l'exprimer au moyen d'un `switch` ...

J. Sam

Avec `break`

Code

```
System.out.print("Entrez un entier: ");
int a = keyb.nextInt();

switch (a) {
case 0 :
  System.out.println("To break");
  break;
case 1 :
  System.out.println("or not");
  break;
case 2 :
  System.out.println("to break");
  break;
default :
  System.out.println
    ("that is the question");
}
```

Exécution

```
Entrez un entier: 0
To break

Entrez un entier: 1
or not

Entrez un entier: 99
that is the question
```

Sans `break`

Code

```
System.out.print("Entrez un entier: ");
int a = keyb.nextInt();

switch (a) {
case 0 :
  System.out.println("To break");
case 1 :
  System.out.println("or not");
case 2 :
  System.out.println("to break");
default :
  System.out.println
   ("that is the question");
}
```

Exécution

```
Entrez un entier: 99
that is the question

Entrez un entier: 2
to break
that is the question

Entrez un entier: 0
To break
or not
to break
that is the question
```

`switch` VS `if..else`

`switch` est moins général que `if..else` :

- La valeur sur laquell on teste doit être soit `char` ou `int`, `byte`, `short` ... ou `String)` (mais ce dernier type seulement depuis Java 7)
- Les cas **doivent être des constantes** (pas de variables)

`switch` et Java >=14

- l'utilisation de `->` à la place de `:` permet d'éviter l'utilisation du `break`
- un cas peut être relatif à *plusieurs constantes*, séparées par des virgules
- l'instruction `switch` peut retourner un résultat.

Une fois maîtrisé le `switch` conventionnel, jetez un oeil à ce petit tutoriel très explicite : https://koor.fr/Java/Tutorial/java_switch_se_14.wp

Java 17 (et 21) parachèvent cet outillage avec des nouveautés puissantes (switch sur des objets, «pattern matching» etc.) qui ne seront d'intérêt que plus tard (abordé au second semestre)

Attention !Le correcteur du MOOC II est encore sur la version 12 de Java

Boucles

Les boucles permettent la mise en œuvre **répétitive** d'un traitement.
La répétition est **contrôlée** par une **condition de continuation**.

On distingue deux types de boucles.

1. boucle avec condition de continuation *a priori* (on veut **tester** la condition **avant d'exécuter** les instructions).
2. boucle avec condition de continuation *a posteriori* (on veut **exécuter** les instructions au moins une fois **avant de tester** la condition).

Boucles – Condition de continuation *a priori*

Syntaxe générale :

```
while (condition)
    Instructions
```

Tant que la condition de continuation est vérifiée, les instructions sont exécutées.

Instructions est soit une **instruction élémentaire**, soit un **bloc d'instructions**. Il est conseillé, de nouveau, de toujours utiliser la syntaxe par bloc.

Boucles – Condition de continuation *a posteriori*

Syntaxe générale :

```
do
    Instructions
while (condition);
```

Les instructions sont exécutées **jusqu'à ce que** la condition de continuation soit fausse (et au moins une fois au départ, indépendamment de la valeur de la condition).

Instructions est soit une **instruction élémentaire**, soit un **bloc d'instructions**. Il est conseillé, de nouveau, de toujours utiliser la syntaxe par bloc.

Boucles : Exemple

Exercice : qu'affiche le code suivant ?

```
int i=5;
while (i > 1) {
        System.out.print(i + " ");
        i = i / 2 ;
}
```

Réponse :

```
5 2
```

Exécution pas à pas de l'exemple

Instruction	effet	i
`int i=5;`	nouvelle variable `i`	5
`while (i > 1)`	teste `i > 1` → `true` ⇨ entre dans la boucle	5
`System.out.print(i + " ")`	affiche 5	5
`i = i / 2 ;`	i = 5 / 2 = 2 (division **entière** !)	2
`while (i > 1)`	teste `i > 1` → `true` ⇨ continue dans la boucle	2
`System.out.print(i + " ")`	affiche 2	2
`i = i / 2 ;`	i = 2 / 2 = 1	1
`while (i > 1)`	teste `i > 1` → `false` ⇨ sort de la boucle	1

Boucles : Exemple 2

Exercice : qu'afficheront les codes suivants ?

```
int i=0;
while (i > 1) {
    System.out.print(i);
    i = i / 2 ;
}
```

n'affichera
rien !

```
int i=0;
do {
    System.out.print(i);
    i = i / 2 ;
} while (i > 1);
```

affichera
0

L'itération `for`

Les itérations permettent l'application itérative d'un traitement, contrôlée par une **initialisation**, une **condition d'arrêt**, et une opération de **mise à jour** de certaines variables.

Syntaxe générale :

```
for (initialisation; condition; mise_a_jour)
        Instructions
```

Note : Une boucle `for` est équivalente à la boucle `while` suivante :

```
{ initialisation;
        while (condition) {
                Instructions
        mise_a_jour;
        }
}
```

Même remarque ici que pour `if` et `while` : *`Instructions`* est soit une **instruction élémentaire**, soit un **bloc d'instruction**. Il est conseillé de toujours utiliser la syntaxe par bloc.

L'itération `for` : Exemple 1

Exemple simple : affichage des carrés des nombres entre 0 et 9

```
for (int i=0; i < 10; i++) {
        System.out.println(i*i);
}
```

Notez qu'ici la variable de contrôle de la boucle, `i`, est déclarée et initialisée dans le `for`.

L'itération `for` : Exemple 2

Remarque : si plusieurs instructions d'initialisation ou de mise à jour sont nécessaires, elles sont séparées par des **virgules**. Elles sont **exécutées de la gauche vers la droite**.
Exemple :

```
for (int i=0, s=0; i < 5; s += i, ++i) {
        System.out.println( i + ", " + s);
}
```

affichera

```
0, 0
1, 0
2, 1
3, 3
4, 6
```

Déroulement de l'exemple précédent

Le déroulement instruction par instruction de l'exemple précédent est :

```
int i=0;                                          i=0
int s=0;                                           s=0
i < 5 ? ☞ oui, donc on continue
System.out.println( i + ", " + s);                0, 0
s += i; (c'est-à-dire s = s + i;)                  s=0+0=0
++i; (c'est-à-dire i = i + 1;)                    i=1
i < 5 ? ☞ oui, donc on continue
System.out.println(i + ", " + s);                 1, 0
s += i;                                            s=0+1=1
++i;                                              i=2
i < 5 ? ☞ oui, donc on continue
System.out.println(i + ", " + s);                 2, 1
s += i;                                            s=1+2=3
++i;                                              i=3
i < 5 ? ☞ oui, donc on continue
```

```
                                                    i=3   s=3
System.out.println(i + ", " + s);                       3, 3
s += i;                                                 s=3+3=6
++i;                                                i=4
```

`i < 5` ? ☞ oui, donc on continue

```
System.out.println(i + ", " + s);                       4, 6
s += i;                                                 s=6+4=10
++i;                                                i=5
```

`i < 5` ? ☞ **NON**, donc on s'arrête

Sauts : `break` et `continue`

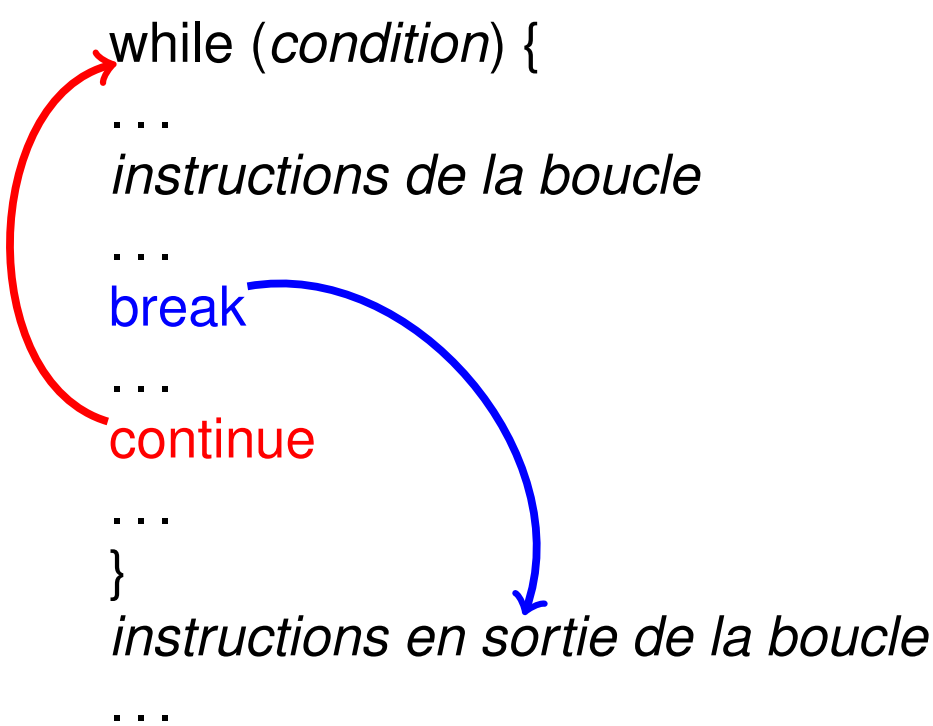
Java fournit deux instructions prédéfinies, `break` et `continue`, permettant de contrôler de façon plus fine le déroulement d'une boucle.

- Si l'instruction `break` est exécutée au sein du bloc intérieur de la boucle, l'exécution de la boucle est interrompue (quelque soit l'état de la condition de contrôle) ;
- Si l'instruction `continue` est exécutée au sein du bloc intérieur de la boucle, l'exécution du bloc est interrompue et la condition de continuation est évaluée pour déterminer si l'exécution de la boucle doit être poursuivie.

Note : il y a un assez large consensus sur le fait que l'utilisation du `break` et `continue` est à déconseiller en général.

> Pour la petite histoire, un bug lié à une mauvaise utilisation de `break;` a conduit à l'effondrement du réseau téléphonique longue distance d'AT&T, le 15 janvier 1990. Plus de 16'000 usagers ont perdu l'usage de leur téléphone pendant près de 9 heures. 70'000'000 d'appels ont été perdus.
>
> [P. Van der Linden, *Expert C Programming*, 1994.]

Instructions `break` et `continue`



Instruction `break` : exemple

Exemple d'utilisation de `break` :
une **mauvaise** (!) façon de simuler une boucle avec condition d'arrêt

```
while (true) {
        instruction1;
        ...
    if (condition arret)
       break;
}
autres instructions;
```

Question : quelle est la bonne façon d'écrire le code ci-dessus ?

Instruction `continue` : exemple

Exemple d'utilisation de `continue` :

```
i = 0;
while (i < 100) {
        ++i;
        if ((i % 2) == 0) continue;
    // la suite n'est executee que pour les
    // entiers impairs
        Instructions;
        ...
}
```

Question : quelle est une meilleure façon d'écrire le code ci-dessus ?

(on suppose que `Instructions; ...` ne modifie pas la valeur de `i`)

Portée : cas des boucles

La déclaration d'une variable **à l'intérieur d'une boucle** est une déclaration associée au bloc de la boucle.

Dans la structure d'itération suivante :

```
for (int i = 0; i < 10; i++) {
        System.out.println(i);
}
// on ne peut plus acceder a i ici
```

la variable `i` est **locale à la boucle**.

Les structures de contrôle

les branchements conditionnels : *si ... alors ...*

```
if (condition)
    instructions
.........................
if (condition 1)
    instructions 1
...
else if (condition N)
    instructions N
else
    instructions N+1
```

```
switch (expression) {
   case valeur:
       instructions;
       break;
   ...
   default:
       instructions;
}
```

les boucles conditionnelles : *tant que ...*

```
while (condition)
    instructions
```

```
do
    instructions
while (condition);
```

les itérations : *pour ... allant de ... à ...*

```
for (initialisation ; condition ; increment)
    instructions
```

les sauts : `break;` et `continue;`

Note : `instructions` représente une instruction élémentaire ou un bloc.
`instructions;` représente une suite d'instructions élémentaires.

Ce que j'ai appris aujourd'hui

- Comment exprimer des comparaisons et des tests logiques en Java (opérateurs relationnels et logiques)
- Comment faire exécuter certaines instructions **de façon répétitive** et/ou **en fonction de certaines conditions**
- A structurer certaines parties de mon code en **bloc**
- Comment écrire en Java les trois structures de contrôle de base :
  - Branchement conditionnels (`if`)
  - Boucles (`while`)
  - Itérations (`for`)
- Ce qu'est la notion de portée d'une variable.

☞ je peux maintenant écrire des programmes plus complexes, plus intéressants.

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 4 :
  - Tableaux : introduction [11 :31]
  - Tableaux : déclaration [11 :09]
  - Tableaux : traitements courants [11 :03]
  - Tableaux : affectation et comparaison [9 :03]
  - Tableaux à plusieurs dimensions [10 :33]
- Vidéos et quiz du MOOC semaine 5 :
  - Tableaux dynamiques [25 :50]
  - String : introduction [9 :48]
  - String : comparaisons [7.08]
  - String : traitements [15 :24]
- Le prochain cours :
  - de 14h15 à 15h (résumé et quelques approfondissements)

Représentation binaire d'un nombre

Nous représentons usuellement les nombres en base 10, selon un système de notation positionnelle.

Exemples :

- **123** (en base 10) vaut : $\mathbf{1} \times 10^2 + \mathbf{2} \times 10^1 + \mathbf{3} \times 10^0$
- **0.54** (en base 10) vaut : $\mathbf{0} \times 10^0 + \mathbf{5} \times 10^{-1} + \mathbf{4} \times 10^{-2}$

Le même principe de représentation est utilisable avec n'importe quelle autre base de numération.

Voici deux exemples de représentation binaire (en base 2) analogues aux précédents :

- `111010` en base 2 vaut : $\mathbf{1} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{0} \times 2^0$ (c'est-à-dire 58 en base 10)
- Le nombre `0.011` en base 2 vaut : $\mathbf{0} \times 2^0 + \mathbf{0} \times 2^{-1} + \mathbf{1} \times 2^{-2} + \mathbf{1} \times 2^{-3}$ (c'est à dire $\frac{1}{4} + \frac{1}{8} = 0.375$ en base 10)

Virgule flottante

Sur un ordinateur, l'ensemble $\mathbb{R}$ des réels est **approximé** par un ensemble **fini**, l'ensemble $\mathbb{F}$ des nombres représentables sur la machine.

Les nombres de $\mathbb{F}$ sont généralement définis selon la représentation dite « en **virgule flottante** » :

$$x = (-1)^s.m.\beta^{e+L-t} = (-1)^s \beta^{e+L} \sum_{i=0}^{t} a_i \beta^{-i}$$

où :

- $m = a_0 a_1 \ldots a_t$ est la **mantisse** (nombre entier de $t+1$ chiffres) avec $0 \leq a_i \leq \beta - 1$;
  (a_0 est le chiffre le plus significatif, a_t le moins significatif)
- $t+1$ est le nombre de chiffres significatifs ;
- e est l'**exposant** : un entier contenu dans un intervalle de valeurs admissibles $[0, U]$; $L < 0 < U$;
- β est la base (dans le cas où cette base vaut 2, ce qui est en informatique, chaque chiffre occupe 1 bit) ;
- s, valant 0 ou 1, est le **bit de signe**.

Virgule flottante (2)

Exemple (simpliste) de représentation en virgule flottante :
$\beta = 2$, $t = 5$, $U = 15$,$L = -7$ (4 bits)

Quel est le nombre représenté par `01010100101` ?

`01010100101` = `0 1010 100101`

c.-à-d. : $s = 0$, $e = \texttt{1010} = 10$, $m = \texttt{100101} = 32+4+1 = 37$
et donc :

$$x = (-1)^s.m.\beta^{e+L-t} = 37 \cdot 2^{10-7-5} = 37/4 = 9.25$$

On peut aussi le lire comme :

$$\underbrace{\texttt{1,00101}}_{m} \cdot 2^{10-7} = \left(1 + \frac{1}{8} + \frac{1}{32}\right) \cdot 8 = 8 + 1 + \frac{1}{4}$$

Autre exemple (simple) :

`10000000001` = `1 0000 000001`

c.-à-d. : $s = 1$, $e = \texttt{0000} = 0$, $m = \texttt{000001} = 1$, et donc :

$$x = (-1)^s.m.\beta^{e+L-t} = -1 \cdot 2^{-12} = -1/4096 = -0.000244140625$$

Virgules flottantes (3)

Les représentations en virgule flottante

$$\mathbb{F}(\beta, t, L, U) = \{x \in \mathbb{R} : x = (-1)^s \beta^{e+L} \sum_{i=0}^{t} a_i \beta^{-i}\} \text{ avec } \beta \geq 2$$

sont en général **normalisées** pour assurer l'*unicité* de la représentation :

- $a_0 \neq 0$ (sauf pour $e = 0$)
  Note : en binaire ($\beta = 2$), a_0 est donc forcément 1 ; on choisit alors de ne pas le représenter explicitement.
- le signe de zéro est fixé arbitrairement (généralement $s = 0$)

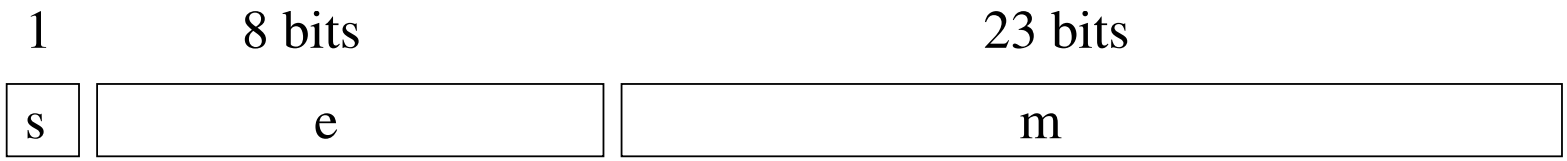
Pour éviter la prolifération de systèmes de calcul numériques différents, la représentation en virgule flottante fait l'objet de **normes** IEEE.
Dans cette norme, les exposant $e = 0$ et $e = U$ (tout à 1) ont des sens particuliers :

- $e = 0$: $m = 0$ ☞ $x = 0$, $m \neq 0$ ☞ m est alors interprété avec $a_0 = 0$ (au lieu de $a_0 = 1$) et e interprété comme 1 (i.e. $e + L = L + 1$, pas L) ;
- $e = U$: $m = 0$ ☞ $x = \infty$, $m \neq 0$ ☞ x est « not a number (NaN) »

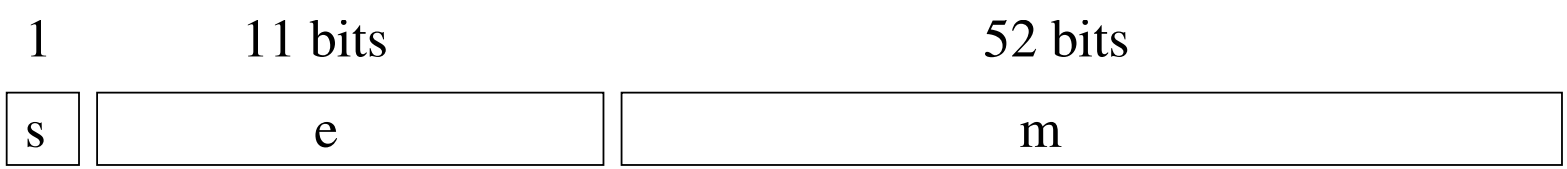
Virgule flottante (4)

Concrètement :

Simple précision (sur 32 bits) : $U = 255$, $L = -127$



Double précision (sur 64 bits) : $U = 2047$, $L = -1023$



Notes : un dépassement de capacité (overflow) se produit lorsqu'une opération sur des nombres en virgule flottante produit un nombre x non-représentable par un nombre en vigule flottante ($x \in]-\infty, -x_{\max}[\cup]x_{\max}, +\infty[$ où $x_{\max}$ est le plus grand nombre réel représentable).

Exemples norme IEC 559 simple précision

`0 00000000 00000000000000000000000`
☞ $x = 0$

`1 00000001 00000000000000000000001`
☞ $x = -2^{1-127} \cdot \left(1 + 2^{-23}\right) = 2^{-126} + 2^{-149} \simeq 1.18e-38$

`0 00000000 00000000000000000000001`
☞ $x = 2^{-126} \cdot 2^{-23} = 2^{-149} \simeq 1.40e-45$

`0 10000000 10000000000000000000000`
☞ $x = 2^{128-127} \cdot \left(1 + 2^{-1}\right) = 2 \cdot 1.5 = 3$

`0 01111111 00000000000000000000000`
☞ $x = 2^{127-127} \cdot 1 = 1$

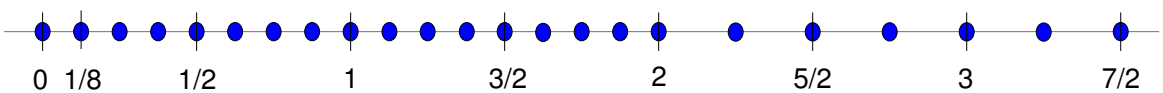
Exercice : `01000000010010010000111111011011`

Distribution non-uniforme des points en virgule flottante

Par construction, les nombres en virgule flottante ne sont **pas espacés de façon identique** dans l'ensemble des réels : plus l'on est proche du plus petit nombre représentable, plus ils sont denses.

(β = 2, t = 3, U = 3, L = -1)

	Exposant			
Mantisse	00	01	10	11
bit de signe → 0 000	0	1	2	∞
0 001	0.125	1.125	2.25	NaN
0 010	0.25	1.25	2.5	NaN
0 011	0.375	1.375	2.75	NaN
0 100	0.5	1.5	3	NaN
0 101	0.625	1.625	3.25	NaN
0 110	0.75	1.75	3.5	NaN
0 111	0.875	1.875	3.75	NaN



Virgule flottante et erreur d'arrondis

Lorsqu'un nombre x de $\mathbb{R}$ n'est pas dans $\mathbb{F}$ (i.e. non-représentable exactement), on manipule à sa place son plus proche arrondi $fl(x)$ (élément de $\mathbb{F}$).

La manipulation de tels nombres est donc forcément **entachée d'erreur**

Exemple :
Comment s'écrit 0.1 ($= 1/10$) en binaire ?

			erreur	écriture binaire
1/10	$\simeq$	1/16 + 1/32	1/160	`0.00011`
	$\simeq$	1/16 + 1/32 + 1/256	3/1280	`0.00011001`
	$\simeq$	1/16 + 1/32 + 1/256 + 1/512	1/2560	`0.000110011`

Question : est-ce que « ça s'arrête » ?
c'est-à-dire est-ce qu'il existe une représentation binaire finie de 1/10 ?

Virgule flottante et erreur d'arrondis (2)

Que vaut le nombre qui s'écrit en binaire $0.0001100110011001100110011\ldots$?

$$
\begin{aligned}
0.\overline{00011} &= 0.1 \times 0.\overline{0011} = \frac{1}{2} \times 0.\overline{0011} \\
&= \frac{1}{2} \sum_{i=1}^{\infty} \frac{1}{2^{4i-1}} + \frac{1}{2^{4i}} = \frac{1}{2} \sum_{i=1}^{\infty} (2+1) \times \frac{1}{2^{4i}} \\
&= \frac{3}{2} \times \left(\frac{1}{1 - \frac{1}{2^4}} - 1 \right) = \frac{3}{2} \times \frac{1}{15} = \frac{1}{10}
\end{aligned}
$$

et donc le nombre binaire $0.\overline{00011}$ vaut (en décimal) $1/10$

Conclusion : toute représentation binaire (de longueur finie) de $1/10$ comportera une erreur

Note : ce n'est pas spécifique au binaire : comment s'écrit $1/3$ en décimal ?

Virgule flottante et erreur d'arrondis (3)

Les erreurs absolue et relative causées en substituant $fl(x)$ à x sont toutefois aisément quantifiables :

erreur relative : $E_{rel}(x) = \frac{|x - fl(x)|}{|x|} \leq 0.5\, \beta^{1-t}$

erreur absolue : $E_x = | x - fl(x) | \leq 0.5\, \beta^{e-t}$

Un problème important est cependant que dans la plupart des méthodes numériques, chaque opération élémentaire et/ou chaque itération (si c'est une méthode itérative) est susceptible d'être entachée d'erreur

- ☞ il y a donc un **effet cumulatif** des erreurs à prendre en compte et qui est dépendant de l'algorithme de calcul utilisé