

# Programmation Orientée Objet

## **3ème Partie**

Gérald Oster  
<oster@loria.fr>

# Plan du cours

- Introduction
- Programmation orientée objet :
  - Classes, objets, encapsulation, composition
  - 1. Utilisation
  - 2. Définition
- Héritage et polymorphisme :
  - Interface, classe abstraite, liaison dynamique
- Généricité
- (Collections)

# 4<sup>ème</sup> Partie : Tableaux à taille fixe et Tableaux dynamiques

## Objectifs de cette partie

---

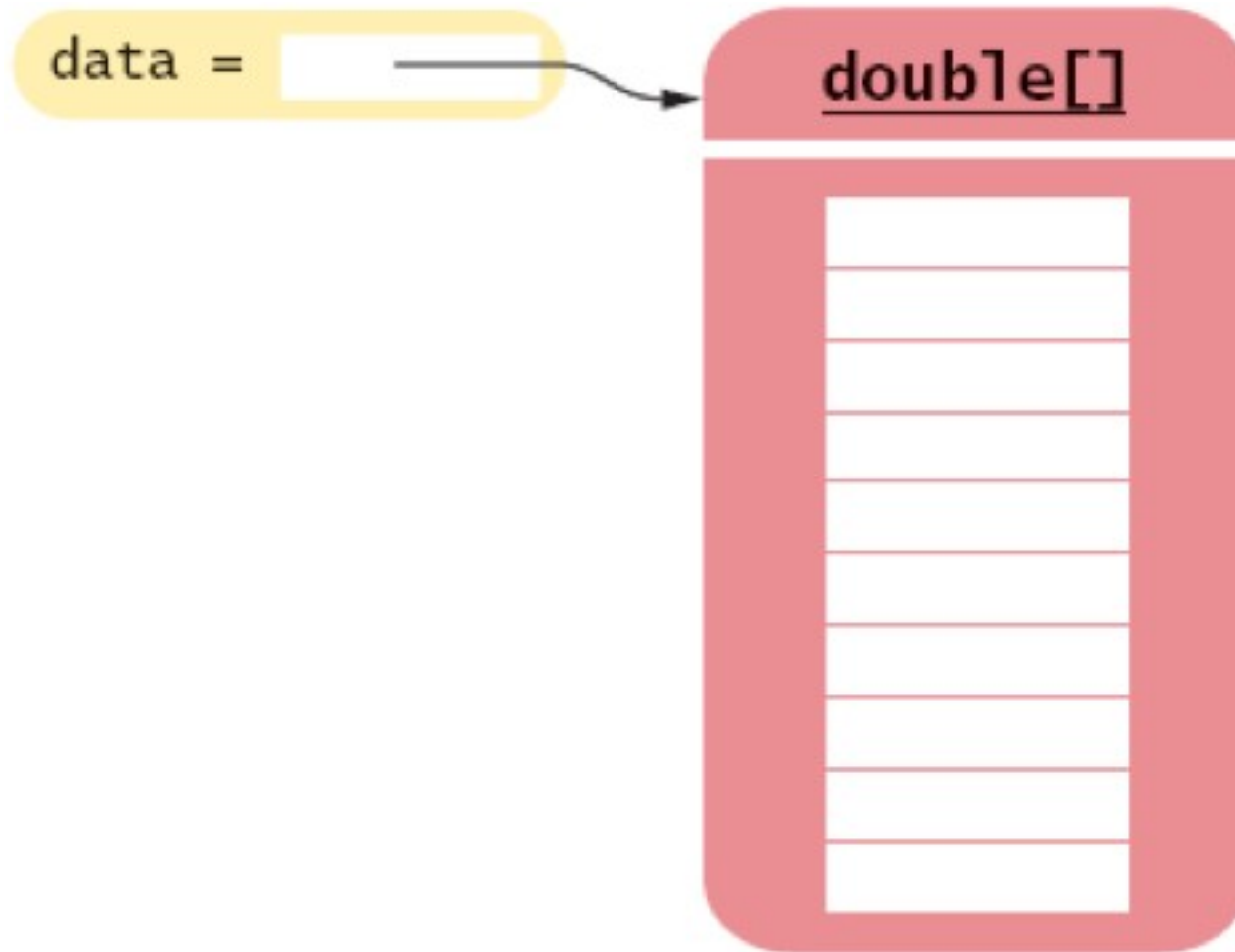
- Comprendre la différence entre tableaux à taille fixe et tableaux dynamiques
- Découvrir les classes enveloppes (*wrapper*), l'auto-boxing, et la généralisation des boucles
- Etudier les algorithmes classiques sur les tableaux
- Découvrir les tableaux à 2 dimensions
- Savoir quand utiliser des tableaux à taille fixe ou des tableaux dynamiques dans vos programmes
- Implémenter des tableaux partiellement remplis
- Comprendre le concept de test de regression

# Tableaux

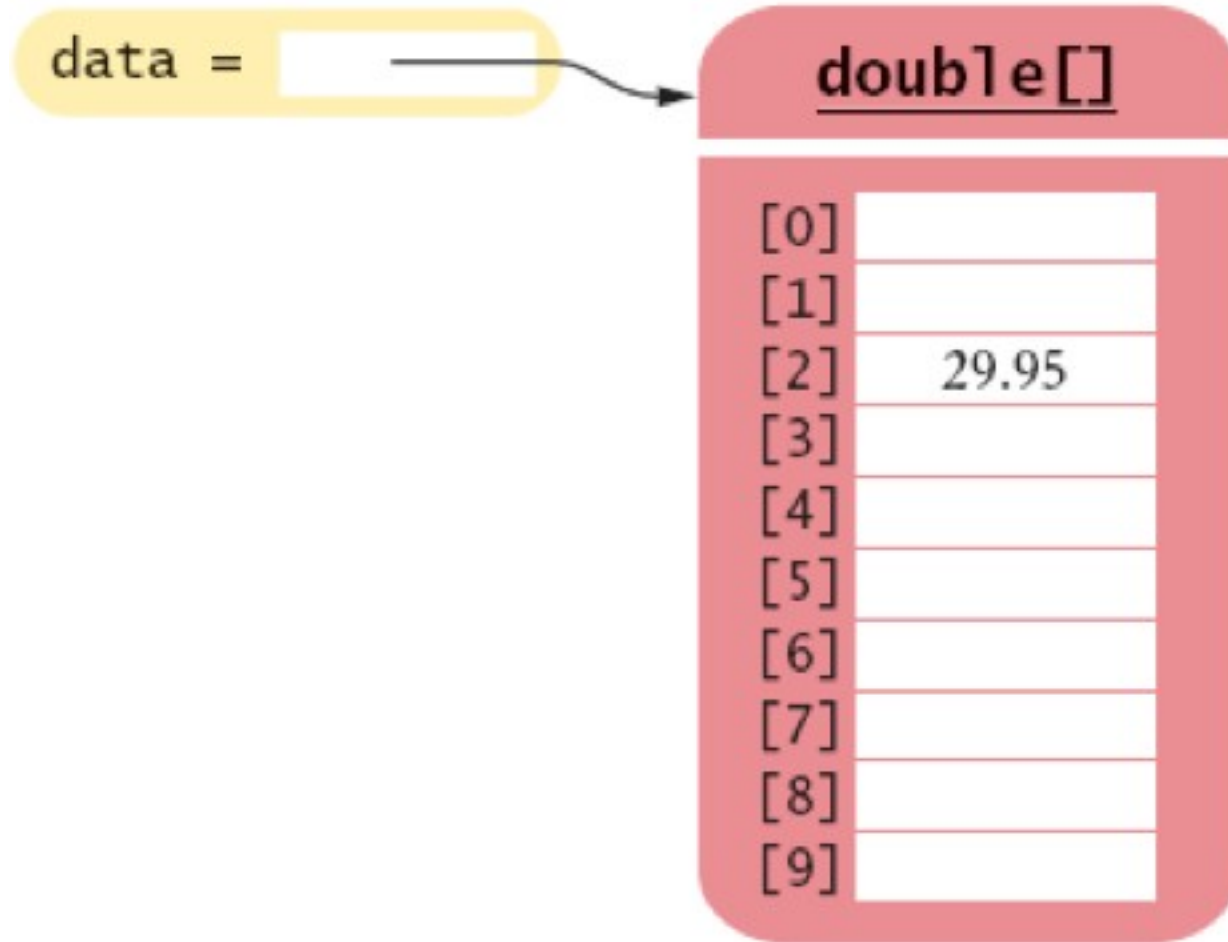
---

- Tableau : Séquence de valeur du même type
- Construction d'un tableau :  
`new double[10]`
- Stocké dans une variable de type `double[]`  
`double[] data = new double[10];`
- Quand un tableau est créé, toutes ses valeurs sont initialisées à une valeur par défaut :
  - Nombre : `0`
  - Booléen : `false`
  - Référence d'objet : `null`

# Tableaux - Référence



# Tableaux – Affectation d'une valeur



# Tableaux

---

- Utilisation d'une valeur stockée :  

```
System.out.println("The value of this data item is "  
    + data[4]);
```
- Taille d'un tableau : `data.length` (C'est n'est pas une methode!)
- Intervalle des indices : de 0 à `length - 1`
- Accès à un élément non existant déclenche une erreur de dépassement des bornes

```
double[] data = new double[10];  
data[10] = 29.95; // ERROR
```

- Limitation : Les tableaux ont une taille fixe



# Syntaxe Construction d'un tableau

---

`new typeName[length]`

**Exemple :**

`new double[10]`

**Objectif :**

Construire un nouveau tableau dont le nombre d'élément est fixé.

## Syntaxe Accès à un élément d'un tableau

---

*arrayReference*[*index*]

**Exemple :**

data[2]

**Objectif :**

Accéder à un élément d'un tableau par l'intermédiaire de son indice.

## Questions

---

Quels sont les éléments contenus dans le tableau après l'exécution de cette séquence d'instructions ?

```
double[] data = new double[10];  
for (int i = 0; i < data.length; i++) data[i] = i * i;
```

**Réponse :** 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, mais pas 100

## Questions /2

---

Qu'affiche la séquence d'instructions suivantes? Ou quelle est l'erreur ? Quand cette erreur est-elle détectée (compilation ou exécution)?

a) `double[] a = new double[10];`  
`System.out.println(a[0]);`

b) `double[] b = new double[10];`  
`System.out.println(b[10]);`

c) `double[] c;`  
`System.out.println(c[0]);`

### Réponses :

a) 0

b) erreur à l'exécution : dépassement des bornes du tableau

c) erreur à la compilation : c n'est pas initialisée

# Tableaux dynamiques : ArrayList

---

- La classe `ArrayList` gère une séquence d'objets
- Peut grandir et diminuer si nécessaire
- La classe `ArrayList` fournit les méthodes pour les tâches communes (insertion, suppression, ...)
- La classe `ArrayList` est une classe *générique* :
- `ArrayList<T>` contient des objets de type `T` :

```
ArrayList<BankAccount> accounts = new
    ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```
- La méthode `size` retourne le nombre d'éléments stockés

# Accès aux éléments d'un tableau dynamique ArrayList

- Utiliser la méthode `get`
- Les indices débutent à 0
- ```
BankAccount anAccount = accounts.get(2);  
// récupère le 3ème élément du tableau
```
- Erreur de dépassement des bornes levée si hors des bornes actuelles
- Erreur de dépassement la plus courante :

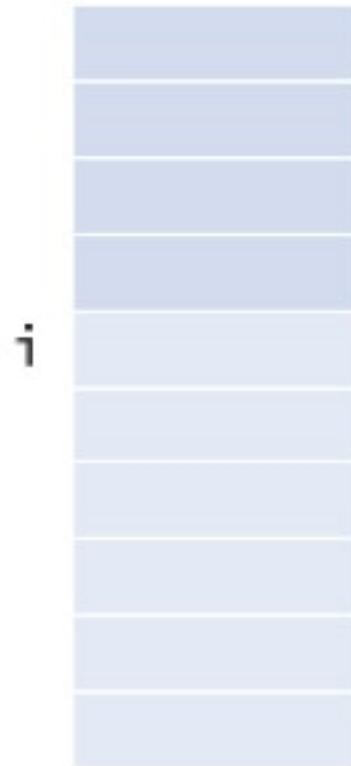
```
int i = accounts.size();  
anAccount = accounts.get(i); // Error  
//legal index values are 0. . .i-1
```

## Ajout d'éléments dans un tableau dynamique ArrayList

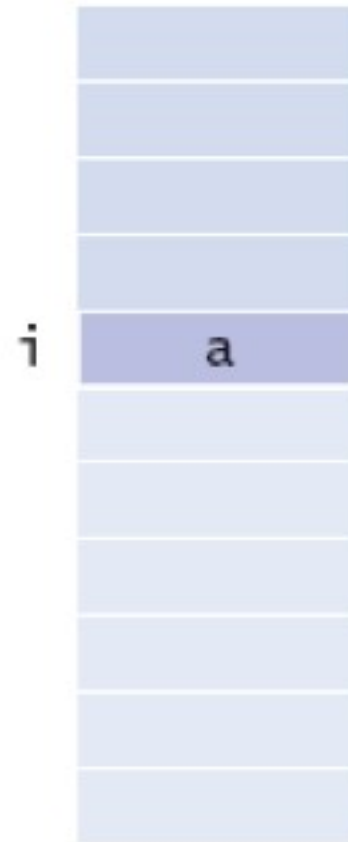
---

- set écrase la valeur existante  
`BankAccount anAccount = new BankAccount(1729);`  
`accounts.set(2, anAccount);`
- add insert une nouvelle valeur à l'indice fourni  
`accounts.add(i, a)`

# Ajout d'éléments dans un tableau dynamique ArrayList /2



Before



After



## Animation 7.1 –

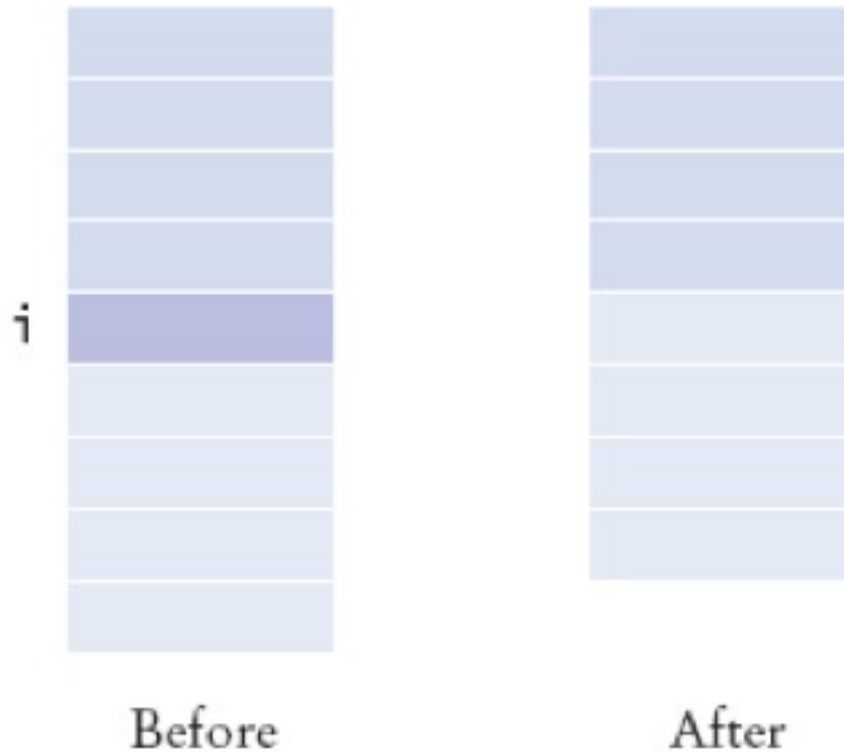
|     |  |
|-----|--|
| [0] |  |
| [1] |  |
| [2] |  |
| [3] |  |
| [4] |  |
| [5] |  |
| [6] |  |

This animation demonstrates inserting an element into an array list.

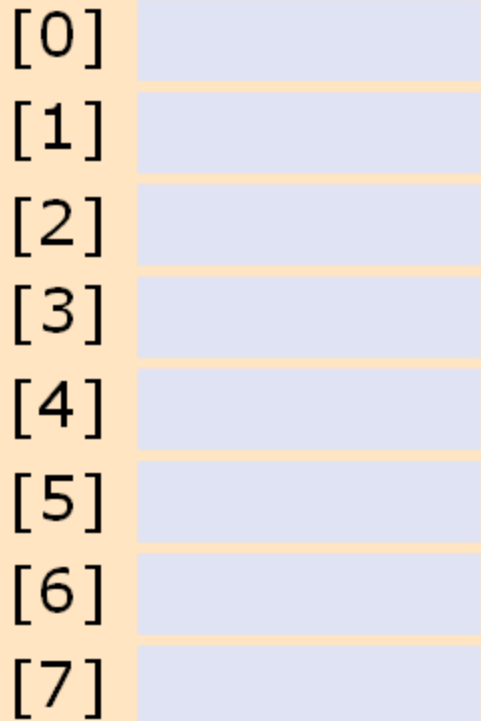


# Suppression d'éléments dans un tableau dynamique

Remove retire l'élément situé à un indice  
`accounts.remove(i)`



## Animation 7.2 –



This animation demonstrates removing an element from an array list.



## ch07/arraylist/ArrayListTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     This program tests the ArrayList class.
05: */
06: public class ArrayListTester
07: {
08:     public static void main(String[] args)
09:     {
10:         ArrayList<BankAccount> accounts
11:             = new ArrayList<BankAccount>();
12:         accounts.add(new BankAccount(1001));
13:         accounts.add(new BankAccount(1015));
14:         accounts.add(new BankAccount(1729));
15:         accounts.add(1, new BankAccount(1008));
16:         accounts.remove(0);
17:
18:         System.out.println("Size: " + accounts.size());
19:         System.out.println("Expected: 3");
20:         BankAccount first = accounts.get(0);
```

## ch07/arraylist/ArrayListTester.java /2

```
21:         System.out.println("First account number: "  
22:             + first.getAccountNumber());  
23:         System.out.println("Expected: 1015");  
24:         BankAccount last = accounts.get(accounts.size() - 1);  
25:         System.out.println("Last account number: "  
26:             + last.getAccountNumber());  
27:         System.out.println("Expected: 1729");  
28:     }  
29: }
```

## ch07/arraylist/BankAccount.java

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance
09:         @param anAccountNumber the account number for this account
10:     */
11:     public BankAccount(int anAccountNumber)
12:     {
13:         accountNumber = anAccountNumber;
14:         balance = 0;
15:     }
16:
17:     /**
18:         Constructs a bank account with a given balance
19:         @param anAccountNumber the account number for this account
20:         @param initialBalance the initial balance
21:     */
```

## ch07/arraylist/BankAccount.java /2

```
22:     public BankAccount(int anAccountNumber, double initialBalance)
23:     {
24:         accountNumber = anAccountNumber;
25:         balance = initialBalance;
26:     }
27:
28:     /**
29:      Gets the account number of this bank account.
30:      @return the account number
31:     */
32:     public int getAccountNumber()
33:     {
34:         return accountNumber;
35:     }
36:
37:     /**
38:      Deposits money into the bank account.
39:      @param amount the amount to deposit
40:     */
41:     public void deposit(double amount)
42:     {
43:         double newBalance = balance + amount;
44:         balance = newBalance;
45:     }
```

## ch07/arraylist/BankAccount.java /3

```
46:
47:     /**
48:         Withdraws money from the bank account.
49:         @param amount the amount to withdraw
50:     */
51:     public void withdraw(double amount)
52:     {
53:         double newBalance = balance - amount;
54:         balance = newBalance;
55:     }
56:
57:     /**
58:         Gets the current balance of the bank account.
59:         @return the current balance
60:     */
61:     public double getBalance()
62:     {
63:         return balance;
64:     }
65:
66:     private int accountNumber;
67:     private double balance;
68: }
```



### **Output:**

Size: 3

Expected: 3

First account number: 1008

Expected: 1008

Last account number: 1729

Expected: 1729

# Questions

---

Comment construit-on un tableau de 10 chaînes de caractères ?

**Réponse :**

```
new String[10];  
new ArrayList<String>();
```

## Questions /2

---

Quel est le contenu de `names` après l'exécution suivante ?

```
ArrayList<String> names = new ArrayList<String>();
```

```
names.add("A");
```

```
names.add(0, "B");
```

```
names.add("C");
```

```
names.remove(1);
```

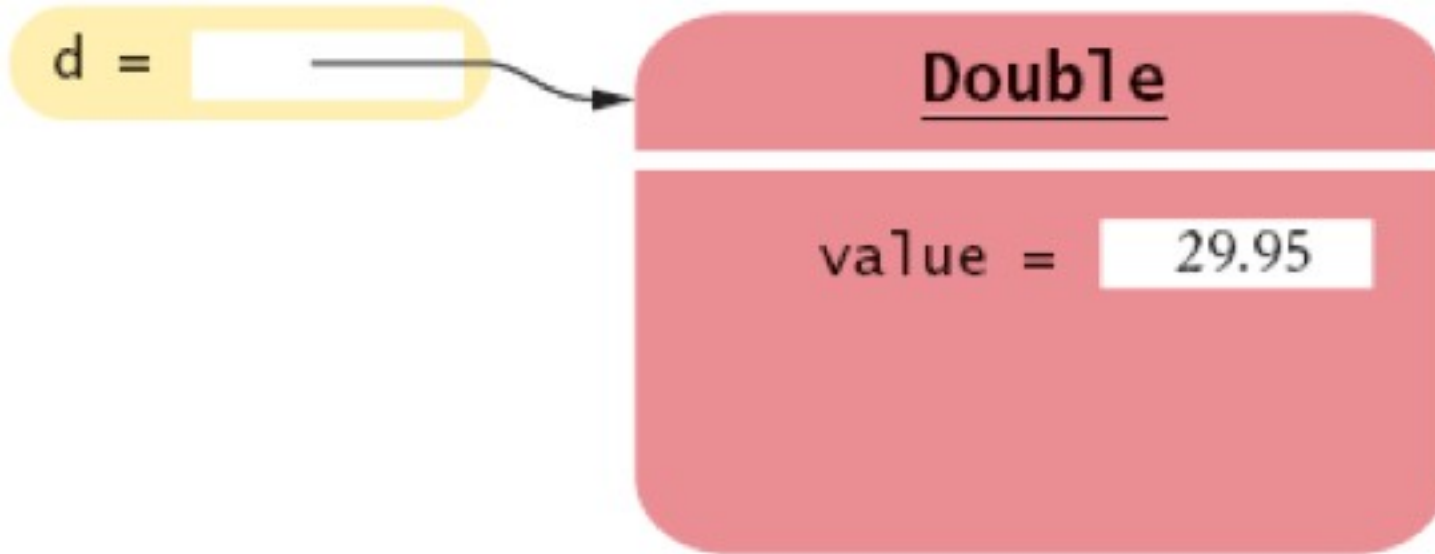
**Réponse :**

`names` contient les chaînes "B" et "C" aux positions 0 et 1

## Classes Enveloppes (Wrappers)

- On ne peut insérer des types primitifs dans une Array List
- Pour traiter des types primitifs comme des objets, on doit utiliser des classes enveloppes :

```
ArrayList<Double> data = new ArrayList<Double>();  
data.add(29.95);  
double x = data.get(0);
```



## Classes Enveloppes (*Wrappers*) I2

Il existe des classes enveloppes pour les 8 types primitifs :

| Primitive Type | Wrapper Class |
|----------------|---------------|
| byte           | Byte          |
| boolean        | Boolean       |
| char           | Character     |
| double         | Double        |
| float          | Float         |
| int            | Integer       |
| long           | Long          |
| short          | Short         |

# Auto-boxing

- Auto-boxing : depuis Java 5.0, la conversion entre type primitif et la classe enveloppe correspondante est automatique

```
Double d = 29.95; // auto-boxing; identique à
    Double d = new Double(29.95);
double x = d; // auto-unboxing; identique à
    x = d.doubleValue();
```

- Auto-boxing fonctionne aussi avec les expressions arithmétiques

```
Double e = d + 1;
```

- Signifie :
  - *auto-unbox d dans un double*
  - *ajoute 1*
  - *auto-box le résultat dans un nouvel objet Double*
  - *Stocke la référence vers le nouvel objet créé dans e*

## Questions

---

Considérons que `ArrayList<Double>` d'une taille  $> 0$ . Comment incrémente-t-on l'élément d'indice 0 ?

**Answer:** `data.set(0, data.get(0) + 1);`

## La boucle for généralisée

- Traverser tous les éléments d'une collection :

```
double[] data = . . . ;
double sum = 0;
for (double e : data) // Doit être lu
                        "pour tout élément e dans data"
    "for each e in data"
{
    sum = sum + e;
}
```

- Alternative "traditionnelle" :

```
double[] data = . . . ;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
    double e = data[i];
    sum = sum + e;
}
```



## La boucle for généralisée /2

- Fonctionne également pour les ArrayLists :

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
    sum = sum + a.getBalance();
}
```
- Equivalent à la boucle for ordinaire :

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    sum = sum + a.getBalance();
}
```

## Syntaxe La boucle généralisée "pour tout"

```
for (Type variable : collection)  
    statement
```

### Exemple :

```
for (double e : data)  
    sum = sum + e;
```

### Objectif :

Exécuter une boucle sur chaque élément d'une collection.

A chaque itération, le prochain élément est affecté à la variable, puis les instructions sont exécutées.

## Questions

---

Ecrivez une boucle "pour tout" qui affiche tous les éléments d'un tableau data.

**Réponse :**

```
for (double x : data) System.out.println(x);
```

## Algorithmes basiques : Compter les éléments satisfaisant

Vérifier pour tous les éléments une condition et compter le nombre d'éléments satisfaisant cette condition.

```
public class Bank
{
    public int count(double atLeast)
    {
        int matches = 0;
        for (BankAccount a : accounts)
        {
            if (a.getBalance() >= atLeast) matches++;
            // Found a match
        }
        return matches;
    }
    . . .
    private ArrayList<BankAccount> accounts;
}
```

## Algorithmes basiques : boucle de recherche

---

Rechercher le premier élément satisfaisant une condition.

```
public class Bank
{
    public BankAccount find(int accountNumber)
    {
        for (BankAccount a : accounts)
        {
            if (a.getAccountNumber() == accountNumber)
                // Found a match
                return a;
        }
        return null; // No match in the entire array list
    }
    . . .
}
```

## Algorithmes basiques : Trouver le maximum/minimum

- Initialiser une valeur candidate avec le premier élément
- Comparer le candidat avec les autres éléments
- Mettre à jour si on trouve une valeur plus grande/petite

- Exemple :

```
BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    if (a.getBalance() > largestYet.getBalance())
        largestYet = a;
}
return largestYet;
```

- Attention : ne fonctionne que si le tableau à au moins 1 élément

## Algorithmes basiques : Trouver le maximum/minimum /2

---

- Si le tableau est vide, retourner `null` :

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
...
```

## ch07/bank/Bank.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     This bank contains a collection of bank accounts.
05: */
06: public class Bank
07: {
08:     /**
09:         Constructs a bank with no bank accounts.
10:     */
11:     public Bank()
12:     {
13:         accounts = new ArrayList<BankAccount>();
14:     }
15:
16:     /**
17:         Adds an account to this bank.
18:         @param a the account to add
19:     */
20:     public void addAccount(BankAccount a)
21:     {
22:         accounts.add(a);
23:     }
```



## ch07/bank/Bank.java /2

```
24:
25:     /**
26:         Gets the sum of the balances of all accounts in this bank.
27:         @return the sum of the balances
28:     */
29:     public double getTotalBalance()
30:     {
31:         double total = 0;
32:         for (BankAccount a : accounts)
33:         {
34:             total = total + a.getBalance();
35:         }
36:         return total;
37:     }
38:
39:     /**
40:         Counts the number of bank accounts whose balance is at
41:         least a given value.
42:         @param atLeast the balance required to count an account
43:         @return the number of accounts having least the given balance
44:     */
45:     public int count(double atLeast)
46:     {
```

## ch07/bank/Bank.java /3

```
47:         int matches = 0;
48:         for (BankAccount a : accounts)
49:         {
50:             if (a.getBalance() >= atLeast) matches++; // Found a match
51:         }
52:         return matches;
53:     }
54:
55:     /**
56:      Finds a bank account with a given number.
57:      @param accountNumber the number to find
58:      @return the account with the given number, or null if there
59:      is no such account
60:     */
61:     public BankAccount find(int accountNumber)
62:     {
63:         for (BankAccount a : accounts)
64:         {
65:             if (a.getAccountNumber() == accountNumber) // Found a match
66:                 return a;
67:         }
68:         return null; // No match in the entire array list
69:     }
70:
```

## ch07/bank/Bank.java /4

```
71:     /**
72:         Gets the bank account with the largest balance.
73:         @return the account with the largest balance, or null if the
74:         bank has no accounts
75:     */
76:     public BankAccount getMaximum()
77:     {
78:         if (accounts.size() == 0) return null;
79:         BankAccount largestYet = accounts.get(0);
80:         for (int i = 1; i < accounts.size(); i++)
81:         {
82:             BankAccount a = accounts.get(i);
83:             if (a.getBalance() > largestYet.getBalance())
84:                 largestYet = a;
85:         }
86:         return largestYet;
87:     }
88:
89:     private ArrayList<BankAccount> accounts;
90: }
```

## ch07/bankBankTester.java

```
01: /**
02:     This program tests the Bank class.
03: */
04: public class BankTester
05: {
06:     public static void main(String[] args)
07:     {
08:         Bank firstBankOfJava = new Bank();
09:         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
10:         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
11:         firstBankOfJava.addAccount(new BankAccount(1729, 15000));
12:
13:         double threshold = 15000;
14:         int c = firstBankOfJava.count(threshold);
15:         System.out.println("Count: " + c);
16:         System.out.println("Expected: 2");
17:
18:         int accountNumber = 1015;
19:         BankAccount a = firstBankOfJava.find(accountNumber);
20:         if (a == null)
```

## ch07/bankBankTester.java /2

```
21:         System.out.println("No matching account");
22:     else
23:         System.out.println("Balance of matching account: " +
                             a.getBalance());
24:     System.out.println("Expected: 10000");
25:
26:     BankAccount max = firstBankOfJava.getMaximum();
27:     System.out.println("Account with largest balance: "
28:         + max.getAccountNumber());
29:     System.out.println("Expected: 1001");
30: }
31: }
```

### Output:

Count: 2

Expected: 2

Balance of matching account: 10000.0

Expected: 10000

Account with largest balance: 1001

Expected: 1001

## Questions

---

Que fait la méthode `find` si il y a deux comptes bancaires avec le même numéro de compte ?

**Réponse :** Elle retourne toujours le premier compte trouvé.

## Questions /2

---

Peut-on utiliser une boucle “pour tout” dans la méthode `getMaximum` ?

**Réponse :** Oui, mais la première comparaison échoue toujours

## Tableaux à 2 dimensions

---

- Pour construire un tableau à 2 dimensions, il faut spécifier la taille pour chaque dimension :

```
final int ROWS = 3;  
final int COLUMNS = 3;  
String[][] board = new String[ROWS][COLUMNS];
```

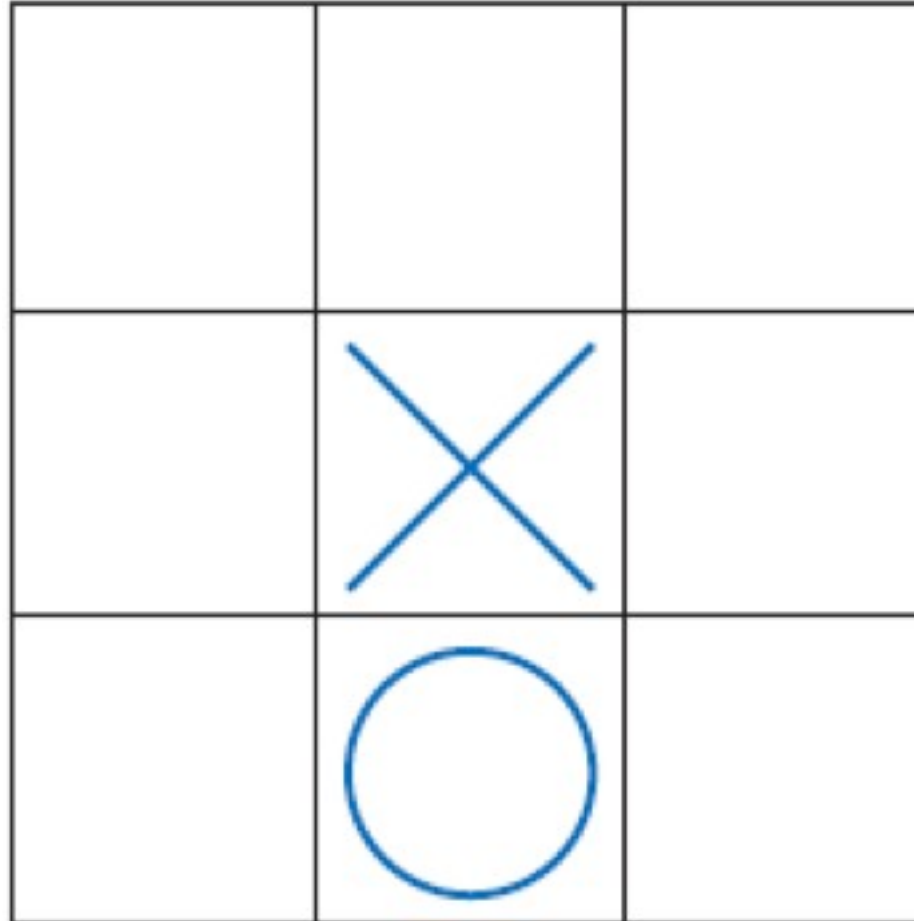
- Accès aux éléments par une paire d'indices `a[i][j]`

```
board[i][j] = "x";
```



# Plateau du Morpion

---



## Parcours de tableau à 2 dimensions

---

Généralement, on utilise deux boucles imbriquées :

```
for (int i = 0; i < ROWS; i++)  
    for (int j = 0; j < COLUMNS; j++)  
        board[i][j] = " ";
```

## ch07/twodim/TicTacToe.java

```
01: /**
02:     A 3 x 3 tic-tac-toe board.
03: */
04: public class TicTacToe
05: {
06:     /**
07:         Constructs an empty board.
08:     */
09:     public TicTacToe()
10:     {
11:         board = new String[ROWS][COLUMNS];
12:         // Fill with spaces
13:         for (int i = 0; i < ROWS; i++)
14:             for (int j = 0; j < COLUMNS; j++)
15:                 board[i][j] = " ";
16:     }
17:
18:     /**
19:         Sets a field in the board. The field must be unoccupied.
20:         @param i the row index
21:         @param j the column index
22:         @param player the player ("x" or "o")
23:     */
```

## ch07/twodim/TicTacToe.java /2

```
24:     public void set(int i, int j, String player)
25:     {
26:         if (board[i][j].equals(" "))
27:             board[i][j] = player;
28:     }
29:
30:     /**
31:      * Creates a string representation of the board, such as
32:      * |x  o|
33:      * |  x |
34:      * |  o|
35:      * @return the string representation
36:      */
37:     public String toString()
38:     {
39:         String r = "";
40:         for (int i = 0; i < ROWS; i++)
41:         {
42:             r = r + "|";
43:             for (int j = 0; j < COLUMNS; j++)
44:                 r = r + board[i][j];
45:             r = r + "|\n";
```

## ch07/twodim/TicTacToe.java /3

---

```
46:     }
47:     return r;
48: }
49:
50: private String[][] board;
51: private static final int ROWS = 3;
52: private static final int COLUMNS = 3;
53: }
```

## ch07/twodim/TicTacToeRunner.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program runs a TicTacToe game. It prompts the
05:     user to set positions on the board and prints out the
06:     result.
07: */
08: public class TicTacToeRunner
09: {
10:     public static void main(String[] args)
11:     {
12:         Scanner in = new Scanner(System.in);
13:         String player = "x";
14:         TicTacToe game = new TicTacToe();
15:         boolean done = false;
16:         while (!done)
17:         {
18:             System.out.print(game.toString());
19:             System.out.print(
20:                 "Row for " + player + " (-1 to exit): ");
21:             int row = in.nextInt();
22:             if (row < 0) done = true;
23:             else
24:                 {
```

## ch07/twodim/TicTacToeRunner.java /2

```
25:         System.out.print("Column for " + player + ": ");
26:         int column = in.nextInt();
27:         game.set(row, column, player);
28:         if (player.equals("x"))
29:             player = "o";
30:         else
31:             player = "x";
32:     }
33: }
34: }
35: }
```

## Output:

```
| |  
| |  
| |
```

Row for x (-1 to exit): **1**

Column for x: **2**

```
| |  
| x |  
| |
```

Row for o (-1 to exit): **0**

Column for o: **0**

```
| o |  
| x |  
| |
```

Row for x (-1 to exit): **-1**



## Questions

---

Comment déclare-t-on un tableau de 4x4 valeurs entières ?

**Réponse :**

```
int[][] array = new int[4][4];
```

## Questions /2

---

Comment calcule-t-on le nombre de case vide d'un plateau de morpion ?

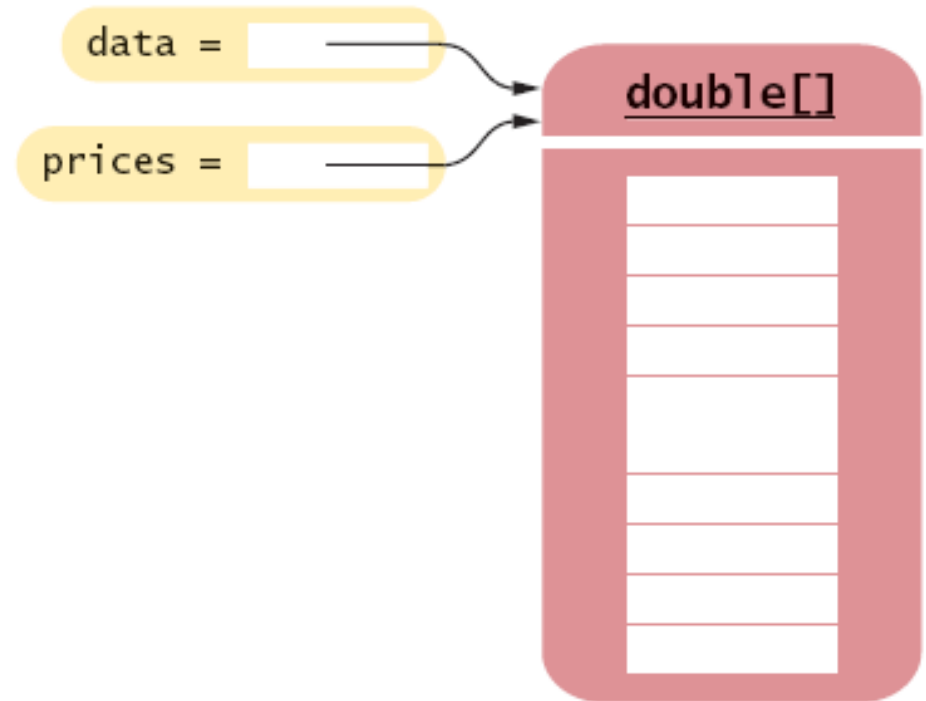
**Réponse :**

```
int count = 0;
for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLUMNS; j++)
        if (board[i][j] == ' ') count++;
```

## Copie de tableaux : Copie par référence

Copier une variable tableau génère une seconde référence vers le même tableau

```
Double[ ] data = new double[10];  
// fill array . . .  
Double[ ] prices = data;
```

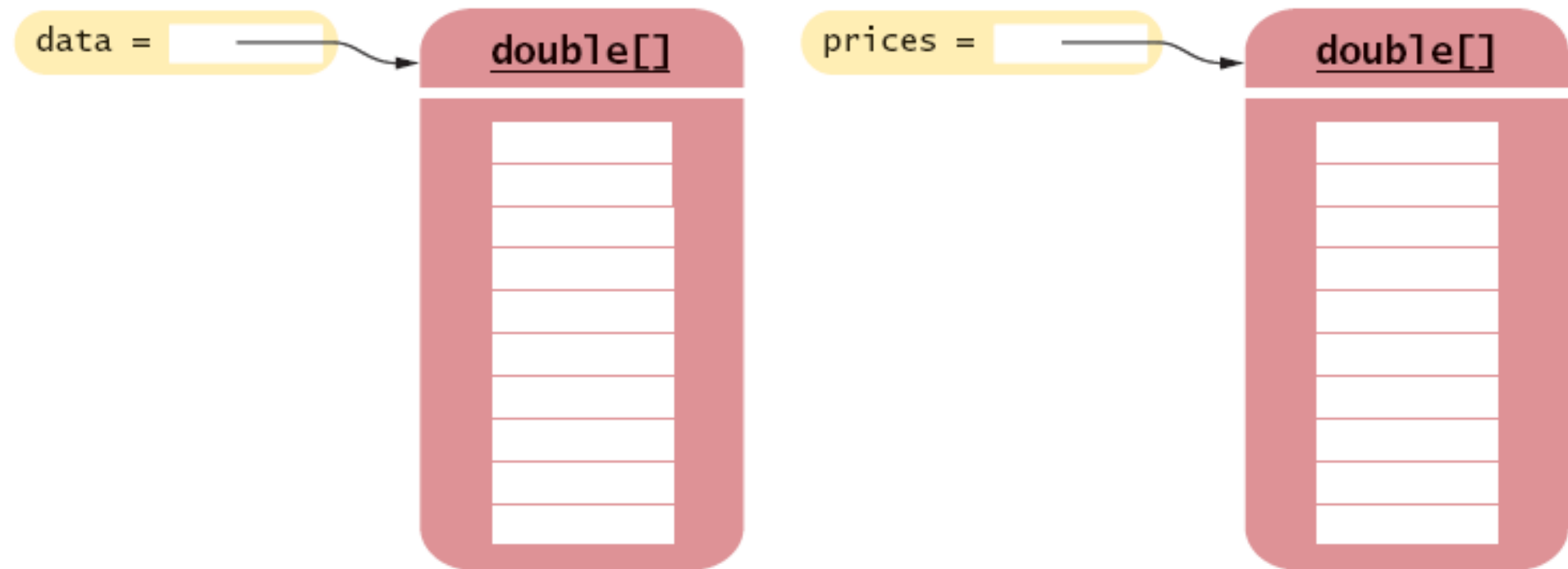


**Figure 7** Two References to the Same Array

# Copie de tableaux : Cloner

Utiliser `clone` pour faire une vraie copie par valeur

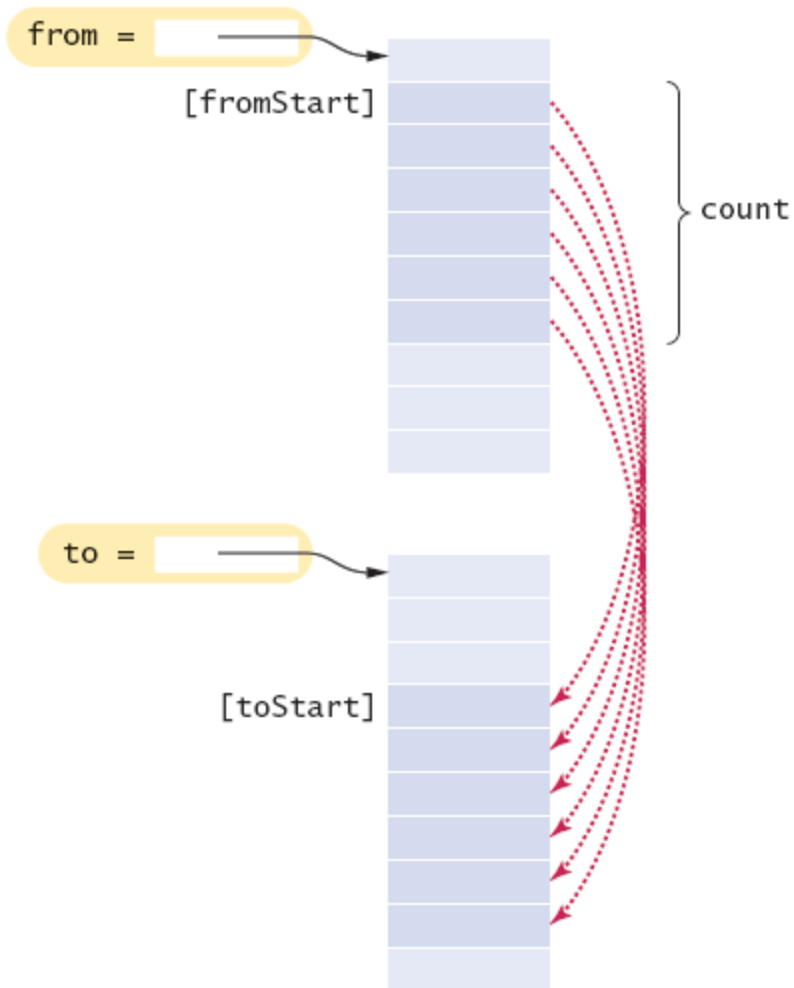
```
Double[ ] prices = (double[ ]) data.clone();
```



**Figure 8** Cloning an Array

# Copie de tableaux : Recopier les éléments d'un tableau

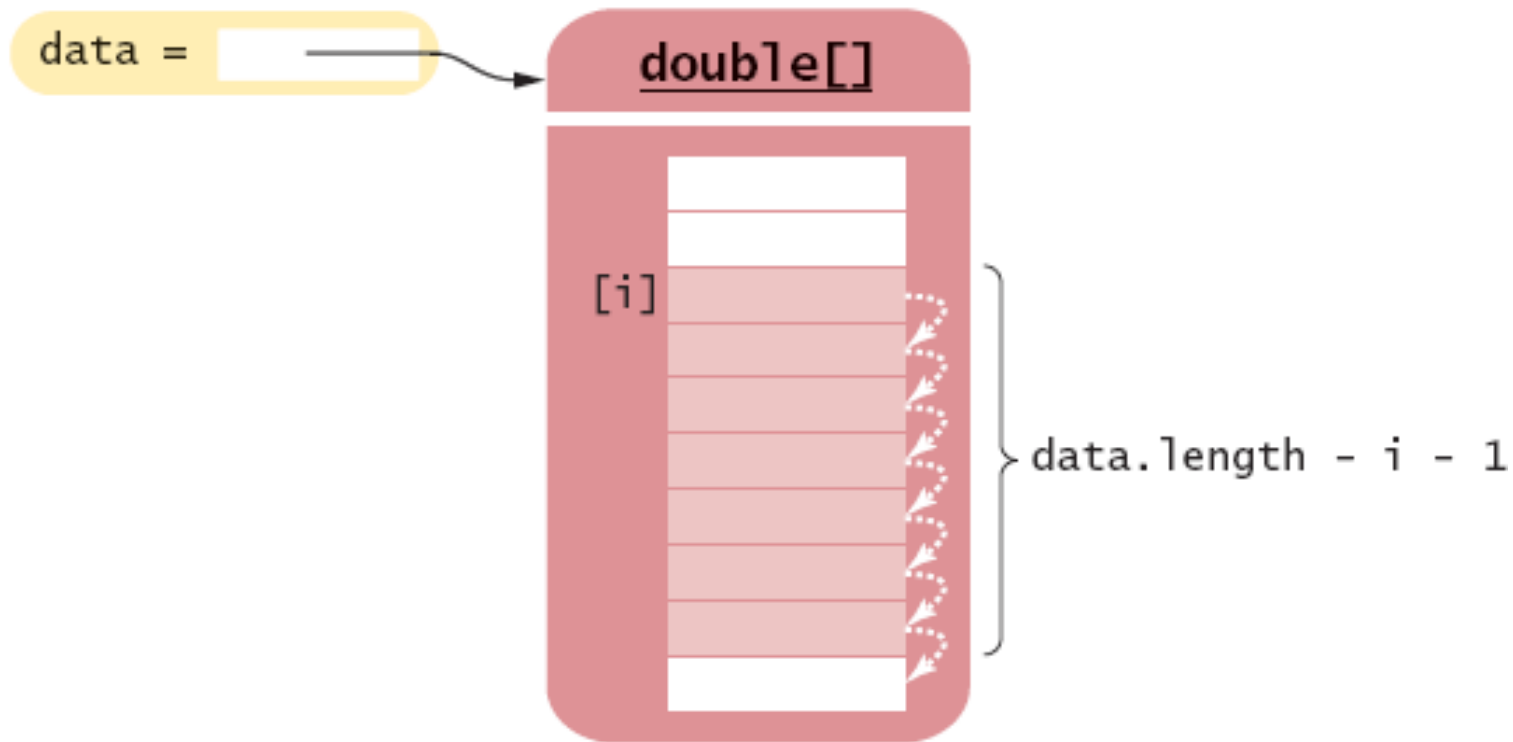
```
System.arraycopy(from, fromStart, to, toStart, count);
```



**Figure 9** The `System.arraycopy` Method

# Insertion d'une entrée dans un tableau

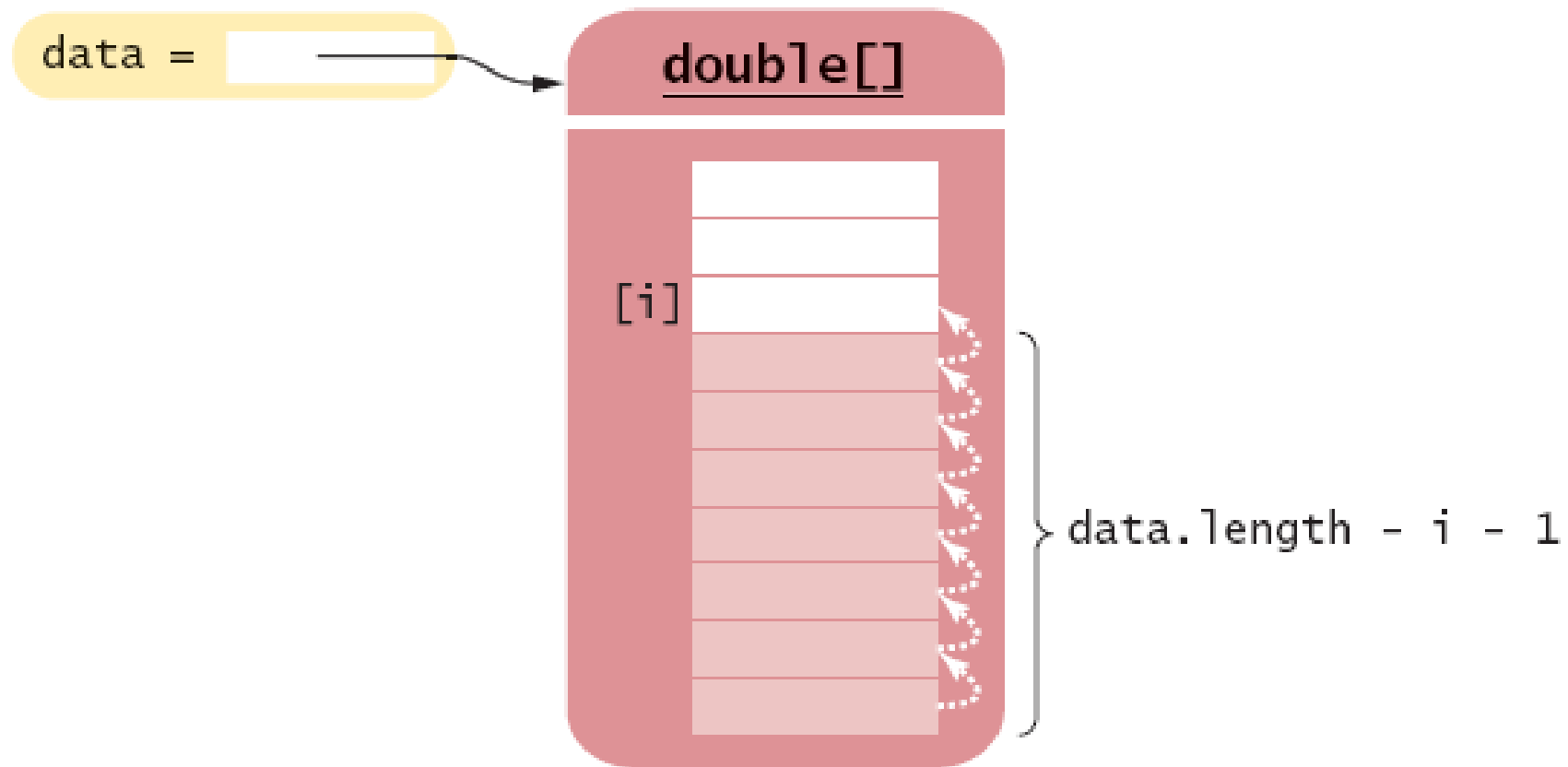
```
System.arraycopy(data, i, data, i + 1, data.length - i  
                - 1);  
data[i] = x;
```



**Figure 10** Inserting a New Element into an Array

# Suppression d'une entrée d'un tableau

```
System.arraycopy(data, i + 1, data, i, data.length - i  
- 1);
```



**Figure 11** Removing an Element from an Array

## Redimensionner un tableau

---

- Si un tableau est plein et que l'on a besoin de plus d'espace, on peut le redimensionner:
- Créer un tableau plus large :  
`double[] newData = new double[2 * data.length];`
- Recopier tous les éléments dans le nouveau tableau :  
`System.arraycopy(data, 0, newData, 0, data.length);`
- Stocker la référence du nouveau tableau dans la variable de référence du tableau :  
`data = newData;`

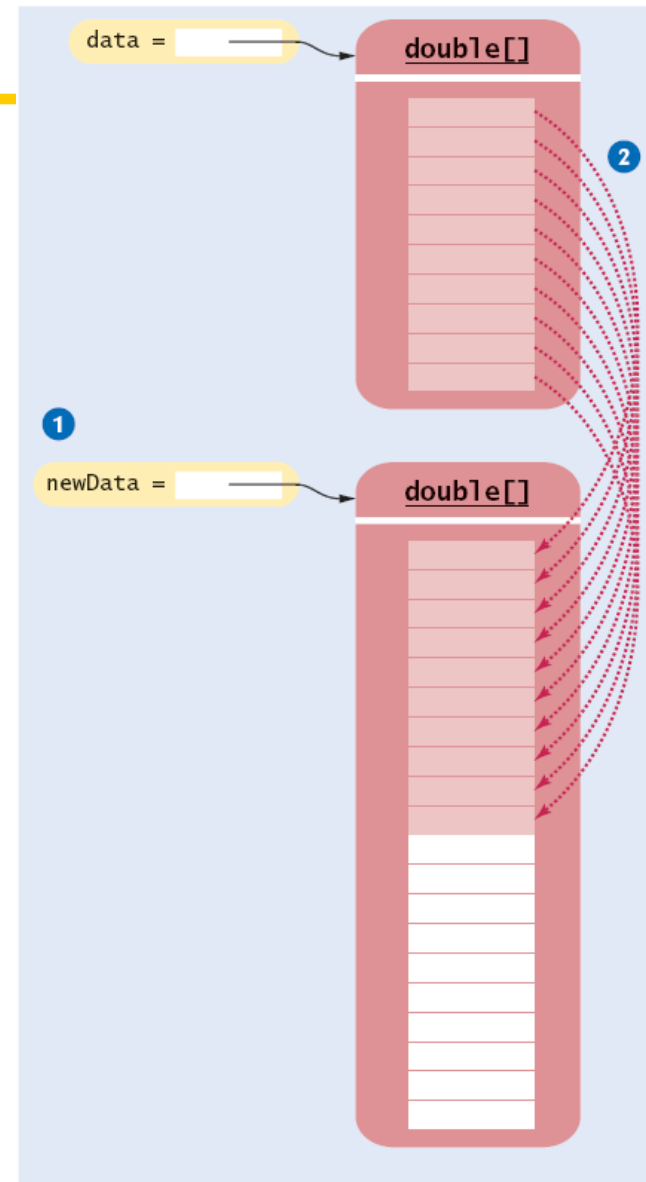


## Redimensionner un tableau /2

---

```
Double[ ] newData = new double[2 * data.length] ①  
System.arraycopy(data, 0, newData, 0, data.length) ②
```

# Redimensionner un tableau /3



**Figure 12** Growing an Array

## Redimensionner un tableau /4

---

```
double[] newData = new double[2 * data.length]; ①  
System.arraycopy(data, 0, newData, 0, data.length); ②  
data = newData; ③
```

# Redimensionner un tableau /5

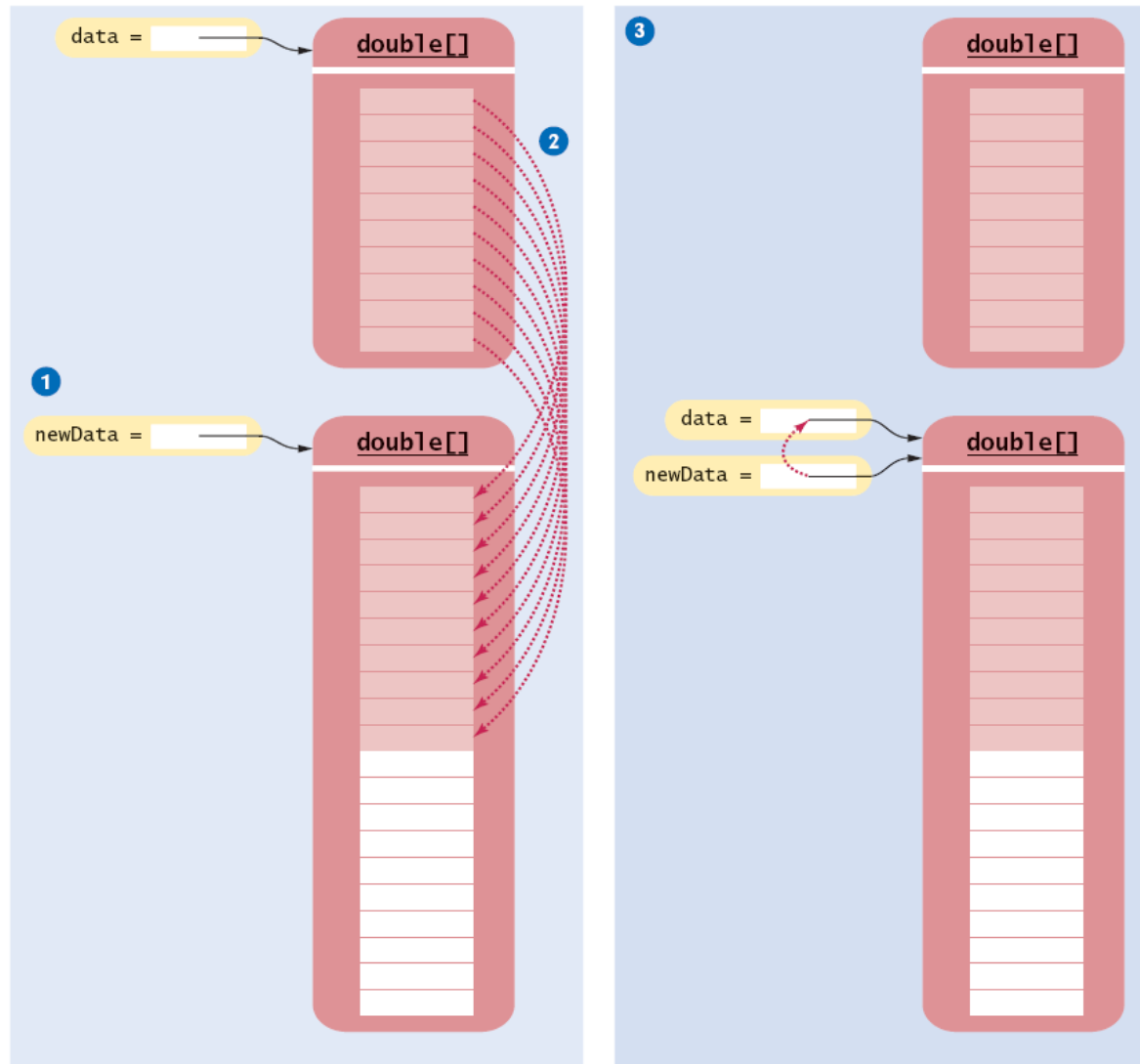


Figure 12 Growing an Array

## Questions

---

Comment ajoute-t-on ou supprime-t-on des éléments au milieu d'une ArrayList ?

**Réponse :**

Utiliser simplement les méthodes pour insérer et supprimer

## Questions /2

---

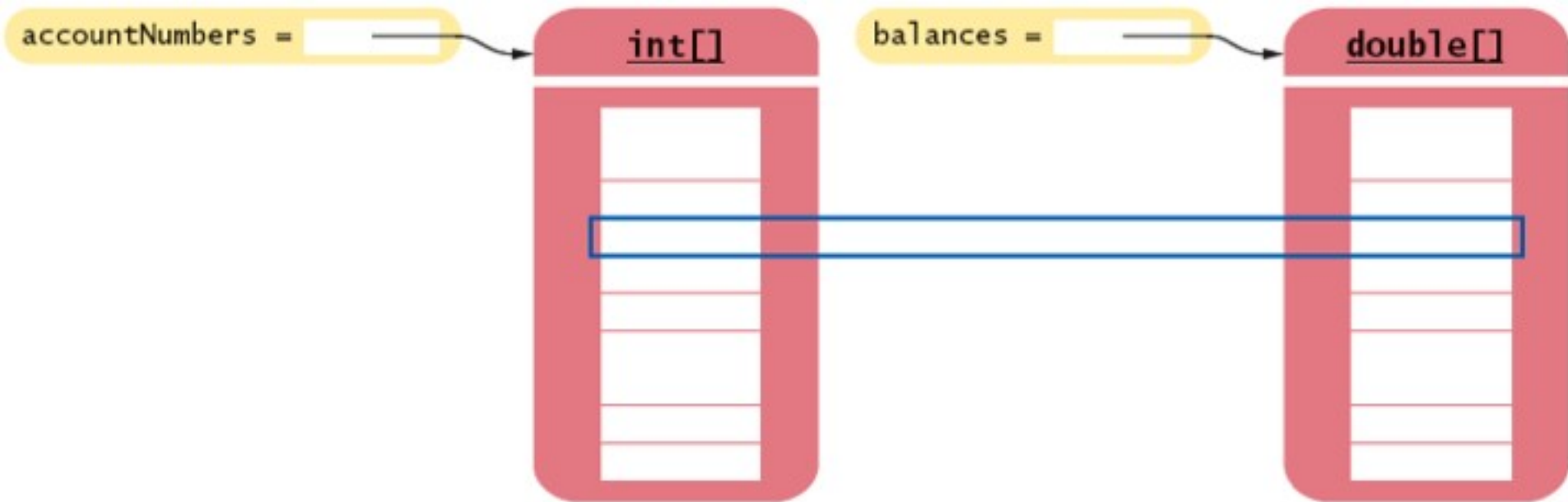
A votre avis, pourquoi double-t-on la taille d'un tableau lorsque l'on a plus d'espace libre au lieu d'ajouter une cellule ?

### **Réponse :**

Allouer un nouveau tableau et recopier les valeurs est coûteux en temps. On ne souhaite donc pas re-exécuter tout le processus à chaque insertion.

# Tableaux parallèles

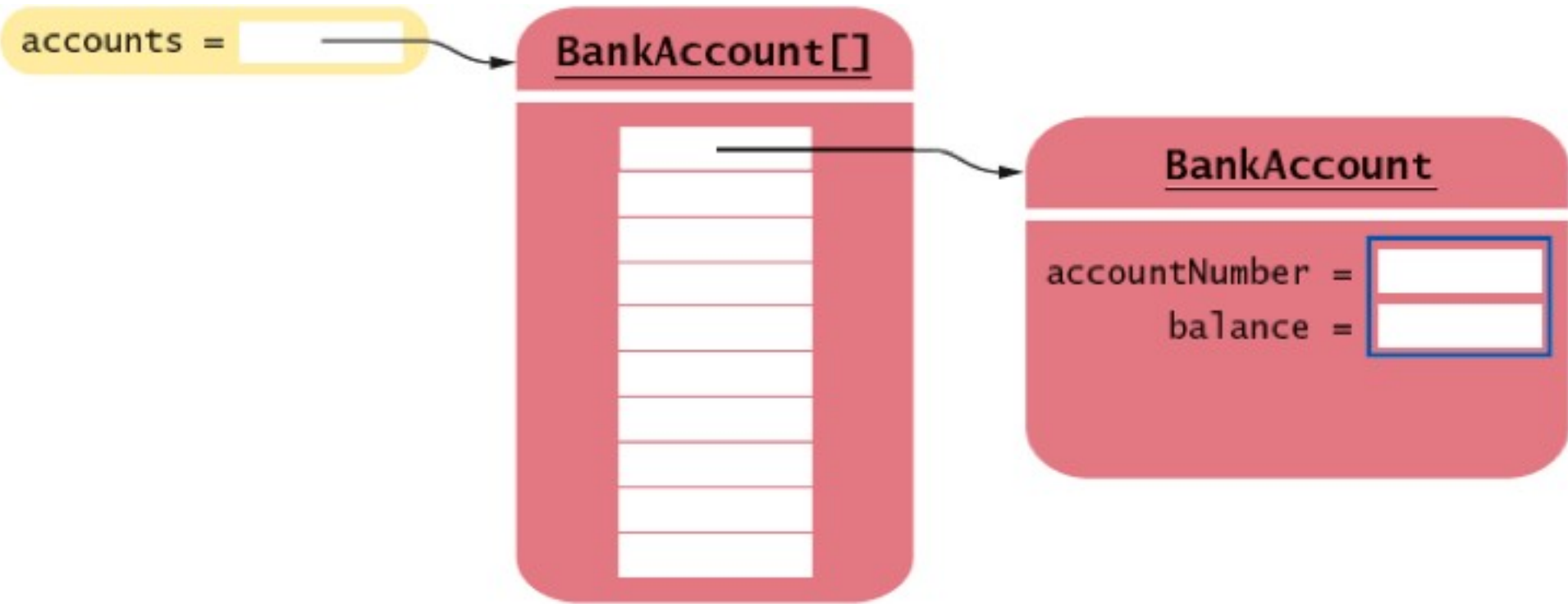
```
// Ne faites pas ça !!!  
int[] accountNumbers;  
double[] balances;
```



## Tableaux parallèles /2

Eviter ce genre de tableaux parallèles en les transformant en 1  
tableau d'objets :

```
BankAccount[] = accounts
```





## Tableaux partiellement remplis

---

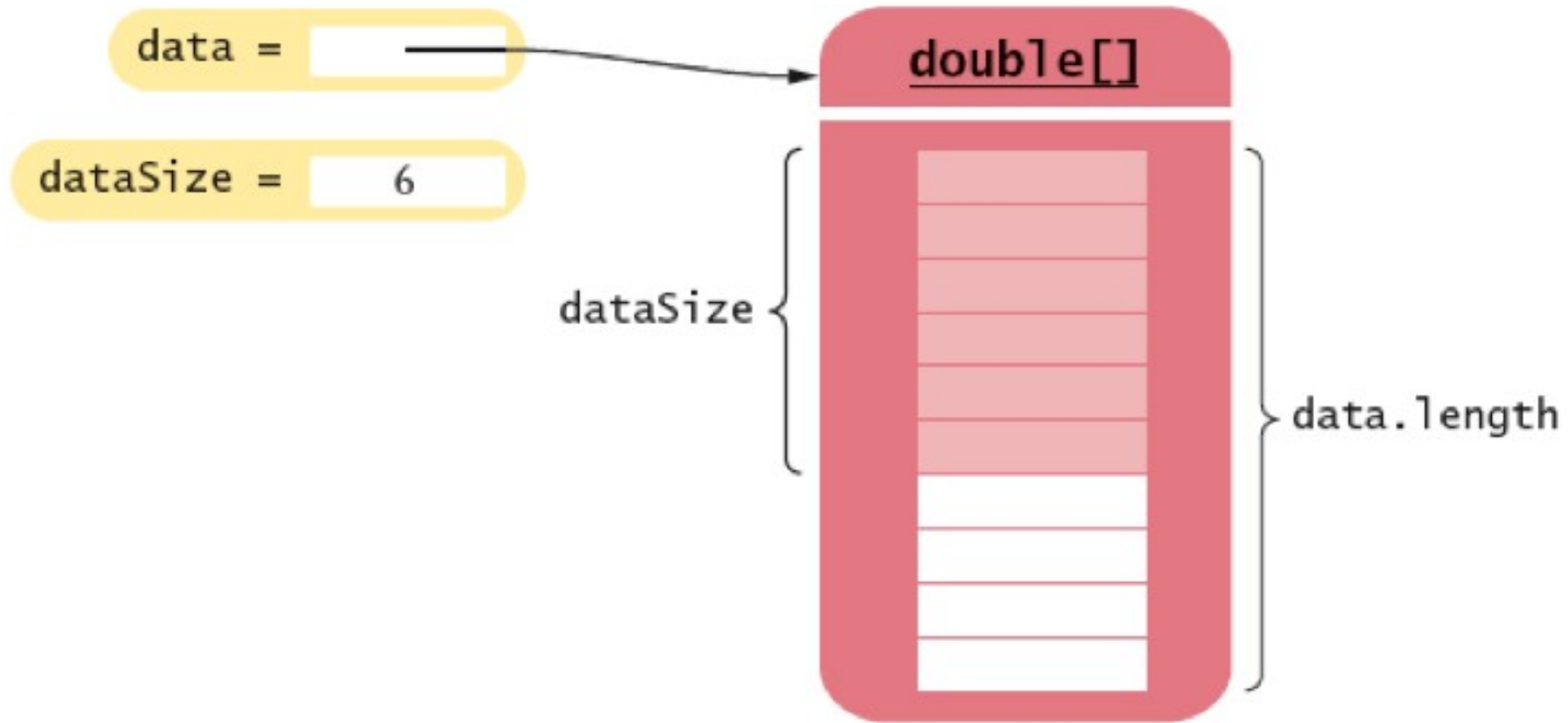
- Longueur tableau = nombre maximum d'éléments
- Généralement, partiellement remplis
- Besoin d'une variable companion pour conserver la taille courante

- Utiliser une convention de notation uniforme :

```
final int DATA_LENGTH = 100;  
double[] data = new double[DATA_LENGTH];  
int dataSize = 0;
```

- Mettre à jour dataSize lors l'insertion/suppression d'une élément :  
data[dataSize] = x;  
dataSize++;

## Tableaux partiellement remplis /2



# Tests de regression

---

- Conserver vos tests (*test case*)
- Utiliser les tests que vous avez conservés pour les versions suivantes de votre programme
- Une suite de test (*test suite*) est un ensemble de test que l'on peut répéter
- Cyclique = un bug fixé re-apparaît souvent dans une version suivante
- Test de regression: répéter les tests précédents pour s'assurer que les erreurs antérieures ne ré-apparaissent pas dans les nouvelles versions

## ch07/regression/BankTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program tests the Bank class.
05: */
06: public class BankTester
07: {
08:     public static void main(String[] args)
09:     {
10:         Bank firstBankOfJava = new Bank();
11:         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
12:         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
13:         firstBankOfJava.addAccount(new BankAccount(1729, 15000));
14:
15:         Scanner in = new Scanner(System.in);
16:
17:         double threshold = in.nextDouble();
18:         int c = firstBankOfJava.count(threshold);
19:         System.out.println("Count: " + c);
20:         int expectedCount = in.nextInt();
21:         System.out.println("Expected: " + expectedCount);
22:
```

## ch07/regression/BankTester.java /2

```
23:         int accountNumber = in.nextInt();
24:         BankAccount a = firstBankOfJava.find(accountNumber);
25:         if (a == null)
26:             System.out.println("No matching account");
27:         else
28:             {
29:                 System.out.println("Balance of matching account: " +
a.getBalance());
30:                 int matchingBalance = in.nextLine();
31:                 System.out.println("Expected: " + matchingBalance);
32:             }
33:     }
34: }
```

# Redirection des entrées-sorties

---

- Stocker les entrées dans un fichier
- ch07/regression/input1.txt:  
15000  
2  
1015  
10000
- Saisissez la commande suivante :  
`java BankTester < input1.txt`
- Sortie :  
Count: 2  
Expected: 2  
Balance of matching account: 10000  
Expected: 10000
- Redirection des sorties :  
`java BankTester < input1.txt > output1.txt`

# Questions

---

Supposons qu'un client a trouvé une erreur dans votre programme. Quelles actions devez vous entreprendre ?

## **Réponse :**

- Ecrire un test qui reproduit l'erreur,
- Corriger l'erreur,
- Conserver le test

# 5<sup>ème</sup> Partie : Introduction à la conception objets



## Objectifs de cette partie

---

- Apprendre à choisir les classes appropriées à implémenter
- Comprendre les notions de *cohésion* et de *couplage*
- Minimiser les effets de bord (*side effects*)
- Documenter les responsabilités de chaque méthodes et leurs appelants avec des pré-conditions et des post-conditions
- Comprendre la différence entre méthodes d'instance et de classe
- Introduire la notion de variable de classe
- Comprendre les règles de portée des variables locales et des variables d'instance
- Découvrir la notion de package

# Découvrir et choisir des classes

---

- Une classe représente un unique concept/notion du monde du problème (chercher les noms dans l'énoncé du problème)
- Le nom d'une classe est généralement un nom qui décrit un concept
- Concepts mathématiques :
  - Point
  - Rectangle
  - Ellipse
- Concepts de la vie de tous les jours :
  - BankAccount
  - CashRegister

## Découvrir et choisir des classes /2

---

- Acteurs – Objets qui “travaille pour vous”  
Scanner  
Random // meilleur nom: RandomNumberGenerator
- Classes utilitaires – pas d’objet (instance) seulement des méthodes de classes  
Math
- Programme principal : contient uniquement une méthode `main`
- Ne transformer pas les actions en classe :  
Paycheck est un meilleur nom que `ComputePaycheck`

## Questions

---

On vous demande d'écrire un programme de jeu d'échec.  
ChessBoard peut-elle être une classe ? Et MovePiece?

**Réponse :** Oui (ChessBoard) et Non (MovePiece).

# Cohésion

- Une classe doit représenter un seul concept
- L'interface publique d'une classe est cohésive si toutes ses fonctionnalités sont en relation avec le concept représenté
- Cette classe manque de cohésion :

```
public class CashRegister
{
    public void enterPayment(int dollars, int quarters,
        int dimes, int nickels, int pennies)
        . . .
    public static final double NICKEL_VALUE = 0.05;
    public static final double DIME_VALUE = 0.1;
    public static final double QUARTER_VALUE = 0.25;
    . . .
}
```

# Cohésion

---

CashRegister, comme décrit précédemment, inclu deux concepts:  
*cash register* et *coin*

Solution : Faire deux classes :

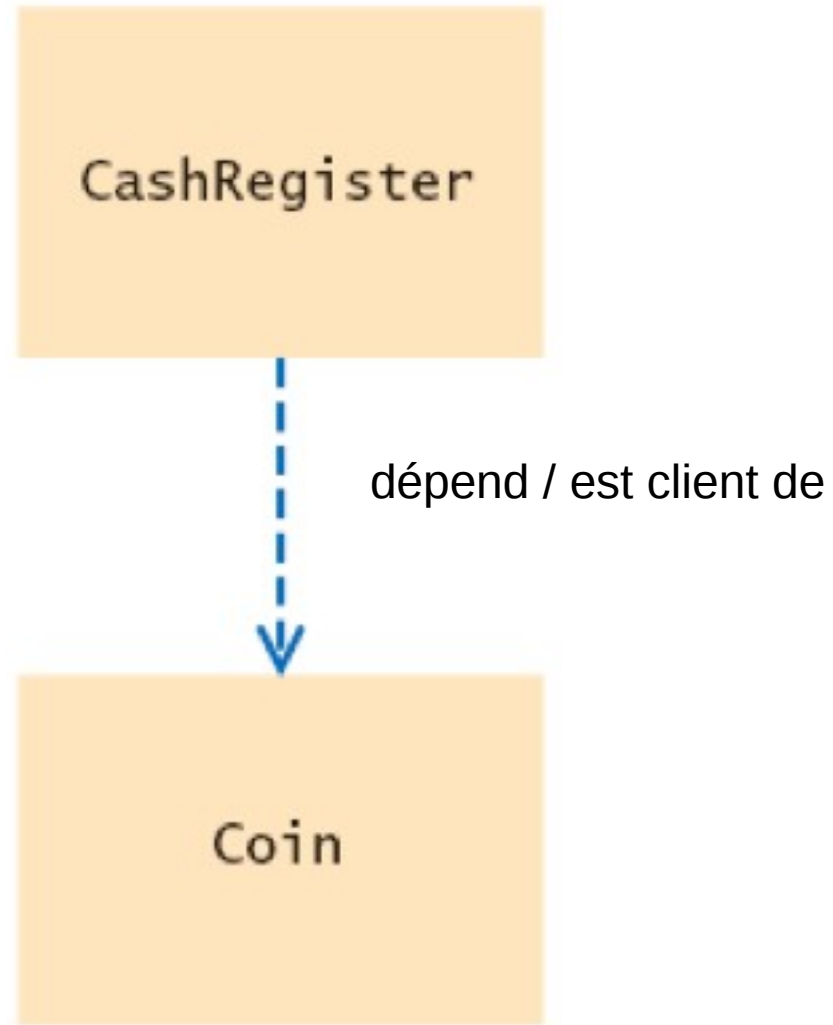
```
public class Coin
{
    public Coin(double aValue, String aName) { . . . }
    public double getValue() { . . . }
    . . .
}
public class CashRegister
{
    public void enterPayment(int coinCount, Coin coinType)
        { . . . }
    . . .
}
```

# Couplage

---

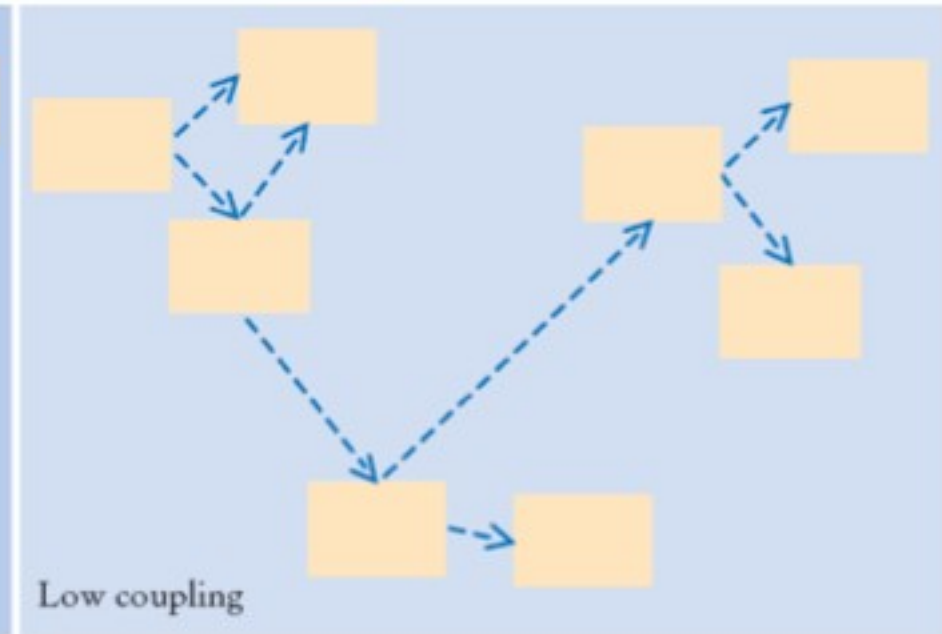
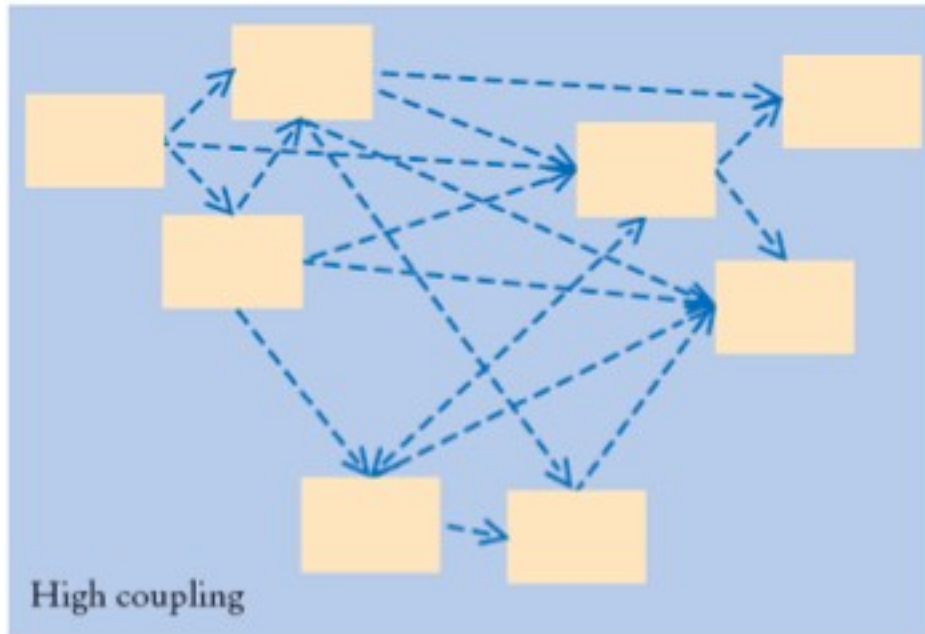
- Une classe *dépend* d'une autre classe si elle utilise des instances de cette seconde classe
- `CashRegister` dépend de `coin` pour déterminer la valeur du paiement
- `coin` ne dépend pas de `CashRegister`
- Couplage fort = beaucoup de dépendances entre classes
- Minimiser le couplage pour minimiser l'impact du changement d'une interface
- Pour visualiser les relations entre classes, dessinez des diagrammes
- UML: Unified Modeling Language. Une notation pour l'analyse et la conception orientée objet

# Couplage





# Couplage fort et faible entre des classes



## Questions

---

Pourquoi la classe `coin` ne dépend pas de la classe `CashRegister` ?

**Réponse :**

Aucune utilisation des méthodes de `coin` n'est requise dans la classe `CashRegister`

## Questions /2

---

Pourquoi doit-on minimiser le couplage entre classes ?

**Réponse :**

Si une classe ne dépend pas d'une autre, alors elle ne peut être affectée par un changement de l'interface de cette dernière.



## Questions

---

La méthode `substring` est elle un accesseur ou modificateur de la classe `String` ?

**Réponse** : c'est un accesseur – appeler `substring` ne modifie pas l'objet. En fait toutes les méthode de la classe `String` sont des accesseurs

## Questions /2

---

La classe `Rectangle` est-elle non mutable ?

**Réponse** : Non – la méthode `translate` est un modificateur.

## Effets de bord (Side Effects)

---

- Effet de bord d'une méthode : toute modification observable de l'extérieur de la méthode

```
public void transfer(double amount, BankAccount other)
{
    balance = balance - amount;
    other.balance = other.balance + amount; // Modifies
    explicit parameter
}
```

- Mettre à jour les paramètres (explicite) d'une méthode peut engendrer des surprises au programmeur ; il est préférable d'éviter ce genre de manipulation si possible

## Effets de bord (Side Effects) /2

---

- Un autre exemple d'effet de bord : l'affichage

```
public void printBalance() // Not recommended
{
    System.out.println("The balance is now $" + balance);
}
```

Mauvaise idée : le message est en anglais, et il utilise `System.out`. Il est préférable de découpler entrée/sortie du fonctionnement de votre classe

- Vous devez minimiser les effets de bord qui modifie autre chose que le receveur (paramètre implicite)



## Questions

---

Si a référence un compte bancaire, l'appel `a.deposit(100)` modifie ce compte bancaire. Est-ce un effet de bord ?

**Réponse :** Non – un effet de bord d'une méthode modifie autre chose que le receveur de l'appel de méthode.

## Questions /2

---

Considérons cette méthode

```
void read(Scanner in)
{
    while (in.hasNextDouble())
        add(in.nextDouble());
}
```

A-t-elle des effets de bord ?

**Réponse :** Oui – La méthode affecte l'état de l'objet Scanner passé en paramètre

## Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre

- ```
void transfer(double amount, double otherBalance)
{
    balance = balance - amount;
    otherBalance = otherBalance + amount;
}
```

- Ne fonctionne pas

- Scenario :

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
System.out.println(savingsBalance);
```

- En Java, une méthode ne peut jamais modifier la valeur d'une variable de type primitive passée en paramètre

## Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /2

```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance);  
...  
void transfer(double amount, double otherBalance)  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
}
```

❶ Before method call

harrysChecking =

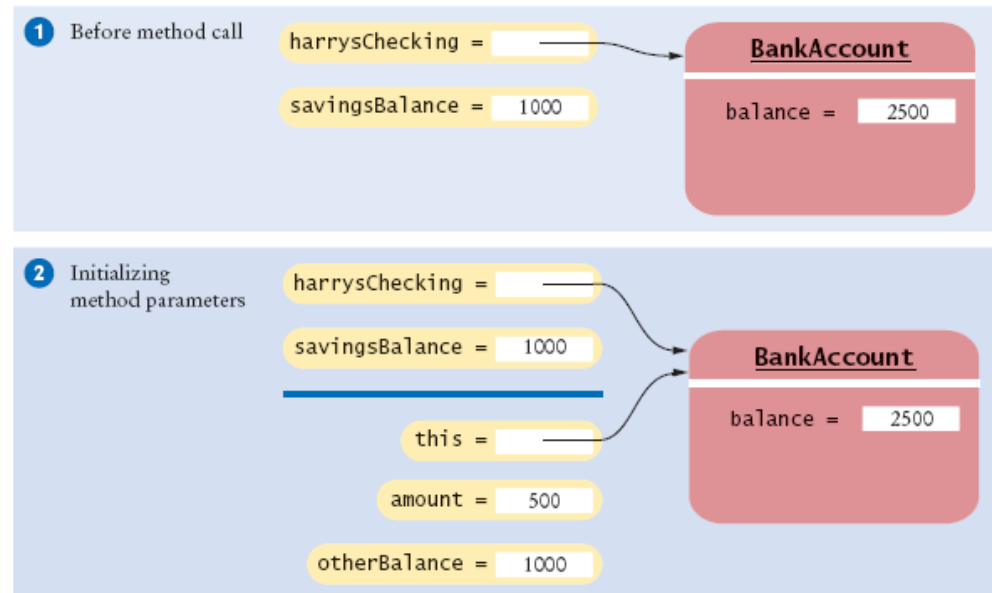
savingsBalance =

**BankAccount**

balance =

# Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /3

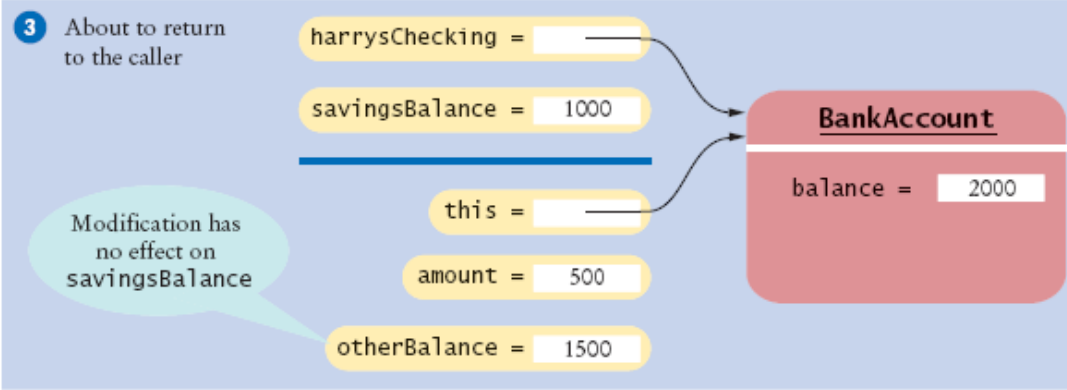
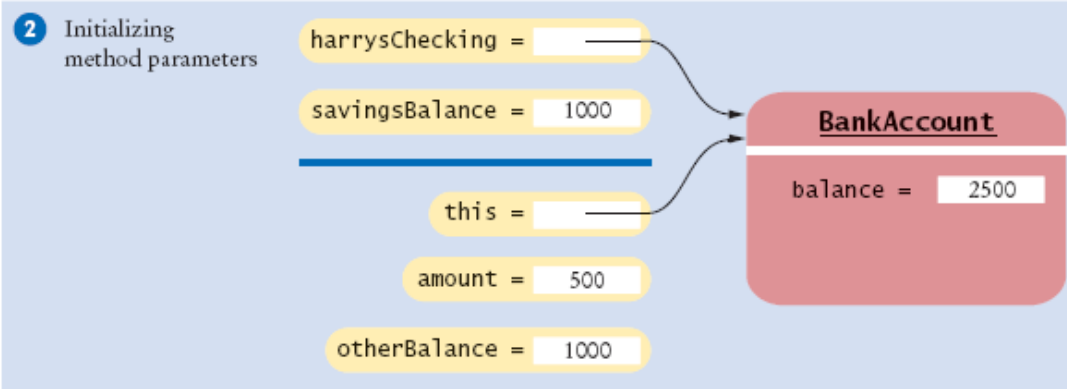
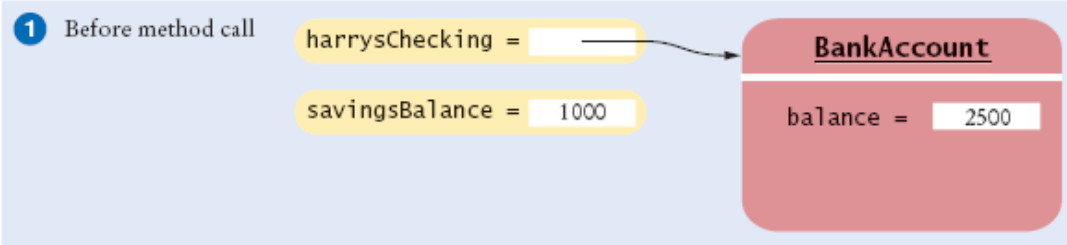
```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance);  
...  
void transfer(double amount, double otherBalance) ❷  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
}
```



## Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /4

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) ❷
{
    balance = balance - amount;
    otherBalance = otherBalance + amount;
} ❸
```

# Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /5



## Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /6

```
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance); ❶  
System.out.println(savingsBalance); ❷  
...  
void transfer(double amount, double otherBalance) ❸  
{  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
} ❹
```



# Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /7

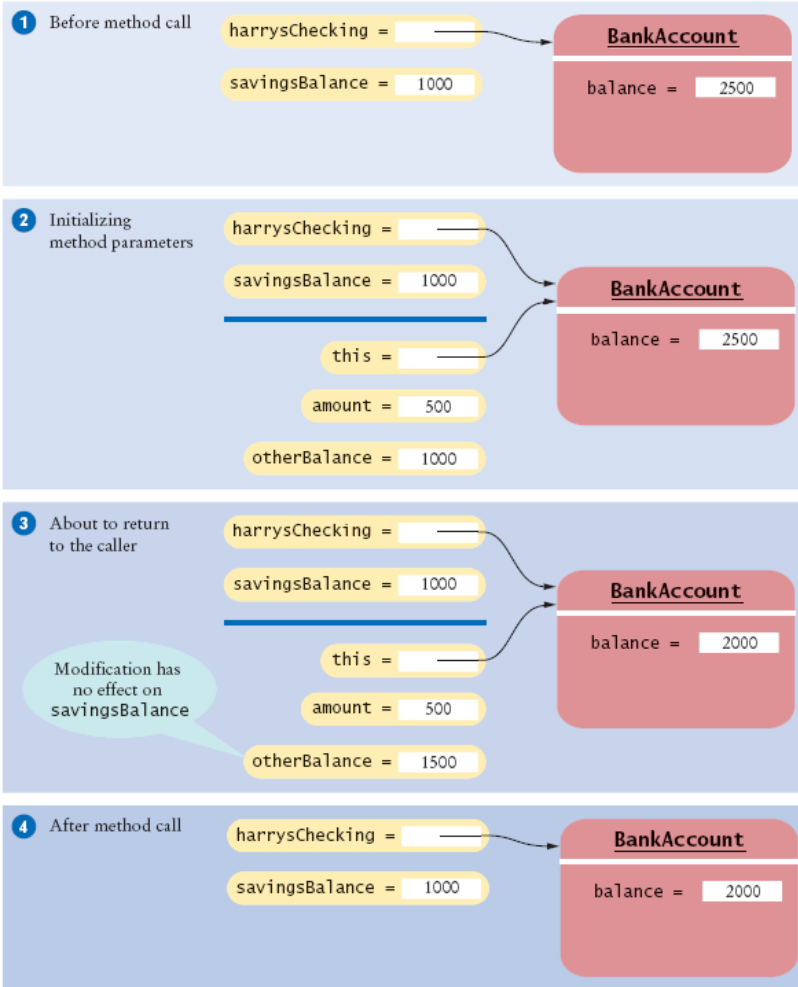


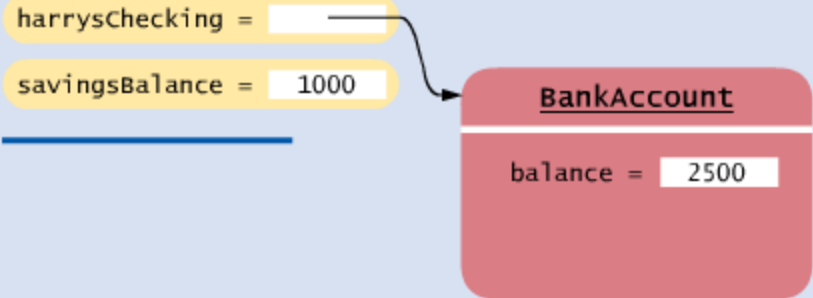
Figure 3 Modifying a Numeric Parameter Has No Effect on Caller

# Animation 8.1 –

```
public static void main(String[] args)
{
    BankAccount harrysChecking = new BankAccount(2500);
    double savingsBalance = 1000;
    harrysChecking.transfer(500, savingsBalance);
    System.out.println(savingsBalance);
}

...

public void transfer(double amount, double otherBalance)
{
    balance = balance - amount;
    otherBalance = otherBalance + amount;
    // won't work
}
```



The implicit parameter variable `this` is created and initialized.

## Passage par valeur et passage par référence

---

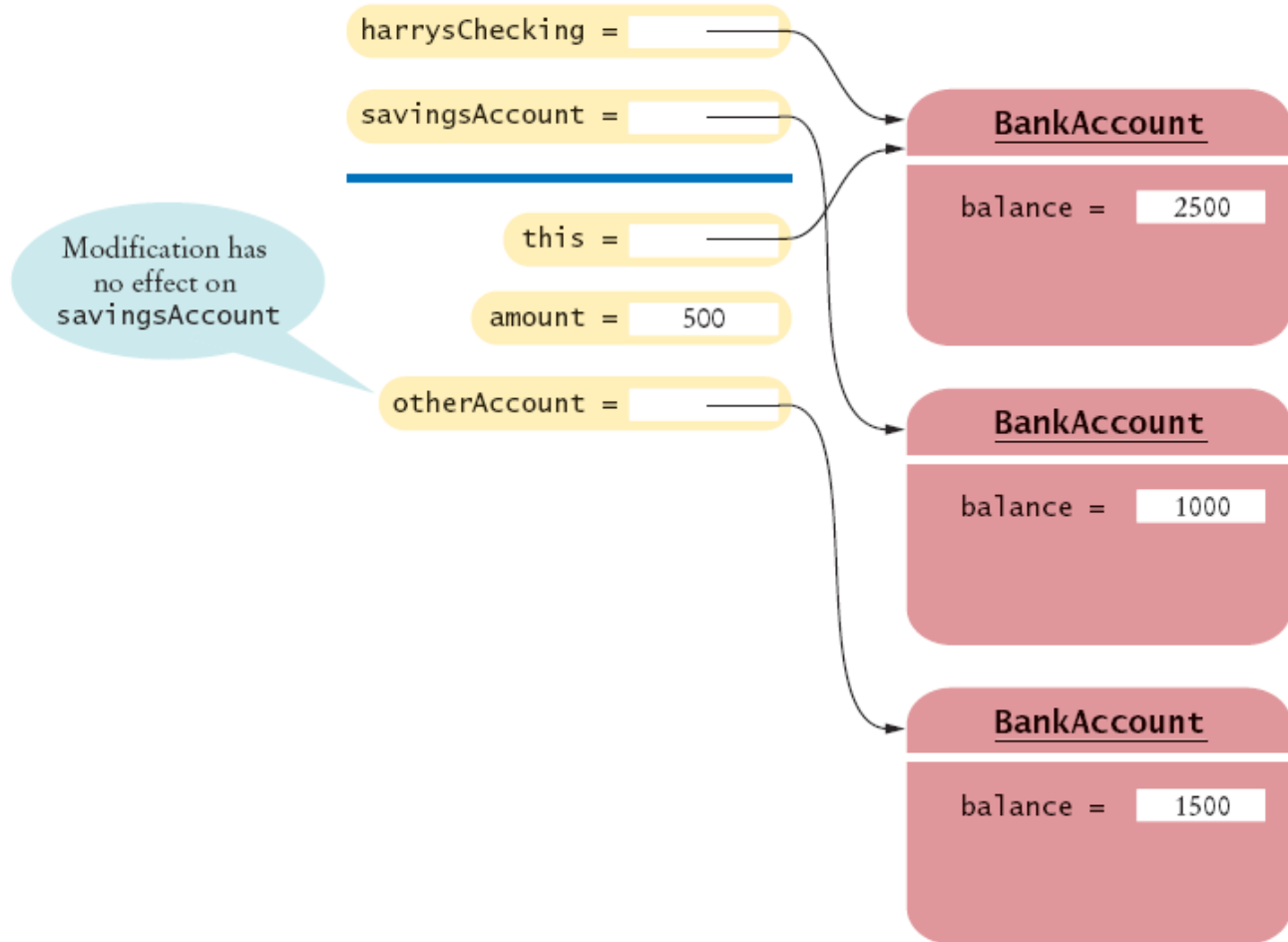
- Passage par valeur : Les paramètres sont copiés dans les variables représentant les paramètres au début de l'exécution de l'appel de méthode
- Passage par référence : Les méthodes peuvent modifier les paramètres
- En Java uniquement des passage par valeur
- Une méthode peut change l'état d'un objet passé en paramètre (référence passée en paramètre), mais ne peut remplacer la référence par une autre référence

## Passage par valeur et passage par référence /2

```
public class BankAccount
{
    public void transfer(double amount, BankAccount
        otherAccount)
    {
        balance = balance - amount;
        double newBalance = otherAccount.balance + amount;
        otherAccount = new BankAccount(newBalance); //
            Won't work
    }
}
```

# Exemple : Passage par valeur

```
harrysChecking.transfer(500, savingsAccount);
```



Modifying an Object Reference Parameter Has No Effect on the Caller

# Préconditions

---

- Précondition: Contrat que l'appelant d'une méthode doit respecter
- Documenter les préconditions pour que l'appelant n'appelle pas avec de mauvaises valeurs en paramètre
- ```
/**  
    Deposits money into this account.  
    @param amount the amount of money to deposit  
    (Precondition: amount >= 0)  
*/
```
- Usage courant:
  - *Pour restreindre les valeurs des paramètres d'une méthode*
  - *Pour s'assurer qu'une méthode n'est appelée que lorsque l'objet est dans un état particulière*
- Si la précondition est violée, la méthode ne garantit pas un résultat correct.

## Préconditions /2

---

- Méthode peuvent lever des exceptions si la préconditions est violée (cf. cours plus tard)  

```
if (amount < 0) throw new IllegalArgumentException();  
balance = balance + amount;
```
- Méthode n'a pas à tester si une condition est satisfaite (Test peut être coûteux)  

```
// if this makes the balance negative, it's the caller's  
    fault  
balance = balance + amount;
```

## Préconditions /3

---

- Méthode peut effectuer une vérification d'assertion

```
assert amount >= 0;  
balance = balance + amount;
```

- Pour activer la vérification des assertions :

```
java -enableassertions MyProg
```

On peut désactiver l'exécution des assertions quand notre programme est complètement testé, cela permet d'améliorer les performances d'exécution

- Tendance des programmeurs débutant (retour silencieux)

```
if (amount < 0)  
    return; // Non recommandé car difficile à déboguer  
balance = balance + amount;
```



# Syntaxe Assertion

```
assert condition;
```

## Exemple :

```
assert amount >= 0;
```

## Objectif :

S'assurer qu'une condition est bien remplie. Si la vérification des assertions est activée et que la condition est violée alors une erreur est levée lors de l'exécution.

# Postconditions

---

- Condition qui est satisfaite après l'exécution d'une méthode
- Si une méthode est appelée en respectant ses préconditions, elle doit garantir ses postconditions
- Il a deux types de postconditions :
  - *La valeur de retour est correctement calculée*
  - *L'objet est dans un certain état après l'exécution de la méthode*

/\*\*

```
Deposits money into this account.
```

```
(Postcondition: getBalance() >= 0)
```

```
@param amount the amount of money to deposit
```

```
(Precondition: amount >= 0) */
```

- Ne documenter pas les postconditions triviales qui répète la clause @return

## Postconditions /2

---

```
amount <= getBalance() // bonne formulation
```

```
amount <= balance // mauvaise formulation
```

- Contrat : Si l'appelant remplit la précondition, la méthode doit assurer la précondition

## Questions

---

Pourquoi souhaiteriez vous ajouter une précondition à une méthode que vous fournissez à un autre programmeur ?

**Réponse :** Ensuite, vous n'aurez plus à tester les mauvaises valeurs – cela devient la responsabilité de l'appel (l'autre développeur).

# Méthodes de classe

---

- Toute méthode appartient à une classe
- Une méthode de classe n'est pas invoquée sur un objet
- Pourquoi écrire une méthode qui n'opère pas sur objet ?  
Raison habituelle : des calculs sur des nombres qui ne sont pas des objets (ne peuvent donc pas recevoir d'appel de méthode).  
Exemple : `x.sqrt()` avec `x` de type `double`

- ```
public class Financial
{
    public static double percentOf(double p, double a)
    {
        return (p / 100) * a;
    }
    // More financial methods can be added here.
}
```

## Méthodes de classe /2

---

- Appeler avec le nom de la classe comme receveur au lieu d'une référence d'objet :  
double tax = **Financial**.percentOf(taxRate, total);
- main est une méthode de classe – il n'existe pas encore d'autres objets

## Variable de classe

---

- Une variable de classe appartient à une classe et non pas à un objet particulier de la classe.
- ```
public class BankAccount
{
    . . .
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber = 1000;
}
```
- Si `lastAssignedNumber` n'était pas `static`, chaque instance de `BankAccount` aurait sa propre valeur de `lastAssignedNumber`

## Variable de classe /2

---

- ```
public BankAccount()  
{  
    // Generates next account number to be assigned  
    lastAssignedNumber++; // Updates the static field  
    // Assigns field to account number of this bank  
    account  
    accountNumber = lastAssignedNumber; // Sets the  
    instance field }  
}
```
- Minimiser l'usage des variables de classe (final static ne sont pas concernés)



## Variable de classe /3

---

- 3 manières d'initialiser :
  1. *Ne rien faire. La variable est initialisée avec 0 (pour les nombres), false (pour les booléens) ou null (pour les références d'objet)*
  2. *Initialiser explicitement*

```
public class BankAccount
{
    . . .
    private static int lastAssignedNumber = 1000;
        // Executed once,
        // when class is loaded }

```
  3. *Utiliser un bloc de code static*
- Les variables de classe devraient toujours être déclarées comme `private`

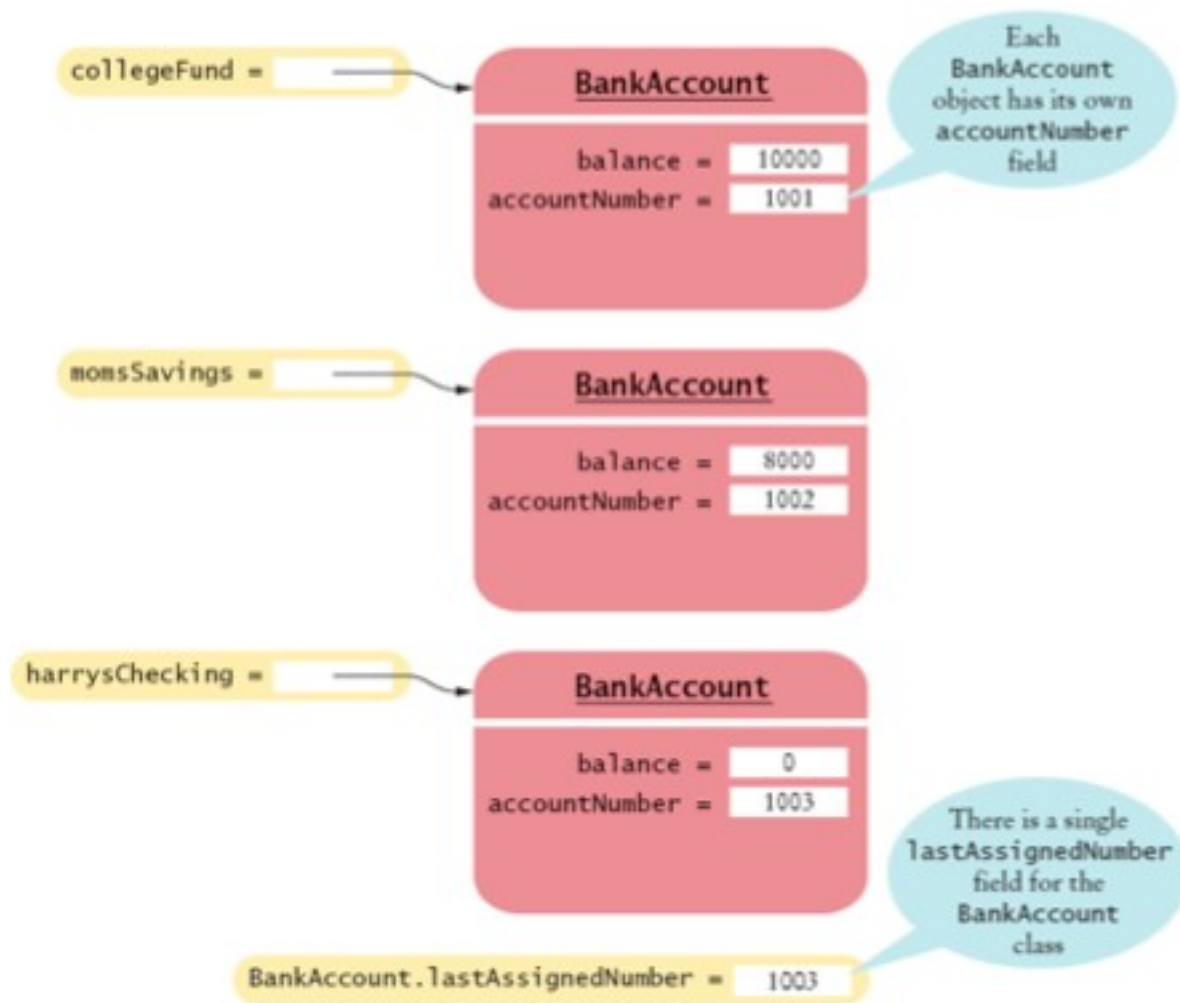
## Variable de classe /4

---

- Exception : les constantes qui peuvent être publique

```
public class BankAccount
{
    . . .
    public static final double OVERDRAFT_FEE = 5; //
        Refer to it as
        // BankAccount.OVERDRAFT_FEE
}
```

# Une variable de classe et une variable d'instance



# Portée des variables locales

---

- Portée des variables : Portion d'un programme où une variable peut être accédée
- Portée d'une variable locale : de sa position de déclaration à la fin du bloc englobant

## Portée des variables locales /2

- Le même nom de variable est utilisée dans plusieurs méthode :

```
public class RectangleTester
{
    public static double area(Rectangle rect)
    {
        double r = rect.getWidth() * rect.getHeight();
        return r;
    }
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(5, 10, 20, 30);
        double a = area(r);
        System.out.println(r);
    }
}
```

- Ces variables sont indépendantes. Leurs portées sont disjointes

## Portée des variables locales /3

- La portée d'une variable locale ne peut contenir la définition d'une variable avec le même nom

```
Rectangle r = new Rectangle(5, 10, 20, 30);  
if (x >= 0)  
{  
    double r = Math.sqrt(x);  
    // Error - can't declare another variable called r  
    here  
    . . .  
}
```

## Portée des variables locales /4

- Pourtant, deux variables locales peuvent avoir le même nom si leurs portées sont disjointes

```
if (x >= 0)
{
    double r = Math.sqrt(x);
    . . .
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK - it is legal to declare another r here
    . . .
}
```

## Portée des variables de classe

---

- Les variables privées ont la portée de la classe : Vous pouvez accéder toutes les variables dans toutes les méthodes de la classe
- On doit préciser le nom de la classe pour accéder aux variables publiques hors de la classe de définition  
`Math.sqrt`  
`harrysChecking.getBalance`
- Dans une méthode, il n'est pas nécessaire (mais fortement conseillé) de préciser le nom de la classe lors de l'appel d'une méthode ou d'une variable de la même classe



## Portée des variables de classe /2

- Un appel non qualifié (méthode ou instance) référence l'objet courant `this`

```
public class BankAccount
{
    public void transfer(double amount, BankAccount other)
    {
        withdraw(amount); // i.e., this.withdraw(amount);
        other.deposit(amount);
    }
    . . .
}
```

# Recouvrement des portées

- Une variable locale peut masquer une variable d'instance qui porte le même nom
- La portée locale gagne sur la portée de classe

```
public class Coin
{
    . . .
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        . . .
        return value;
    }
    private String name;
    private double value; // Field with the same name
}
```

## Recouvrement des portées /2

---

- On peut toujours accéder aux variables masquées en les qualifiant avec la référence `this`  
`value = this.value * exchangeRate;`

# Organisation des classes en paquetage

---

- Paquetage (*Package*): Ensemble de classe en relation
- Pour placer des classes dans un paquetage, il faut ajouter la ligne :  
`package packageName;`  
comme première instruction dans le code source de la classe
- Noms de paquetage sont formées par une succession d'identifiants séparés par des points

## Organisation des classes en paquetage /2

---

- Par exemple, pour placée la classe `Financial` dans un paquetage nommé `com.horstmann.bigjava`, le fichier `Financial.java` doit débuté par :

```
package com.horstmann.bigjava;
```

```
public class Financial  
{  
    . . .  
}
```

- Paquetage par défaut, aucune instruction `package`

# Paquetages couramment utilisés dans la librairie Java

---

<b>Paquetage</b>	<b>Objectif</b>	<b>Classe exemple</b>
java.lang	Fonctionnalité du langage	Math
java.util	Utilitaires	Random
java.io	Entrée / sortie	PrintStream
java.awt	Abstract Windowing Toolkit (interface graphique)	Color
java.applet	Applets	Applet
java.net	Réseau	Socket
java.sql	Accès base de données	ResultSet
javax.swing	Swing (interface graphique)	JButton

# Syntaxe Paquetage

---

```
package packageName;
```

## **Exemple :**

```
package com.horstmann.bigjava;
```

## **Objectif :**

Déclarer que toutes les classes de ce fichier appartiennent à un paquetage spécifique.

# Importer des paquetages

---

- On peut toujours utiliser une classe sans importer son paquetage  

```
java.util.Scanner in = new java.util.Scanner(System.in);
```
- Fatigant d'utiliser le nom qualifié
- Importation permet d'utiliser les noms courts des classes  

```
import java.util.Scanner; . . .  
Scanner in = new Scanner(System.in)
```
- On peut importer toutes les classes d'un paquetage  

```
import java.util.*;
```
- Jamais nécessaire d'importer `java.lang`
- Jamais nécessaire d'importer les autres classes d'un même paquetage



# Nom de paquetage et localisation des classes

---

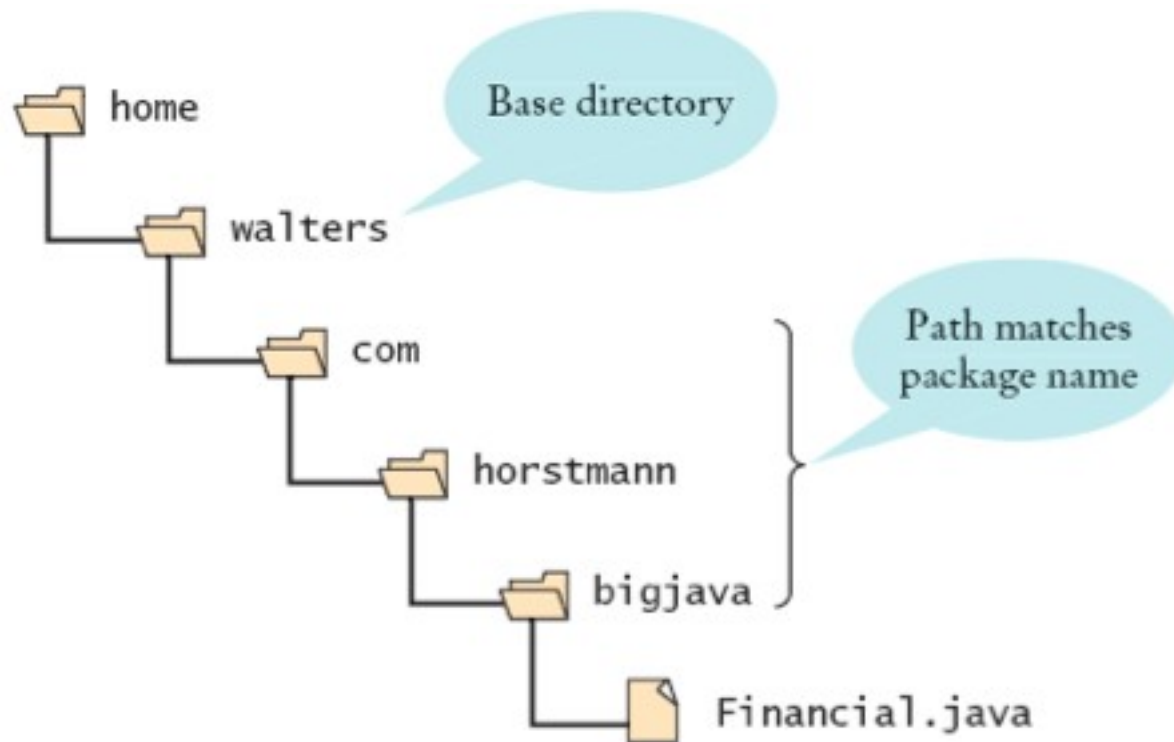
- Utiliser des noms de paquetages pour éviter les clash sur les noms  
`java.util.Timer` vs. `javax.swing.Timer`
- Noms de paquetages ne doivent pas être ambiguë
- Recommandation : débute par un nom de domaine inversé  
`com.horstmann.bigjava`  
`fr.esial.uhp`
- Les chemins doivent concorder avec les noms de paquetages  
`com/horstmann/bigjava/Financial.java`

## Nom de paquetage et localisation des classes /2

---

- Les chemins sont recherchés à partir du classpath  
export CLASSPATH=/home/walters/lib:  
set CLASSPATH=c:\home\walters\lib;
- Class path contient les répertoires de base où les répertoires des paquetages sont placés

# Répertoires de base et sous répertoires des paquetages



# Questions

---

Parmi les éléments suivants, lesquels sont des paquetages ?

- a. java
- b. java.lang
- c. java.util
- d. java.lang.Math

**Réponse :**

- a. No
- b. Yes
- c. Yes
- d. No