

# Programmation Orientée Objet

## 2ème 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)

## 1ère Partie : Types fondamentaux

### Objectifs de cette partie

---

- Comprendre les nombres entiers et les nombres à virgule flottante
- Connaître les limitations des types numériques
- Être conscient des causes des erreurs de dépassement et d'arrondi
- Utiliser correctement les constantes
- Ecrire des expressions arithmétiques en Java
- Utiliser le type `String` pour définir et manipuler des chaînes de caractères
- Apprendre à lire des données en entrée

## Types numériques

- `int`: valeurs entières, pas de partie décimale  
1, -4, 0
- `double`: nombre à virgule flottante (précision double)  
0.5, -3.11111, 4.3E24, 1E-14
- Un calcul numérique peut engendrer un dépassement (*overflow*) si son résultat sort de l'intervalle de définition du type numérique  

```
int n = 1000000;  
System.out.println(n * n); // prints -727379968
```
- En Java : 8 type primitifs dont 4 types entiers et 2 types flottants

## Types primitifs

Type	Description	Taille
<code>int</code>	Type entier, intervalle -2,147,483,648 . . . 2,147,483,647	4 octets
<code>byte</code>	Type décrivant un unique octet, intervalle -128 . . . 127	1 octet
<code>short</code>	Type entier "court", intervalle -32768 . . . 32767	2 octets
<code>long</code>	Type entier "long", intervalle -9,223,372,036,854,775,808 . . . -9,223,372,036,854,775,807	8 octets
<code>double</code>	Type réel (virgule flottante, double précision), intervalle approximatif $\pm 10^{308}$ et environ 15 décimales significatives	8 octets
<code>float</code>	Type réel (virgule flottante, simple précision), intervalle approximatif $\pm 10^{38}$ et environ 7 décimales significatives	4 octets
<code>char</code>	Type caractère représentant un code dans la table d'encodage Unicode	2 octets
<code>boolean</code>	Type booléen avec 2 valeurs de vérité false and true	1 bit

## Types numériques : Nombres à virgule flottante

- Des erreurs d'arrondi surviennent quand une conversion exacte vers un nombre n'est pas possible  

```
double f = 4.35;  
System.out.println(100 * f); // prints 434.99999999999994
```
- En Java: Il est interdit d'affecter à une variable entière une expression à virgule flottante  

```
double balance = 13.75;  
int dollars = balance; // Erreur
```
- Transtypage (*cast*): utiliser pour convertir une valeur d'un type à un autre  

```
int dollars = (int) balance; // OK
```

Le transtypage supprime la partie décimale

## Types numériques : Nombres à virgule flottante /2

`Math.round` convertit un nombre à virgule flottante vers le nombre entier le plus proche

```
long rounded = Math.round(balance); // if balance is 13.75, then  
// rounded is set to 14
```

## Syntaxe Transtypage

```
(typeName) expression
```

Exemple :

```
(int) (balance * 100)
```

Objectif :

Convertir une expression d'un type vers un autre type.

## Constantes: mot-clé final

- Une variable déclarée `final` est une constante
- Une fois sa valeur affectée, elle ne peut être modifiée
- Les constantes nommées rendent les programmes plus facile à lire et à maintenir
- Convention: nom des constantes entièrement en majuscule

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;
payment = dollars + quarters * QUARTER_VALUE
        + dimes * DIME_VALUE + nickels * NICKEL_VALUE
        + pennies * PENNY_VALUE;
```

## Constantes: mots-clés static final

- Si une valeur constante est utilisée dans plusieurs méthodes, déclarer celle-ci avec les variables d'instance en ajoutant les mots clés `static` et `final`
- Donner un accès public aux constantes `static final` pour utiliser celles-ci hors de la classe qui les déclare

```
public class Math
{
    . . .
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
}

double circumference = Math.PI * diameter;
```

## Syntaxe Définition d'une constante

Dans une méthode :

```
final typeName variableName = expression;
```

Dans une classe :

```
accessSpecifier static final typeName variableName =
    expression;
```

Exemple :

```
final double NICKEL_VALUE = 0.05; public static final
double LITERS_PER_GALLON = 3.785;
```

Objectif :

Pour définir une constante dans une méthode ou une classe.

## ch04/cashregister/CashRegister.java

```
01: /**
02:  * A cash register totals up sales and computes change due.
03:  */
04: public class CashRegister
05: {
06:     /**
07:      * Constructs a cash register with no money in it.
08:      */
09:     public CashRegister()
10:     {
11:         purchase = 0;
12:         payment = 0;
13:     }
14:
15:     /**
16:      * Records the purchase price of an item.
17:      * @param amount the price of the purchased item
18:      */
19:     public void recordPurchase(double amount)
20:     {
21:         purchase = purchase + amount;
22:     }
```

## ch04/cashregister/CashRegister.java /2

```
23:
24:     /**
25:      * Enters the payment received from the customer.
26:      * @param dollars the number of dollars in the payment
27:      * @param quarters the number of quarters in the payment
28:      * @param dimes the number of dimes in the payment
29:      * @param nickels the number of nickels in the payment
30:      * @param pennies the number of pennies in the payment
31:      */
32:     public void enterPayment(int dollars, int quarters,
33:                             int dimes, int nickels, int pennies)
34:     {
35:         payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
36:             + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
37:     }
38:
39:     /**
40:      * Computes the change due and resets the machine for the next
41:      * customer.
42:      * @return the change due to the customer
43:      */
44:     public double giveChange()
45:     {
```

## ch04/cashregister/CashRegister.java /3

```
45:         double change = payment - purchase;
46:         purchase = 0;
47:         payment = 0;
48:         return change;
49:     }
50:
51:     public static final double QUARTER_VALUE = 0.25;
52:     public static final double DIME_VALUE = 0.1;
53:     public static final double NICKEL_VALUE = 0.05;
54:     public static final double PENNY_VALUE = 0.01;
55:
56:     private double purchase;
57:     private double payment;
58: }
```

## ch04/cashregister/CashRegisterTester.java

```
01: /**
02:  * This class tests the CashRegister class.
03:  */
04: public class CashRegisterTester
05: {
06:     public static void main(String[] args)
07:     {
08:         CashRegister register = new CashRegister();
09:
10:         register.recordPurchase(0.75);
11:         register.recordPurchase(1.50);
12:         register.enterPayment(2, 0, 5, 0, 0);
13:         System.out.print("Change: ");
14:         System.out.println(register.giveChange());
15:         System.out.println("Expected: 0.25");
16:
17:         register.recordPurchase(2.25);
18:         register.recordPurchase(19.25);
19:         register.enterPayment(23, 2, 0, 0, 0);
20:         System.out.print("Change: ");
21:         System.out.println(register.giveChange());
22:         System.out.println("Expected: 2.0");
23:     }
24: }
```

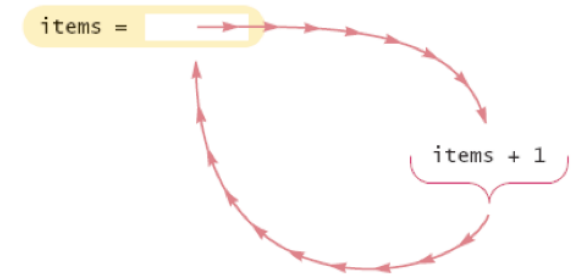
## ch04/cashregister/CashRegisterTester.java /2

### Output:

Change: 0.25  
Expected: 0.25  
Change: 2.0  
Expected: 2.0

## Affectation, Incrémement, Décrémentation

- Affectation est différente de l'égalité mathématique :  
`items = items + 1;`
- `items++` est la même chose que `items = items + 1`
- `items-` soustrait 1 de `items`



## Opérateurs Arithmétiques

- `/` est l'opérateur de division
- Si les deux arguments sont des entiers, le résultat est un entier. Le reste est supprimé (division entière)
- `7.0 / 4` donne 1.75  
`7 / 4` donne 1
- Le reste de la division entière est obtenu avec `%` ("modulo")  
`7 % 4` donne 3

## Opérateurs Arithmétiques /2

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;

// Compute total value in pennies
int total = dollars * PENNIES_PER_DOLLAR + quarters *
    PENNIES_PER_QUARTER + nickels * PENNIES_PER_NICKEL +
    dimes * PENNIES_PER_DIME + pennies;
// Use integer division to convert to dollars, cents
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```

## La classe `Math`

- Classe `Math` contient des méthodes telles que `sqrt` et `pow`
- Pour calculer  $x^n$ , on écrit `Math.pow(x, n)`
- Pourtant pour calculer  $x^2$  il est plus simple (et efficace) de calculer `x * x`
- Pour calculer la racine carrée d'un nombre, utilise `Math.sqrt(x)`
- En Java,  $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

peut s'exprimer par

`(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)`

## Quelques méthodes mathématiques

Fonction	Résultat
<code>Math.sqrt(x)</code>	Racine carrée
<code>Math.pow(x, y)</code>	Puissance $x^y$
<code>Math.exp(x)</code>	Exponentielle $e^x$
<code>Math.log(x)</code>	Logarithme naturel (népérien)
<code>Math.sin(x)</code> , <code>Math.cos(x)</code> , <code>Math.tan(x)</code>	sinus, cosinus, tangente ( $x$ en radians)
<code>Math.round(x)</code>	Valeur entière la plus proche de $x$
<code>Math.min(x, y)</code> , <code>Math.max(x, y)</code>	minimum, maximum

## Analyser une expression

$$\begin{array}{c} (-b + \text{Math.sqrt}(b * b - 4 * a * c)) / (2 * a) \\ \underbrace{\quad \underbrace{b^2} \quad \underbrace{4ac} \quad \underbrace{2a}}_{b^2 - 4ac} \\ \underbrace{\quad \underbrace{\sqrt{b^2 - 4ac}}}_{-b + \sqrt{b^2 - 4ac}} \\ \underbrace{\quad \underbrace{\frac{-b + \sqrt{b^2 - 4ac}}{2a}} \end{array}$$

## Appel de méthode de classe (static)

- Une méthode déclarée `static` n'opère pas sur un objet de type `double x = 4;`  
`double root = x.sqrt(); // Error`
- Méthodes statiques sont déclarées dans des classes
- Convention de nommage : Nom de classe débute par une lettre majuscule; le nom d'une référence vers un objet débute par une minuscule  
`Math`  
`System.out`

## Syntaxe Appel de méthode de classe (statique)

```
ClassName.methodName(parameters)
```

Exzemple :

```
Math.sqrt(4)
```

Objectif :

Invoquer une méthode de classe (qui n'opère donc pas sur une instance/un objet) et lui passer des valeurs en paramètre.

## Chaînes de caractères

- `String` représente une séquence de caractères
- Ce sont des objets de la classe `String`
- Chaînes constantes :  
`"Hello, World!"`
- Chaînes "variables" :  
`String message = "Hello, World!";`
- Longueur d'une chaîne de caractères :  
`int n = message.length();`
- Chaîne vide : `""`

## Concaténation

- Utiliser l'opérateur `+` :

```
String name = "Dave";  
String message = "Hello, " + name; // message is "Hello,  
Dave"
```

- Si une des opérandes de l'opérateur `+` est de type `String`, l'autre opérande est convertie automatiquement en une chaîne de caractères

```
String a = "Agent"; int n = 7; String bond = a + n; //  
bond is "Agent7"
```

## Concaténation dans une expression d'affichage

- Il est utile d'utiliser la concaténation pour réduire le nombre d'instructions.

- Par exemple :

```
System.out.print("The total is ");  
System.out.println(total);
```

ce qui est équivalent à :

```
System.out.println("The total is " + total);
```

## Conversion entre chaînes de caractères et valeurs numériques

- Conversion vers un nombre :

```
int n = Integer.parseInt(str);  
double x = Double.parseDouble(x);
```

- Conversion vers une chaîne de caractères :

```
String str = "" + n;  
str = Integer.toString(n);
```

## Sous chaîne

- `String greeting = "Hello, World!";`  
`String sub = greeting.substring(0, 5);` // sub is "Hello"

- Indiquer le début et la longueur de la chaîne à extraire

- Première position est 0

```
H e l l o , W o r l d !  
0 1 2 3 4 5 6 7 8 9 10 11 12
```

```
           5  
           ┌───┐  
H e l l o , W o r l d !  
0 1 2 3 4 5 6 7 8 9 10 11 12
```

## Alphabets internationaux



A German Keyboard

## Alphabets internationaux /2

	จ	ฐ	ฑ	ฒ	ศ	ะ	ฎ	เ	็	อ	๗		เ
ก	ฅ	ท	น	ษ	ั	ญ	แ	็	๑	๔		๒	
ข	ฌ	บ	ย	ล	า	ุ	โ	็	๒	๗		โ	
ช	ฌ	ป	ร	ห	า		ใ	็	๓	๗		ใ	
ค	ฅ	ด	ฝ	ภ	พิ	ิ	ไ	็	๔			ไ	
ค	ฅ	ต	ฝ	ล	อ	็	ใ	็	๕				
พ	ฅ	ถ	พ	ภ	พิ	็	ใ	็	๕	๖			
ง	ฅ	ท	พ	ว	๗	็	็		๗				

The Thai Alphabet



## Alphabets internationaux /2

		Sm.	Lg.
<b>CLASSIC SOUPS</b>			
清 燉 雞 湯	57.	House Chicken Soup (Chicken, Celery, Potato, Onion, Carrot) .....	1.50 2.75
雞 飯 湯	58.	Chicken Rice Soup .....	1.85 3.25
雞 麵 湯	59.	Chicken Noodle Soup .....	1.85 3.25
廣 東 雲 吞	60.	Cantonese Wonton Soup.....	1.50 2.75
蕃 茄 蛋 湯	61.	Tomato Clear Egg Drop Soup .....	1.65 2.95
雲 吞 湯	62.	Regular Wonton Soup .....	1.10 2.10
酸 辣 湯	63.	Hot & Sour Soup .....	1.10 2.10
蛋 花 湯	64.	Egg Drop Soup.....	1.10 2.10
雲 吞 湯	65.	Egg Drop Wonton Mix.....	1.10 2.10
豆 腐 菜 湯	66.	Tofu Vegetable Soup .....	NA 3.50
雞 玉 米 湯	67.	Chicken Corn Cream Soup .....	NA 3.50
蟹 肉 玉 米 湯	68.	Crab Meat Corn Cream Soup.....	NA 3.50
海 鮮 湯	69.	Seafood Soup.....	NA 3.50

A Menu with Chinese Characters

## Lecture depuis l'entrée standard

- System.in offre que des fonctionnalités limitées - lecture d'un octet à la fois
- Depuis Java 5.0, la classe Scanner est fournie et permet de lire de manière simple une valeur saisie depuis le clavier
- `Scanner in = new Scanner(System.in);`  
`System.out.print("Enter quantity:");`  
`int quantity = in.nextInt();`
- `nextDouble` lit un double
- `nextLine` lit une ligne (jusqu'à un retour charriot)
- `nextWord` lit un mot (jusqu'à un espace)

## ch04/cashregister/CashRegisterSimulator.java

```
01: import java.util.Scanner;
02:
03: /**
04:  This program simulates a transaction in which a user pays for an
05:  item
06:  and receives change.
07: */
08: public class CashRegisterSimulator
09: {
10:     public static void main(String[] args)
11:     {
12:         Scanner in = new Scanner(System.in);
13:
14:         CashRegister register = new CashRegister();
15:
16:         System.out.print("Enter price: ");
17:         double price = in.nextDouble();
18:         register.recordPurchase(price);
19:
20:         System.out.print("Enter dollars: ");
21:         int dollars = in.nextInt();
```

...

## ch04/cashregister/CashRegisterSimulator.java /2

### Output:

```
Enter price: 7.55
Enter dollars: 10
Enter quarters: 2
Enter dimes: 1
Enter nickels: 0
Enter pennies: 0
Your change: is 3.05
```

## 2<sup>ème</sup> Partie : Décisions

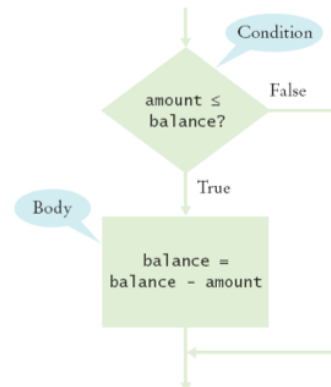
### Objectifs de cette partie

- Implémenter des décisions en utilisant l'instruction `if`
- Comprendre comment regrouper les instructions dans des blocs
- Apprendre à comparer des valeurs entières, des nombres à virgule flottante, des chaînes de caractères et des objets
- Déterminer l'ordre d'exécution des instructions dans des décisions à branches multiples
- Programmer des conditions en utilisant des opérateurs booléens et des variables
- Comprendre l'importance de la couverture des tests

### L'instruction `if`

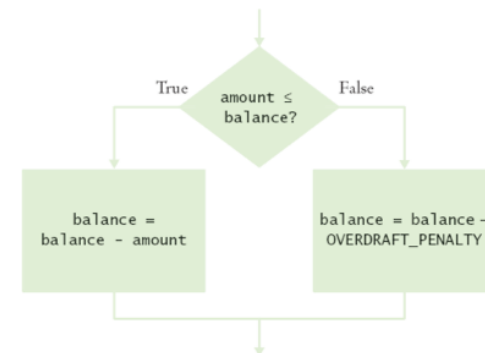
- L'instruction `if` permet à un programme d'exécuter des traitements différents selon une condition

```
If (amount <= balance)
    balance = balance - amount;
```



### Les instructions `if/else`

```
If (amount <= balance)
    balance = balance - amount;
else
    balance = balance - OVERDRAFT_PENALTY
```



## Différents types d'instructions

- Instruction simple

```
balance = balance - amount;
```

- Instruction composée

```
if (balance >= amount) balance = balance - amount;
```

et

while, for, etc. (voir partie suivante)

- Bloc d'instructions

```
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

## Syntaxe L'instruction if

```
if (condition)
    statement
if (condition)
    statement1
else
```

### Exemple :

```
if (amount <= balance)
    balance = balance - amount;
if (amount <= balance)
    balance = balance - amount;
else
```

### Objectif :

Exécuter une instruction lorsqu'une condition est vraie ou fausse.

## Syntaxe Bloc d'instructions

```
{
    statement1
    statement2
    . . .
}
```

### Exemple :

```
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

### Objectif :

Regrouper plusieurs instructions pour former une instructions.

## Comparaison de valeurs : Opérateurs relationnels

- Opérateurs relationnels compare des valeurs

Java	Notation	Description
>	>	Supérieur à
>=	≥	Supérieur ou égal à
<	<	Inférieur à
<=	≤	Inférieur ou égal à
==	=	Egalité
!=	≠	Différence

- L'opérateur == dénote le test d'égalité

```
a = 5; // Affecter 5 à la variable a
if (a == 5) . . . // Test si a est égal à 5
```

## Comparaison de nombres à virgule flottante

- Le code suivant :

```
double r = Math.sqrt(2);
double d = r * r -2;
if (d == 0)
    System.out.println("sqrt(2)squared minus 2 is 0");
else
    System.out.println("sqrt(2)squared minus 2 is not 0
        but " + d);
```

- Affiche :

```
sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16
```

## Comparaison de nombres à virgule flottante /2

- Afin d'éviter les erreurs dues au arrondis, n'utilisez pas `==` pour comparer des nombres réels.
- Pour comparer des valeurs réelles, testez si elles sont suffisamment *proches* :

```
|x - y| ≤ ε
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
    // x est approximativement égal à y
```

- $\epsilon$  est un nombre infiniment petit tel que  $10^{-14}$

## Comparaison de chaînes de caractères

- N'utilisez pas `==` pour des chaînes de caractères

```
if (input == "Y") // FAUX!!!
```

- Utilisez la méthode `equals` :

```
if (input.equals("Y"))
```

- `==` teste l'identité, `equals` teste l'égalité de contenu

- Pour tester de manière non sensible à la case ("Y" ou "y")

```
if (input.equalsIgnoreCase("Y"))
```

## Comparaison de chaînes de caractères /2

- `s.compareTo(t) < 0` signifie :  
`s` précède `t` dans le dictionnaire

- "car" précède "cargo"

- Les majuscules précèdent les minuscules  
"Hello" précède "car"

- Ordre lexicographique

c a r g o

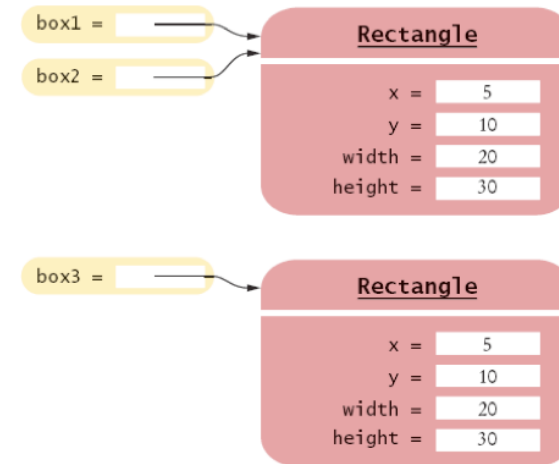
c a t h o d e

Letters r comes  
match before t

## Comparaison d'objets

- `==` teste l'identité, `equals` pour des contenus identiques
- ```
Rectangle box1 = new Rectangle(5, 10, 20, 30);  
Rectangle box2 = box1;
```
- ```
Rectangle box3 = new Rectangle(5, 10, 20, 30);  
box1 != box3,
```
- mais `box1.equals(box3)`  
`box1 == box2`
- Remarque : `equals` doit être définie dans la classe

## Comparaison d'objets (de références d'objets) /2



## Tester la référence null

- La référence `null` ne référence aucun objet  

```
String middleInitial = null; // Not set  
if ( . . . )  
    middleInitial = middleName.substring(0, 1);
```
- Elle peut être utilisée dans les tests :  

```
if (middleInitial == null)  
    System.out.println(firstName + " " + lastName);  
else  
    System.out.println(firstName + " " + middleInitial +  
        ". " + lastName);
```
- Utiliser `==` et non pas `equals` pour tester par rapport à `null`
- `null` n'est pas la même chose que la chaîne vide `""`

## Alternatives multiples : Séquences de comparaisons

```
if (condition1)  
    statement1;  
else if (condition2)  
    statement2;  
    . . .  
else  
    statement4;
```

- La première condition satisfaite déclenche son exécution
- L'ordre a son importance

```
if (richter >= 0) // toujours testé  
    r = "Generally not felt by people";  
else if (richter >= 3.5) // jamais testé  
    r = "Felt by many people, no destruction";  
    . . .
```

## Alternatives multiples : Séquences de comparaisons /2

- N'omettez pas l'instruction `else`

```
if (richter >= 8.0)
    r = "Most structures fall";
if (richter >= 7.0) // else omis -- ERREUR
    r = "Many buildings destroyed
```

## ch05/quake/Earthquake.java

```
01: /**
02:  * A class that describes the effects of an earthquake.
03:  */
04: public class Earthquake
05: {
06:     /**
07:      * Constructs an Earthquake object.
08:      * @param magnitude the magnitude on the Richter scale
09:      */
10:     public Earthquake(double magnitude)
11:     {
12:         richter = magnitude;
13:     }
14:
15:     /**
16:      * Gets a description of the effect of the earthquake.
17:      * @return the description of the effect
18:      */
19:     public String getDescription()
20:     {
```

## ch05/quake/Earthquake.java /2

```
21:     String r;
22:     if (richter >= 8.0)
23:         r = "Most structures fall";
24:     else if (richter >= 7.0)
25:         r = "Many buildings destroyed";
26:     else if (richter >= 6.0)
27:         r = "Many buildings considerably damaged, some collapse";
28:     else if (richter >= 4.5)
29:         r = "Damage to poorly constructed buildings";
30:     else if (richter >= 3.5)
31:         r = "Felt by many people, no destruction";
32:     else if (richter >= 0)
33:         r = "Generally not felt by people";
34:     else
35:         r = "Negative numbers are not valid";
36:     return r;
37: }
38:
39: private double richter;
40: }
```

## ch05/quake/EarthquakeRunner.java

```
01: import java.util.Scanner;
02:
03: /**
04:  * This program prints a description of an earthquake of a given
05:  * magnitude.
06:  */
07: public class EarthquakeRunner
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:
13:         System.out.print("Enter a magnitude on the Richter scale: ");
14:         double magnitude = in.nextDouble();
15:         Earthquake quake = new Earthquake(magnitude);
16:         System.out.println(quake.getDescription());
17:     }
```

### Output:

Enter a magnitude on the Richter scale: 7.1 Many buildings destroyed

## Alternatives multiples : branches imbriquées

- Branche imbriquée dans une autre branche

```
if (condition1)
{
    if (condition1a)
        statement1a;
    else
        statement1b;
}
else
    statement2;
```

## Exemple : Déclaration d'impôts

If your filing status is Single		If your filing status is Married	
Tax Bracket	Percentage	Tax Bracket	Percentage
\$0 ... \$21,450	15%	0 ... \$35,800	15%
Amount over \$21,450, up to \$51,900	28%	Amount over \$35,800, up to \$86,500	28%
Amount over \$51,900	31%	Amount over \$86,500	31%

## Branches imbriquées

- Calcul du niveau d'imposition en fonction d'un status et du niveau de revenu  
(1) condition sur le status, (2) pour chaque status condition sur le niveau de revenu
- Ce processus de décision à 2 niveaux se reflète dans les 2 niveaux d'imbrications des instructions `if`
- Le test sur le niveau de revenu est *imbriqué* dans le test sur le status

## Branches imbriquées /2

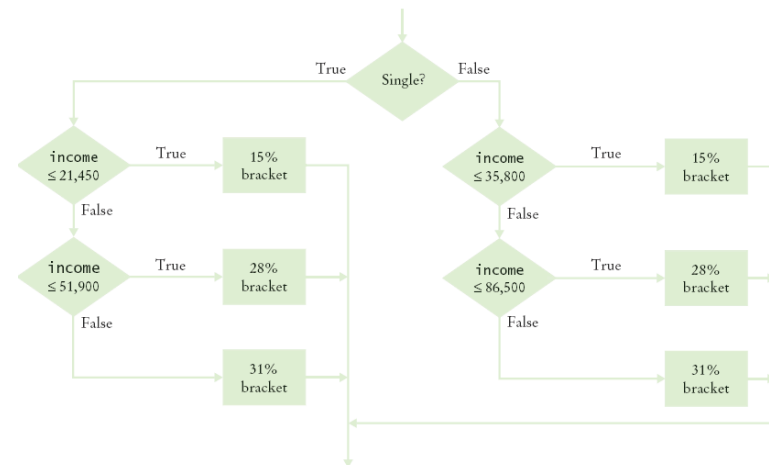


Figure 5 Income Tax Computation Using 1992 Schedule

## ch05/tax/TaxReturn.java

```
01: /**
02:  * A tax return of a taxpayer in 1992.
03:  */
04: public class TaxReturn
05: {
06:     /**
07:      * Constructs a TaxReturn object for a given income and
08:      * marital status.
09:      * @param anIncome the taxpayer income
10:      * @param aStatus either SINGLE or MARRIED
11:      */
12:     public TaxReturn(double anIncome, int aStatus)
13:     {
14:         income = anIncome;
15:         status = aStatus;
16:     }
17:
18:     public double getTax()
19:     {
20:         double tax = 0;
21:
22:         if (status == SINGLE)
23:         {
```

## ch05/tax/TaxReturn.java /2

```
24:             if (income <= SINGLE_BRACKET1)
25:                 tax = RATE1 * income;
26:             else if (income <= SINGLE_BRACKET2)
27:                 tax = RATE1 * SINGLE_BRACKET1
28:                     + RATE2 * (income - SINGLE_BRACKET1);
29:             else
30:                 tax = RATE1 * SINGLE_BRACKET1
31:                     + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1)
32:                     + RATE3 * (income - SINGLE_BRACKET2);
33:         }
34:     else
35:     {
36:         if (income <= MARRIED_BRACKET1)
37:             tax = RATE1 * income;
38:         else if (income <= MARRIED_BRACKET2)
39:             tax = RATE1 * MARRIED_BRACKET1
40:                 + RATE2 * (income - MARRIED_BRACKET1);
41:         else
42:             tax = RATE1 * MARRIED_BRACKET1
43:                 + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1)
44:                 + RATE3 * (income - MARRIED_BRACKET2);
45:     }
46: }
```

## ch05/tax/TaxReturn.java /3

```
47:         return tax;
48:     }
49:
50:     public static final int SINGLE = 1;
51:     public static final int MARRIED = 2;
52:
53:     private static final double RATE1 = 0.15;
54:     private static final double RATE2 = 0.28;
55:     private static final double RATE3 = 0.31;
56:
57:     private static final double SINGLE_BRACKET1 = 21450;
58:     private static final double SINGLE_BRACKET2 = 51900;
59:
60:     private static final double MARRIED_BRACKET1 = 35800;
61:     private static final double MARRIED_BRACKET2 = 86500;
62:
63:     private double income;
64:     private int status;
65: }
```

## ch05/tax/TaxCalculator.java

```
01: import java.util.Scanner;
02:
03: /**
04:  * This program calculates a simple tax return.
05:  */
06: public class TaxCalculator
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         System.out.print("Please enter your income: ");
13:         double income = in.nextDouble();
14:
15:         System.out.print("Are you married? (Y/N) ");
16:         String input = in.next();
17:         int status;
18:         if (input.equalsIgnoreCase("Y"))
19:             status = TaxReturn.MARRIED;
20:         else
21:             status = TaxReturn.SINGLE;
22:     }
```



## ch05/tax/TaxCalculator.java /2

```
23:     TaxReturn aTaxReturn = new TaxReturn(income, status);
24:
25:     System.out.println("Tax: "
26:         + aTaxReturn.getTax());
27: }
28: }
```

### Output:

```
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
```

## Utilisation des expressions booléennes : le type `boolean`



George Boole (1815-1864): logicien, mathématicien, et philosophe. Créateur d'une logique dite la logique de Boole

- valeur d'une expression `amount < 1000` est `true` OU `false`.
- type `boolean` : une de ces deux valeurs de vérité

## Utilisation des expressions booléennes : Prédicat

- Un prédicat est une méthode retournant une valeur booléenne

```
public boolean isOverdrawn()
{
    return balance < 0;
}
```

- Peut être utilisé dans des expressions conditionnelles

```
if (harrysChecking.isOverdrawn())
```

- Exemple : prédicats de la classe `Character` :  
`isDigit`, `isLetter`, `isUpperCase`, `isLowerCase`

- `if (Character.isUpperCase(ch)) ...`

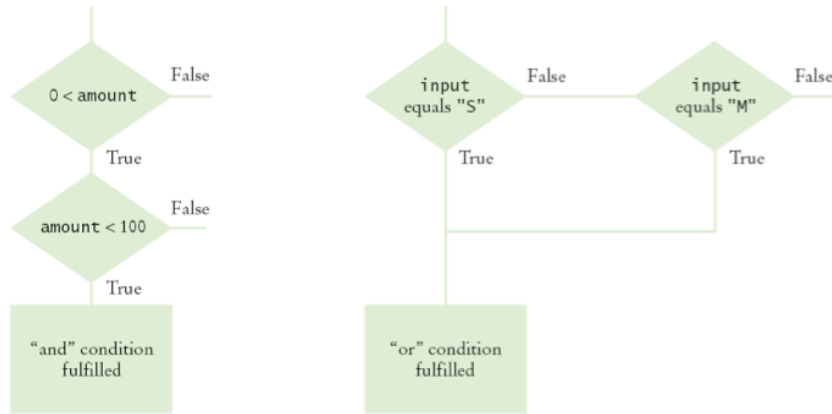
- prédicats de la classe `Scanner` :

```
hasNextInt() et hasNextDouble()
if (in.hasNextInt()) n = in.nextInt();
```

## Utilisation des expressions booléennes : Opérateurs

- `&&` et
- `||` ou
- `!` négation
- `if (0 < amount && amount < 1000) ...`
- `if (input.equals("S") || input.equals("M")) ...`

## Opérateurs && et ||



## Tables de vérité

A	B	A && B
true	true	true
true	false	false
false	Any	false

A	B	A    B
true	Any	true
false	true	true
false	false	false

A	! A
true	false
false	true

## Utilisation de variables booléennes

- `private boolean married;`
- Affecte la valeur de vérité dans une variable :  
`married = input.equals("M");`
- Utilisation dans des conditionnelles :  
`if (married) . . . else . . . if (!married) . . .`
- Parfois dénommé drapeau (*flag*)
- Il est inutile d'écrire  
`if (married == true) . . .`
- Il suffit simplement d'écrire  
`if (married) . . .`

## Couverture des tests

- Black-box testing: tester les fonctionnalités sans prendre en considération la structure interne de l'implémentation
- White-box testing: considérer l'implémentation de la structure interne lors de la conception des tests
- Test coverage: mesure combien de parties du programme sont testées
- Assurez vous que chaque partie de vos programmes sont testées au moins une fois par un test  
i.e. Chaque branche d'une conditionnelle doit être testée
- Boundary test cases: tests aux limites, valeurs qui sont aux limites des intervalles des valeurs acceptables
- Conseil : Ecrivez les tests avant d'écrire votre programme → donne une idée de ce que le programme doit faire

## 3<sup>ème</sup> Partie : Itérations

### Objectifs de cette partie

---

- Savoir écrire des programmes comportant des boucles en utilisant les instructions `while`, `for`, et `do`
- Eviter les boucles infinies et les erreurs de dépassement de pas
- Comprendre le fonctionnement des boucles imbriquées

### Boucles `while`

---

- Exécute un bloc de code de manière répétitive
- Une condition contrôle jusque quand la boucle doit s'exécuter

```
while (condition)
    statement
```

- Généralement, l'instruction `statement` est un bloc d'instructions (délimitées par `{ }`)

### Calcul l'accroissement d'un investissement

---

- Investissement de \$10,000, 5% d'intérêt chaque année

Année	Balance
0	\$10,000
1	\$10,500
2	\$11,025
3	\$11,576.25
4	\$12,155.06
5	\$12,762.82

## Calcul l'accroissement d'un investissement /2

- Quand un compte bancaire a-t-il atteint une somme précise ?

```
• while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

## ch06/invest1/Investment.java

```
01: /**
02:  A class to monitor the growth of an investment that
03:  accumulates interest at a fixed annual rate.
04: */
05: public class Investment
06: {
07:     /**
08:      Constructs an Investment object from a starting balance and
09:      interest rate.
10:      @param aBalance the starting balance
11:      @param aRate the interest rate in percent
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
19:
20:     /**
21:      Keeps accumulating interest until a target balance has
22:      been reached.
23:      @param targetBalance the desired balance
24:     */
```

## ch06/invest1/Investment.java /2

```
25:     public void waitForBalance(double targetBalance)
26:     {
27:         while (balance < targetBalance)
28:         {
29:             years++;
30:             double interest = balance * rate / 100;
31:             balance = balance + interest;
32:         }
33:     }
34:
35:     /**
36:      Gets the current investment balance.
37:      @return the current balance
38:     */
39:     public double getBalance()
40:     {
41:         return balance;
42:     }
43:
44:     /**
45:      Gets the number of years this investment has accumulated
46:      interest.
```

## ch06/invest1/Investment.java /3

```
47:         @return the number of years since the start of the investment
48:     */
49:     public int getYears()
50:     {
51:         return years;
52:     }
53:
54:     private double balance;
55:     private double rate;
56:     private int years;
57: }
```

## ch06/invest1/InvestmentRunner.java

```
01: /**
02:  This program computes how long it takes for an investment
03:  to double.
04:  */
05: public class InvestmentRunner
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         Investment invest = new Investment(INITIAL_BALANCE, RATE);
12:         invest.waitForBalance(2 * INITIAL_BALANCE);
13:         int years = invest.getYears();
14:         System.out.println("The investment doubled after "
15:             + years + " years");
16:     }
17: }
```

## ch06/invest1/InvestmentRunner.java /2

### Output:

The investment doubled after 15 years

## Animation 6.1

Program Code	Local variables				
	<table border="1"><thead><tr><th>i</th><th>s</th></tr></thead><tbody><tr><td></td><td></td></tr></tbody></table>	i	s		
i	s				

This animation demonstrates the process of hand tracing a loop. When you trace a loop, you keep track of the current line of code and the current values of the variables. Whenever a variable's value changes, you cross out the old value and write in the new value. Click on the "Next" button to see the next tracing step. It is a good idea for you to predict what action will occur *before* hitting the button. Keep clicking the "Next" button until the loop exits.

6-01 Tracing a Loop

## Flowchart While

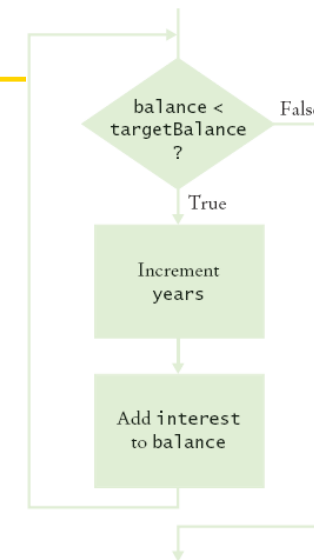


Figure 1 Flowchart of a while Loop

## Syntaxe Instruction `while`

```
while (condition)
    statement
```

### Exemple:

```
while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

### Objectif :

Exécuter de manière répétitive (tant que la condition est vraie) une séquence d'instructions.

## Erreur courante : boucle infinie

- ```
int years = 0;
while (years < 20)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```
- ```
int years = 20;
while (years > 0)
{
    years++; // Oops, should have been years-
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```
- La boucle s'exécute infiniment – le programme doit être tué

## Erreur courante : dépassement limite

```
• int years = 0;
  while (balance < 2 * initialBalance)
  {
      years++;
      double interest = balance * rate / 100;
      balance = balance + interest;
  }
  System.out.println("The investment reached the target
  after " + years + " years.");
```

Est-ce que `years` doit débiter à 0 ou à 1 ?

Est-ce que le test doit être `<` ou `<=` ?

## Eviter ce type d'erreur

- Considérer le scénario avec des valeurs simples :  
initialisation `balance: $100`  
intérêt `rate: 50%`  
après 1 an, `balance` est égale à \$150  
après 2 ans, elle est égale à \$225, soit plus de \$200  
donc l'investissement a doublé en 2 ans  
La boucle s'est exécutée 2 fois, incrémentant `years` chaque fois  
*Donc* : `years` doit être initialisé à 0, et non pas 1.
- intérêt: 100%  
après 1 an : `balance` est égale à `2 * initialBalance`  
la boucle doit s'arrêter  
*Donc* : la condition doit utiliser `<`
- Penser! Ne compiler pas bêtement en essayant des valeurs au hasard

## Boucles do

- Exécute le corps de la boucle au moins une fois :

```
do
    statement
while (condition);
```

- Exemple: Valider une saisie

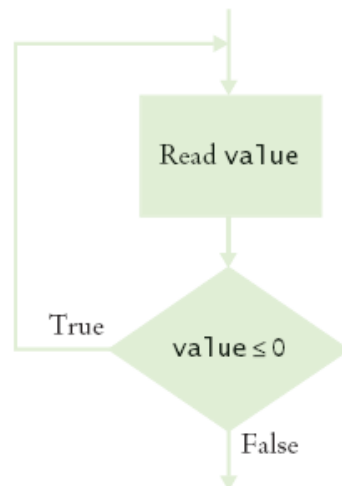
```
double value;
do
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
}
while (value <= 0);
```

## Boucles do /2

- Alternative :

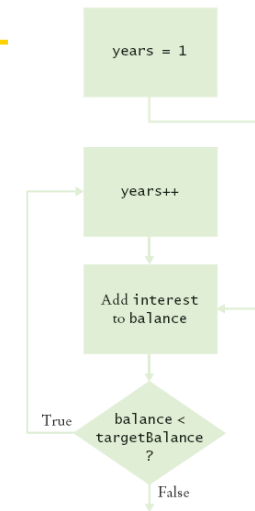
```
boolean done = false;
while (!done)
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
    if (value > 0) done = true;
}
```

## Flowchart do



Flowchart of a do Loop

## Code Spaghetti



Spaghetti Code

## Boucles for

- `for (initialization; condition; update)`  
    `statement`

- **Exemple :**

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

- **Equivalent à**

```
initialization;  
while (condition)  
{ statement;  
  update; }
```

## Boucles for /2

- **Autres exemples:**

```
for (years = n; years > 0; years--) . . .  
for (x = -10; x <= 10; x = x + 0.5) . . .
```

## Flowchart for

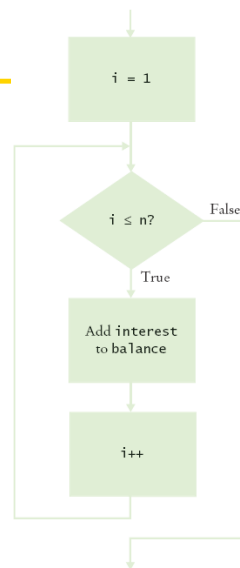


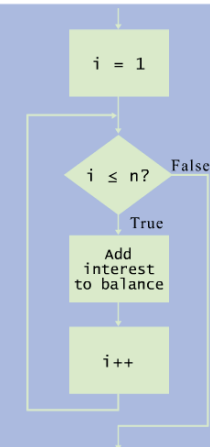
Figure 2 Flowchart of a for Loop

## Animation 6.2

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

i	n	balance	rate	interest
3	1000	10		

This animation demonstrates the for loop.





## Syntaxe Instruction for

```
for (initialization; condition; update)
    statement
```

### Exemple:

```
for (int i = 1; i <= n; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

### Objectif :

Initialiser une variable, puis exécuter un bloc d'instruction et mettre à jour une expression tant qu'une condition est vraie.

## ch06/invest2/Investment.java

```
01: /**
02:  A class to monitor the growth of an investment that
03:  accumulates interest at a fixed annual rate
04:  */
05: public class Investment
06: {
07:     /**
08:      Constructs an Investment object from a starting balance and
09:      interest rate.
10:      @param aBalance the starting balance
11:      @param aRate the interest rate in percent
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
19:
20:     /**
21:      Keeps accumulating interest until a target balance has
22:      been reached.
```

## ch06/invest2/Investment.java /2

```
23:     @param targetBalance the desired balance
24:     */
25:     {
26:         while (balance < targetBalance)
27:         {
28:             years++;
29:             double interest = balance * rate / 100;
30:             balance = balance + interest;
31:         }
32:     }
33: }
34:
35: /**
36:  Keeps accumulating interest for a given number of years.
37:  @param n the number of years
38:  */
39: public void waitYears(int n)
40: {
41:     for (int i = 1; i <= n; i++)
42:     {
43:         double interest = balance * rate / 100;
44:         balance = balance + interest;
```

## ch06/invest2/Investment.java /3

```
45:     }
46:     years = years + n;
47: }
48:
49: /**
50:  Gets the current investment balance.
51:  @return the current balance
52:  */
53: public double getBalance()
54: {
55:     return balance;
56: }
57:
58: /**
59:  Gets the number of years this investment has accumulated
60:  interest.
61:  @return the number of years since the start of the investment
62:  */
63: public int getYears()
64: {
65:     return years;
66: }
```

## ch06/invest2/Investment.java /4

```
67:
68:     private double balance;
69:     private double rate;
70:     private int years;
71: }
```

## ch06/invest2/InvestmentRunner.java

```
01: /**
02:     This program computes how much an investment grows in
03:     a given number of years.
04: */
05: public class InvestmentRunner
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         final int YEARS = 20;
12:         Investment invest = new Investment(INITIAL_BALANCE, RATE);
13:         invest.waitYears(YEARS);
14:         double balance = invest.getBalance();
15:         System.out.printf("The balance after %d years is %.2f\n",
16:             YEARS, balance);
17:     }
18: }
```

### Output:

The balance after 20 years is 26532.98

## Erreur courant : Point virgule

### • Point virgule manquant

```
for (years = 1;
    (balance = balance + balance * rate / 100) <
        targetBalance;
    years++)
    System.out.println(years);
```

### • Point virgule qui ne devrait pas être là

```
sum = 0;
for (i = 1; i <= 10; i++);
    sum = sum + i;
System.out.println(sum);
```

## Boucles imbriquées

### • Créer un motif en triangle

```
[]
[][]
[][][]
[][][][]
```

### • Boucle “sur les lignes”

```
for (int i = 1; i <= n; i++)
{
    // dessiner une ligne du triangle
}
```

### • Dessiner une ligne du triangle est une autre boucle

```
for (int j = 1; j <= i; j++)
    r = r + "[";
r = r + "\n";
```

### • Assembler les deux boucles → Boucles imbriquées

## ch06/triangle1/Triangle.java

```
01: /**
02:  This class describes triangle objects that can be displayed
03:  as shapes like this:
04:  []
05:  [] []
06:  [] [] []
07: */
08: public class Triangle
09: {
10:     /**
11:      Constructs a triangle.
12:      @param aWidth the number of [] in the last row of the triangle.
13:     */
14:     public Triangle(int aWidth)
15:     {
16:         width = aWidth;
17:     }
18:
19:     /**
20:      Computes a string representing the triangle.
21:      @return a string consisting of [] and newline characters
22:     */
```

## ch06/triangle1/Triangle.java /2

```
23:     public String toString()
24:     {
25:         String r = "";
26:         for (int i = 1; i <= width; i++)
27:         {
28:             // Make triangle row
29:             for (int j = 1; j <= i; j++)
30:                 r = r + "[]";
31:             r = r + "\n";
32:         }
33:         return r;
34:     }
35:
36:     private int width;
37: }
```

## File TriangleRunner.java

```
01: /**
02:  This program prints two triangles.
03: */
04: public class TriangleRunner
05: {
06:     public static void main(String[] args)
07:     {
08:         Triangle small = new Triangle(3);
09:         System.out.println(small.toString());
10:
11:         Triangle large = new Triangle(13);
12:         System.out.println(large.toString());
13:     }
14: }
```

## File TriangleRunner.java /2

### Output:

```
[]
>[] []
>[] [] []

>[]
>[] []
>[] [] []
>[] [] [] []
>[] [] [] [] []
>[] [] [] [] [] []
>[] [] [] [] [] [] []
>[] [] [] [] [] [] [] []
>[] [] [] [] [] [] [] [] []
>[] [] [] [] [] [] [] [] [] []
>[] [] [] [] [] [] [] [] [] [] []
```

## Manipuler des “marqueurs”

- Marqueur : Utiliser pour indiquer la fin d’une série de données

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
    We are done
else
{
    double x = Double.parseDouble(input);
    . . .
}
```

## Demi-boucle

- Parfois, la condition de terminaison d’une boucle doit être évaluée au milieu de la boucle
- Il faut introduire une variable booléenne pour contrôler l’exécution de la boucle :

```
boolean done = false;
while (!done)
{
    Print prompt
    String input = read input;
    if (end of input indicated)
        done = true;
    else
    {
        Process input
    }
}
```

## ch06/dataset/DataAnalyzer.java

```
01: import java.util.Scanner;
02:
03: /**
04:  This program computes the average and maximum of a set
05:  of input values.
06: */
07: public class DataAnalyzer
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         DataSet data = new DataSet();
13:
14:         boolean done = false;
15:         while (!done)
16:         {
17:             System.out.print("Enter value, Q to quit: ");
18:             String input = in.next();
19:             if (input.equalsIgnoreCase("Q"))
20:                 done = true;
```

## ch06/dataset/DataAnalyzer.java /2

```
21:         else
22:         {
23:             double x = Double.parseDouble(input);
24:             data.add(x);
25:         }
26:     }
27:
28:     System.out.println("Average = " + data.getAverage());
29:     System.out.println("Maximum = " + data.getMaximum());
30: }
31: }
```

## ch06/dataset/DataSet.java

---

```
01: /**
02:  Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:     /**
07:      Constructs an empty data set.
08:     */
09:     public DataSet()
10:     {
11:         sum = 0;
12:         count = 0;
13:         maximum = 0;
14:     }
15:
16:     /**
17:      Adds a data value to the data set
18:      @param x a data value
19:     */
20:     public void add(double x)
21:     {
```

## ch06/dataset/DataSet.java /2

---

```
22:         sum = sum + x;
23:         if (count == 0 || maximum < x) maximum = x;
24:         count++;
25:     }
26:
27:     /**
28:      Gets the average of the added data.
29:      @return the average or 0 if no data has been added
30:     */
31:     public double getAverage()
32:     {
33:         if (count == 0) return 0;
34:         else return sum / count;
35:     }
36:
37:     /**
38:      Gets the largest of the added data.
39:      @return the maximum or 0 if no data has been added
40:     */
```

## ch06/dataset/DataSet.java /3

---

```
41:     public double getMaximum()
42:     {
43:         return maximum;
44:     }
45:
46:     private double sum;
47:     private double maximum;
48:     private int count;
49: }
```

### Output:

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```