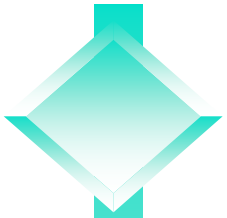
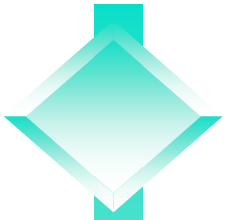


UNIVERSITÉ PARIS SORBONNE PARIS IV

Cours n° 3

Traitements en Java



Sommaire

1. Structures de contrôle

- Branchement conditionnel
- Sélection à choix multiples
- Boucles conditionnelles
- Itérations
- Ruptures de séquence

2. Fonctions

- Définition de fonction
- Paramètres d'entrée et de sortie
- Appel de fonction
- Passage de paramètres
- Structuration du code

Rôle des structures de contrôle

Une structure de contrôle permet de modifier l'ordre séquentiel d'exécution des instructions d'un programme (flux d'exécution)

Faire exécuter des instructions

- en fonction de certaines **conditions**
- de façon **répétitive**

Principales structures de contrôle

- **Branchement conditionnel**
  si … alors … sinon … finSi
- **Sélection à choix multiples**
  suivant … cas … faire … finFaire
- **Boucle (conditionnelle)**
  faire … finFaire tant que… ;
  tant que … faire … finFaire
- **Itération**
  pour … allant de … à … faire … finFaire

En langage pseudo-naturel : faire … finFaire correspond à la notion de blocs d'instructions (**{** et **}**)

Branchement conditionnel (1/3)

Permet d'exécuter des traitements selon certaines conditions (alternatives de traitements)

if (<condition>**)** <blocV> [**else** <blocF>]

condition : expression (logique)
blocV : bloc d'instructions (alternative de condition vérifiée)
blocF : bloc d'instructions (alternative de condition non vérifiée)

Remarque :

L'alternative **else** est optionnelle

Rappel :

Un **bloc** est une séquence d'instructions délimitée par **{** et **}**
Le bloc peut être vide ou restreint à une seule instruction (simple ou composée) et dans ce cas les délimiteurs peuvent être omis

Branchement conditionnel (2/3)

Contrôle d'exécution

```
   inst1;
   if (condition) {
      instV1;
      instV2;
   }
   else {
      instF1;
      instF2;
      instF3;
   }
α  inst3;
```

Flux d'exécution

inst1;
si condition est évaluée à ***true***
    instV1; instV2;
    // continuer en séquence en α
     inst3;

sinon // **condition** est évaluée à ***false***
    instF1; instF2; instF3;
    // continuer en séquence en α
    inst3;

Règle d'indentation à respecter

Les **instructions** constitutives d'un bloc doivent être **indentées**, c'est-à-dire mises en retrait par **tabulation** (au moins 3 caractères) relativement à la structure de bloc (**{** et **}**) pour mettre en valeur le **séquencement** des **instructions** à l'exécution

Branchement conditionnel (3/3)

Branchement sans alternative // Afficher la valeur absolue d'un entier x

```
int x=-21, valAbsolue=x;
if (x < 0)
   valAbsolue=-x;
System.out.println("valeur absolue de " + x + " : "
                   + valAbsolue );
```

```
valeur absolue de -21 : 21
```

Branchement avec alternative // Que fait ce programme

```
// importer le paquetage système
// import java.lang.System;

char c='s';
if (c=='S'||c=='s') {
   System.out.println("Stop, j'arrête : je sors du programme!");
   System.exit(0);
}
else {
   System.out.println("Je continue en séquence...");
   System.out.println("tranquillement.");
}
```

```
Stop, j'arrête : je sors
du programme!
```

Sélection à choix multiples (1/3)

Traitements à effectuer pour certaines valeurs (discrètes) d'une expression (de type entier ou caractère)

switch (<expression>**) { {case** <valeur> **:** <instructions> **;}**
[**default :** <instructions>**;**] **}**

expression : le discriminant de type entier ou caractère
valeur : une valeur ou une constante du type de **expression**

Conseil

Lorsque le type de l'expression le permet,
préférer le **switch** à l'enchaînement de branchements conditionnels dès que le **niveau d'imbrication** dépasse **2**

Sélection à choix multiples (2/3)

Contrôle d'exécution

```
inst1;
switch (expression) {
   case V1 : instV1_1;
             instV1_2;
             break;
   case V2 : instV2_1;
   case V3:  instV3_1;
   case V4:
   case V5:  instV5_1;
             break;
   default:  instDefaut;
}
inst3;
```

Flux d'exécution

inst1;
Si expression est évaluée à V1
instV1_1; instV1_2; inst3;
Si expression est évaluée à V2
instV2_1; instV3_1; instV5_1; inst3;
Si expression est évaluée à V3
instV3_1; instV5_1; inst3;
Si expression est évaluée à V4
instV5_1; inst3;
Si expression est évaluée à V5
instV5_1; inst3;
Si expression est évaluée à toute autre valeur que V1, V2, V3, V4 ou V5
instDefaut; inst3;

Remarque :

L'instruction **break** interrompt l'exécution en séquence et provoque la **sortie** du bloc ***switch***

Sélection à choix multiples (3/3)

Exemple :

```
char c='h';
switch (c) {
   case 'o':
   case 'O':  System.out.println("oui");
              break;
   case 'n':
   case 'N':  System.out.println("non");
              break;
   default :System.out.println("peut-être");
}
```

c vaut ‘o’ ou ‘O ‘
oui

c vaut ’n’ ou ‘N’
non

C vaut ‘h’ (ou tout autre caractère que ‘o’, ‘O’, ‘n’, ‘N’)
peut-être

Rôles des boucles conditionnelles

Permet, avec un contrôle *a posteriori* ou *a priori et* par une condition de continuation, de répéter un traitement

Boucle *a posteriori* : do … while (…);

do <*blocB*> **while (**<*condition*>**);**

blocB : bloc d'instructions de la boucle (instructions à répéter)
condition : expression (logique) de continuation de boucle

Boucle a priori : do … while (…);

while (<*condition*>**)** <*blocB*>**;**

condition : expression (logique) de continuation de boucle
blocB : bloc d'instructions de la boucle (instructions à répéter)

Boucle conditionnelle - *a posteriori* : do … while (…); (1/2)

Contrôle d'exécution

```
inst1;
do {
   instB1;
   instB2;
} while (condition);
inst3;
```

Flux d'exécution

inst1;
1. instB1; instB2; // fin de bloc
si (**condition** est évaluée à ***true***)
    aller en **1.**
sinon inst3;

Remarque :

Une **instruction** (au moins) de la boucle doit avoir un **effet de bord** sur la **condition**

Boucle conditionnelle - *a posteriori* : do … while (…); (2/2)

Exemple :

```
// Tirer à pile ou face jusqu'à obtenir face,
// afficher le résultat de chaque tirage
// et le nombre total de jets
// on simule face (resp. pile) par l'entrée
// d'un positif (resp. négatif)

int nbJets=0, entier;
do {
    entier=Keyboard.getInt("Entier ? ");
    if (entier<0)
       System.out.println("pile");
    else
       System.out.println("face");
    nbJets++;
} while (entier<0);
System.out.println("Face,gagné en "+ nbJets+" coup(s)");
```

```
Entier ? -1
pile
Entier ? -20
pile
Entier ? 0
face
Face,gagné en 3 coup(s)
```

Boucle conditionnelle - *a priori* : while (…) …; (1/2)

Contrôle d'exécution

```
    inst1;
     while (condition) {
α      instB1;
       instB2;
    }
    inst3;
```

Flux d'exécution

inst1;
1. **si** (**condition** est évaluée à ***true***)
   // continuer en séquence en α
   instB1; instB2; // fin de bloc
   aller en **1.**
   sinon inst3;

Remarque :

Une **instruction** (au moins) de la boucle doit avoir un **effet de bord** sur la **condition**

Boucle conditionnelle - *a priori* : while (…) …; (2/2)

Exemple :

// Calculer la moyenne des entiers positifs
// saisis au clavier jusqu'à la saisie
// d'un négatif

```
Entier ? 1
Entier ? 2
Entier ? -1
Moyenne des entiers
positifs : 1.5
```

```
int nbEntiers=0, cumul=0, entier;
while ((entier=Keyboard.getInt("Entier ? "))>=0) {
    nbEntiers++;
    cumul+=entier;
}
if (nbEntiers==0)
    System.out.println("Pas d'entrée d'entier positif");
else
    System.out.println("Moyenne des entiers positifs : "
                       +(float)cumul/nbEntiers);
```

Rôle de l'itération

Permet un traitement itératif contrôlée par un mécanisme de variables d'itération et de critère d'arrêt

for (*<initialisation_s>***;** *<condition>***;** *<mise_s_à _jour>***)** <blocB>

initialisation_s : une ou plusieurs opérations d'initialisation des variables d'itération
condition : condition d'arrêt d'itération
mise_s_à _jour : une ou plusieurs opérations de mise à jour des variables d'itération

Remarque :

Structure de contrôle **très adaptée** lorsque le **nombre d'itérations** est **connu**

Equivalence avec la boucle while

```
{
   <initialisation_s>;
   while (<condition>) {
       …;  //instructions de blocB
       <mise_s_à _jour>)
   }
}
```

Itération - for (…; …; …) …; (1/2)

Contrôle d'exécution

```
    inst1;
    for (instI1,instI2; condition; instMAJ1, instMAJ2) {
       instB1;
       instB2;
    }
β   inst3;
```

Flux d'exécution

inst1;
instI1, instI2;
1. **si** (**condition** est évaluée à ***true***)
   // continuer en séquence en β
   inst3;
   sinon
   instB1; instB2; // fin de bloc
   instMAJ1, instMAJ2;
   aller en **1.**

Remarque 1 :

Une **instruction** (au moins) de la boucle doit avoir un **effet de bord** sur la **condition**

Remarque 2 :

Les instructions **instI1** et **instI2** sont **locales au bloc** de l'itération *for*
Les **règles de visibilité s'appliquent** aux variables impliquées dans ces instructions : elles ne sont **pas référençables** à **l'extérieur du bloc**

Itération - for (...; ...; ...) ...; (2/2)

Exemple :

// Calculer la moyenne des 30 premiers
// entiers positifs

```
Moyenne des 30
premiers entiers
positifs : 14.5
```

```
    int borne=30;
β   int cumul=0;
    for (int i=0; i<borne; i++) {
       cumul+=i;
    }
    System.out.println("Moyenne des "+borne+" premiers entiers
                       positifs : "+(float)cumul/borne);
```

Question – Sans l'instruction β, aurait-on pu écrire l'itération ***for*** ?

```
for (int i=0, cumul=0; i<borne; i++)
```

Ruptures de séquence

Instructions de rupture de séquence : break et continue

break

- Exclusivement présente dans une **boucle** ou une **clause de sélection** (*case*)
- Interrompt le flux d'exécution dans la boucle (la plus interne) ou l'alternative de sélection en provoquant un **saut** vers l'**instruction suivant la structure de contrôle**

continue

- Exclusivement présente dans une **boucle**
- Interrompt le flux d'exécution des instructions du bloc en provoquant un **transfert d'exécution** à l'**itération suivante** de la boucle
  Dans le cas d'une boucle **for** : mise à jour des variables d'itération et ré-évalution de la condition d'arrêt

Conseil

Utiliser, break et continue, avec **discernement**

Notion de fonction

Portion de code réutilisable caractérisée par :

- Un **identificateur** : le nom de la fonction
- Des **arguments/paramètres formels** (les « ***entrées*** ») : **interface** entre le code externe (à la fonction) et le corps (code interne) de la fonction
- Un **corps** : le bloc d'instructions de traitement de la fonction, manipulant ses propres variables locales et soumis aux règles de portée/visibilité
- Une **valeur de retour** (la « ***sortie*** ») : spécifiée dans l'instruction *return* dans le corps de la fonction

Exemple : fonction **mini** calculant le plus petit de deux réels

Corps de la fonction
Relations entre la sortie et les entrées
Traitement/calcul de la sortie en fonction des entrées

Exemple de définition d'une fonction en Java

```
/**
 * Minimum de deux réels
 * @param x : premier argument réel
 * @param y : deuxième argument réel
 * @return : le minimum de x et y
 */


static double mini(double x, double y){
  double leMin;
  if (x<y) leMin=x;
  else leMin=y;
  return (leMin);
}
```

Commentaire en Javadoc

Notion de paramètres d'entrée, de sortie, d'entrée/sortie

Paramètres d'entrée

- **Donnée** dont la valeur est **nécessaire** au **traitement**
- Cette valeur n'est **pas modifiée** par le traitement

Paramètres de sortie

- **Donnée** dans laquelle est stockée un **résultat du calcul**
- Sa **valeur initiale** n'est **pas utilisée** pour le traitement

Paramètres d'entrée/sortie

- **Donnée** dont la valeur est **nécessaire au calcul**
- **Donnée** dans laquelle est stockée un **résultat du calcul**

Fonction en mathématiques vs fonction en informatique

La **fonction en mathématiques** admet des **paramètres d'entrée** et une **valeur de retour** (modèle idéal pour les tests)

La **fonction en informatique** a un sens plus large : les **paramètres** (liste *éventuellement vide*) peuvent être des **3 types** précédemment **cités** et la **valeur de retour** est *éventuelle*

Définition de fonction

Corps de la fonction : son bloc instructions

```
<type_retour>  <id_fonction> ({<type> <id_arg>[,]})
<bloc_instructions>
```

type_retour : type de la valeur renvoyée par la fonction
id_fonction : nom de la fonction
type : type de l'argument
id_arg : identificateur de l'argument
bloc_instructions : corps de la fonction

Remarque :

- Les **arguments/paramètres** de la fonction sont des **variables locales** au **bloc** d'instructions de cette fonction
- La **valeur retournée** est indiquée par l'instruction :
  `return (<expression>);` expression pouvant être réduite à une variable ou à une valeur littérale

Exécution et gestion mémoire

Au lancement d'un programme

- Le code est chargé (*code segment*)
- L'espace mémoire nécessaire pour stocker les variables globales est alloué (*data segment*)
- Les variables globales qui le nécessitent sont initialisées
- La fonction main est appelée

A chaque appel de fonction

- L'espace mémoire nécessaire pour stocker les paramètres formels et les variables locales est alloué (*stack segment*)
- Les paramètres formels sont initialisés par recopie des valeurs des paramètres effectifs
- Les variables locales qui le nécessitent sont initialisées

Au retour de la fonction

- L'espace mémoire alloué à l'appel est désalloué

Appel de fonction

Appel de la fonction avec des paramètres effectifs

Soit la fonction `double mini(double x, double y){ }`
L'appel de la fonction se fait dans un bloc dit « appelant »
`mini(e1,e2);` // e1 et e2 sont les paramètres effectifs

Evaluation de l'appel

1. Les **paramètres effectifs** e1 et e2 sont **évalués** et les valeurs sont **affectées** aux **paramètres formels** x et y de la fonction : **x=e1**, **y=e2**
2. Le **programme** correspondant au corps de la fonction est **exécuté**
3. **L'expression de retour** (spécifiée par le *return*) est **évaluée** et **retournée** comme résultat de l'appel
4. Toutes les **variables locales** sont **désallouées**

Exemple d'appel :

```
double a=20, b=10;
double lePlusPetit=mini(2*a,5*b);
System.out.println("Min("+a+", "+b+") : " + lePlusPetit);
System.out.println("Minimum (2*a, 5*b) : " + mini(2*a, 5*b));
```

Passage de paramètre par valeur (1/2)

Paramètre formel : `<type_parametre> x`

- **x** est une **variable locale** du **corps** de la fonction
- **x est** une **copie** du **paramètre effectif**
  La **valeur** du **paramètre effectif** est **copié** dans **x**
- Le **paramètre effectif ne peut être modifié**
- Le **paramètre effectif** passé par valeur peut être une **constante**, une **variable** ou une **expression**
- **Le paramètre effectif** est un **paramètre d'entrée**

Pas d'effet de bord :

Les modifications du paramètre formel effectuées à l'intérieur de la fonction ne sont pas répercutées (sur le paramètre effectif) à l'extérieur de la fonction

En Java : Seuls les paramètres de type primitif sont passés par valeur

Passage de paramètre par valeur (2/2)

Exemple :

```
public class PassageValeur {
   static void f(int x){
      int y=10;
      x*=y;
      System.out.println("x="+x);
   }
   public static void main(String[] args) {
      int valeur=1;
      f(valeur);
      System.out.println("valeur="+valeur);
   }
}
```

10
20
30

10
20
30

```
x=30
valeur=3
```

Passage de paramètre par référence

En Java les variables de type autre que les types primitifs sont des variables références sur les objets implantés en mémoire

Paramètre formel : `<type_parametre> x`

Où type_parametre est un type autre que primitif
Entre autres, les types utilisateur (classes non standards)

- **x** est une **référence**, **variable locale** du **corps** de la fonction
- **x** est un **synonyme/alias** du **paramètre effectif**
  Le paramètre effectif est copié dans la référence locale x
- Le **paramètre effectif** est **modifié si x** est **modifié**
- Le **paramètre effectif** doit être une **variable**
- **Le paramètre effectif** est un **paramètre d'entrée/sortie**

Effet de bord :

Modifier le paramètre formel dans le corps de la fonction revient à modifier le paramètre effectif : les modifications sont répercutées à l'extérieur de la fonction

Variables et références

Une variable en Java est définie par un type et peut contenir

- **une valeur de type primitif** (boolean, char, byte, short, int, long, double)
- **une référence vers un objet quelconque** (instance d'une classe standard ou non)

Une référence à un objet est une copie de l'adresse (mémoire) où est mémorisé cet objet

Déclaration

```
int x;      // Elabore une variable x qui peut contenir des entiers
            // (réservation mémoire)
```

Soit une classe Personne

`Personne p;` // **Déclare** une variable référence p qui désignera des instances de la classe Personne (réservation mémoire de p : une adresse)

Instanciation et affectation permettent d'associer une valeur à une variable

```
x=10;       // Affectation directe (sans instanciation pour un type primitif)
p=new Personne(); // Instanciation d'un nouvel objet de type Personne
//(réservation mémoire le l'objet par new) et on affecte à la variable p l'adresse
//mémoire du nouvel objet (adresse de l'objet copié dans la référence p)
```

Polymorphisme – Surcharge des fonctions

Plusieurs fonctions de même nom peuvent être définies si ces fonctions n'ont pas les mêmes listes de paramètres
(nombre de paramètres distincts ou types de paramètres différents)

Exemple :

```
static void affiche(int x) {
  System.out.println("Entier : "+x);
}
static void affiche(double x) {
  System.out.println("Réel : "+x);
}
static void affiche(int x1, int x2) {
   System.out.println("Couple : ("+x1+","+x2+")");
}

// Appel des fonctions
affiche(18); affiche(6.57); affiche(4,1);
```

```
entier : 18
reel : 6.57
couple : (4,1)
```

Fonctions dans le fichier du point d'entrée

Structuration du code – Solution 1

```
package sem03;
public class Minimum {
   public static void main(String[] args) {
      double a=20., b=10.;
      double minimum=mini(a,b);
      System.out.println("Minimum de " + a + " et " + b + " : "
                         + minimum);
      System.out.println("Minimum de 2 et 3 : " + mini(2,3));
   }
   /**
    * Minimum de deux réels
    * @param x : premier argument réel
    * @param y : deuxième argument réel
    * @return : le minimum de x et y
    */
   static double mini(double x, double y){
      double leMin;
      if (x<y) leMin=x;
      else leMin=y;
      return (leMin);
   }
}
```

Structuration du code – Solution 2 (1/2)

Développement d'un composant LibMath (librairie mathématique)
Une classe (un fichier) : LibMath.java

```
package sem03;
/**
 * Librairie mathématique. Des fonctions usuelles…
 */
public class LibMath {
   /**
    * Minimum de deux réels
    * @param x : premier argument réel
    * @param y : deuxième argument réel
    * @return : le minimum de x et y
   */
   static double mini(double x, double y){
      double leMin;
      if (x<y) leMin=x;
      else leMin=y;
      return (leMin);
   }
}
```

Structuration du code – Solution 2 (2/2)

Utilisation du composant LibMath : l'applicatif
Une classe point d'entrée (un fichier) : TestMin.java

```
package cours03;
/**
 * Applicatif de test de la fonction min de LibMath…
 */
public class TestMin {

   public static void main(String[] args) {
      double a=20, b=10;
      double minimum=LibMath.mini(a,b);
      System.out.println("Minimum de " + a + " et " + b + " : "
                         + minimum);
      System.out.println("Minimum de 9 et 9 : « +
                          LibMath.mini(9,9));
   }
}
```