

Le langage Java



Dr. Ing. Ramzi FARHAT

Histoire du langage JAVA

- Java hérite principalement sa syntaxe (procédurale) du C.
- Langage généraliste.
- Plusieurs simplifications notables par rapport au C++.
- Très vaste bibliothèque de classes standard (plus de 3000 classes dans plus de 160 paquetages pour le JDK 1.5)
- A partir de 1993, chez Sun, développement pour créer un langage adapté à Internet.
- En 1995, annonce officielle de Java (conçu, entre autres, par James Gosling, Patick Naughton, Chris Warth, Ed Frank, Mike Sheridan et Bill Joy).
- Milieu 1996, sortie du Java 1.02, première version distribuée par JavaSoft (filiale de Sun).
- En 2009 rachat de Sun par Oracle
- Sep 2018, sortie du Java 11

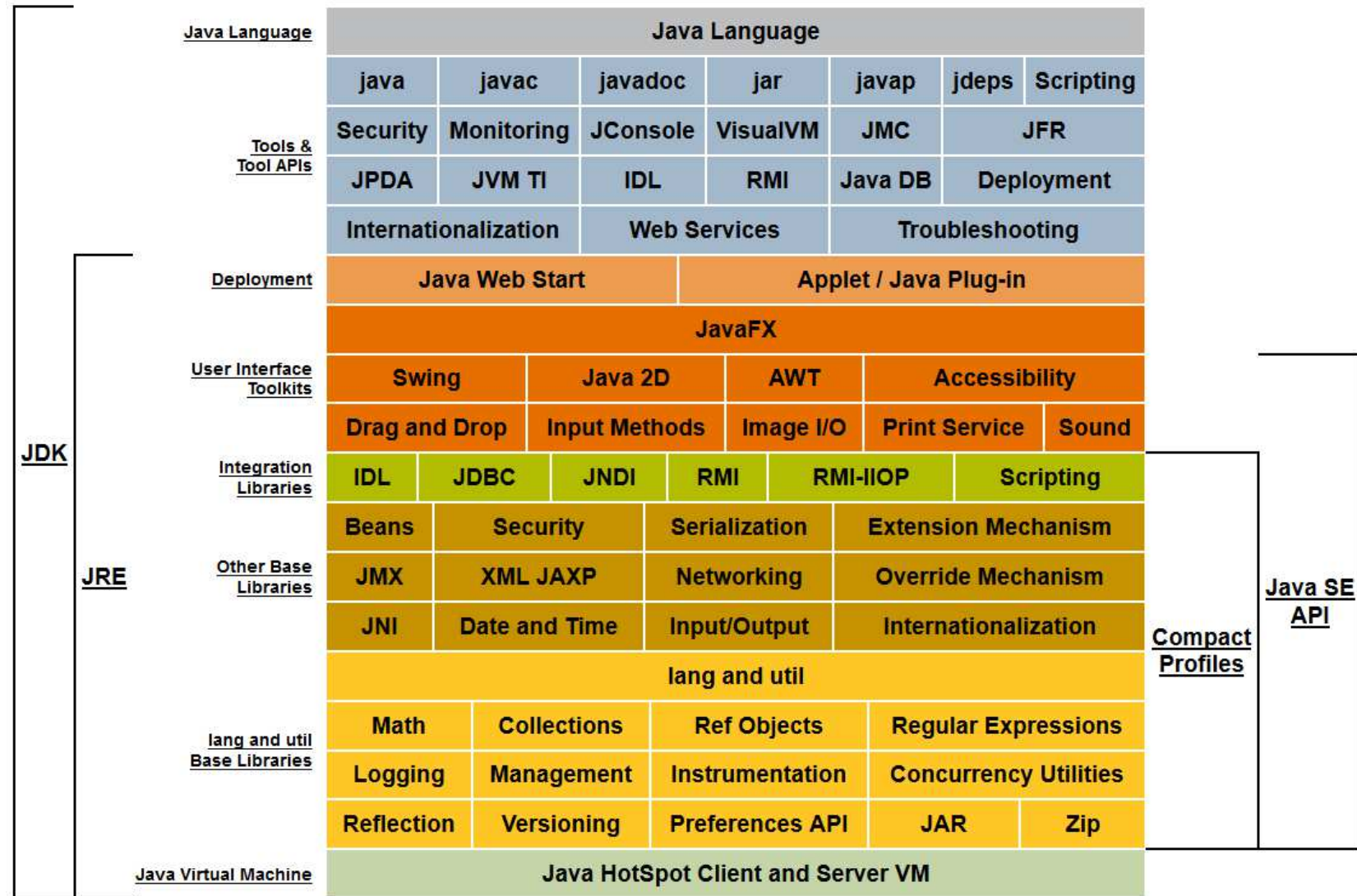
Caractéristiques

- Java est un langage :
 - simple
 - orienté objet
 - réparti
 - interprété /compilé
 - robuste
 - sûr
 - indépendant de l'architecture
 - portable
 - Efficace
 - multitâches ou multi-activités (multi-thread)
 - dynamique
 - Licence GNU General Public License (logiciel libre)

Frameworks JAVA

- On distingue essentiellement 3 grandes *plateformes* :
 - Java SE : applications pour poste de travail.
 - Java EE : applications serveurs.
 - Java ME : applications mobiles.

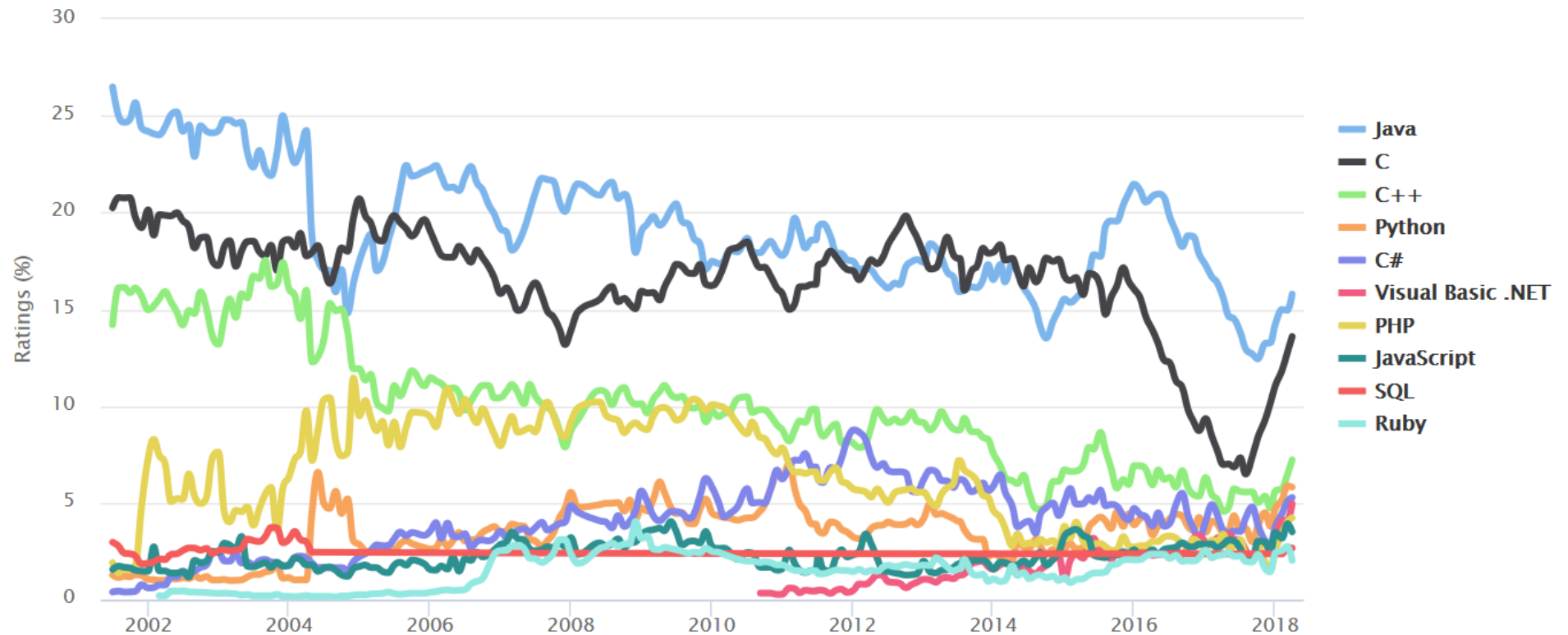
Plateforme JAVA SE 8



Popularité de Java

TIOBE Programming Community Index

Source: www.tiobe.com







Classement Spectrum IEEE 2017






Choose a Ranking (choose a weighting or make your own)

IEEE Spectrum Trending **Jobs** Open Custom

Edit Ranking | Add a Comparison |  

Language Types (click to hide)

 Web  Mobile  Enterprise  Embedded

Language Rank	Types	Jobs Ranking
1. Java	  	100.0
2. C	  	99.4
3. Python	 	99.3
4. C++	  	92.2
5. JavaScript	 	89.9
6. C#	  	86.4
7. PHP		80.5
8. HTML		79.7
9. Ruby	 	76.6
10. Swift	 	76.4

<https://spectrum.ieee.org/>

Plan

- Introduction à JAVA
- Notions de base
- Programmation Objet
- Exceptions
- Structures de données
- Communication avec les bases de données
- Programmation générique
- Collections
- Threads
- Bases de la programmation graphique

A thick blue horizontal bar with rounded ends at the top of the slide.

INTRODUCTION A JAVA

Structure d'un Programme

Access Modifier Class name

```
public class HelloWorld {
```

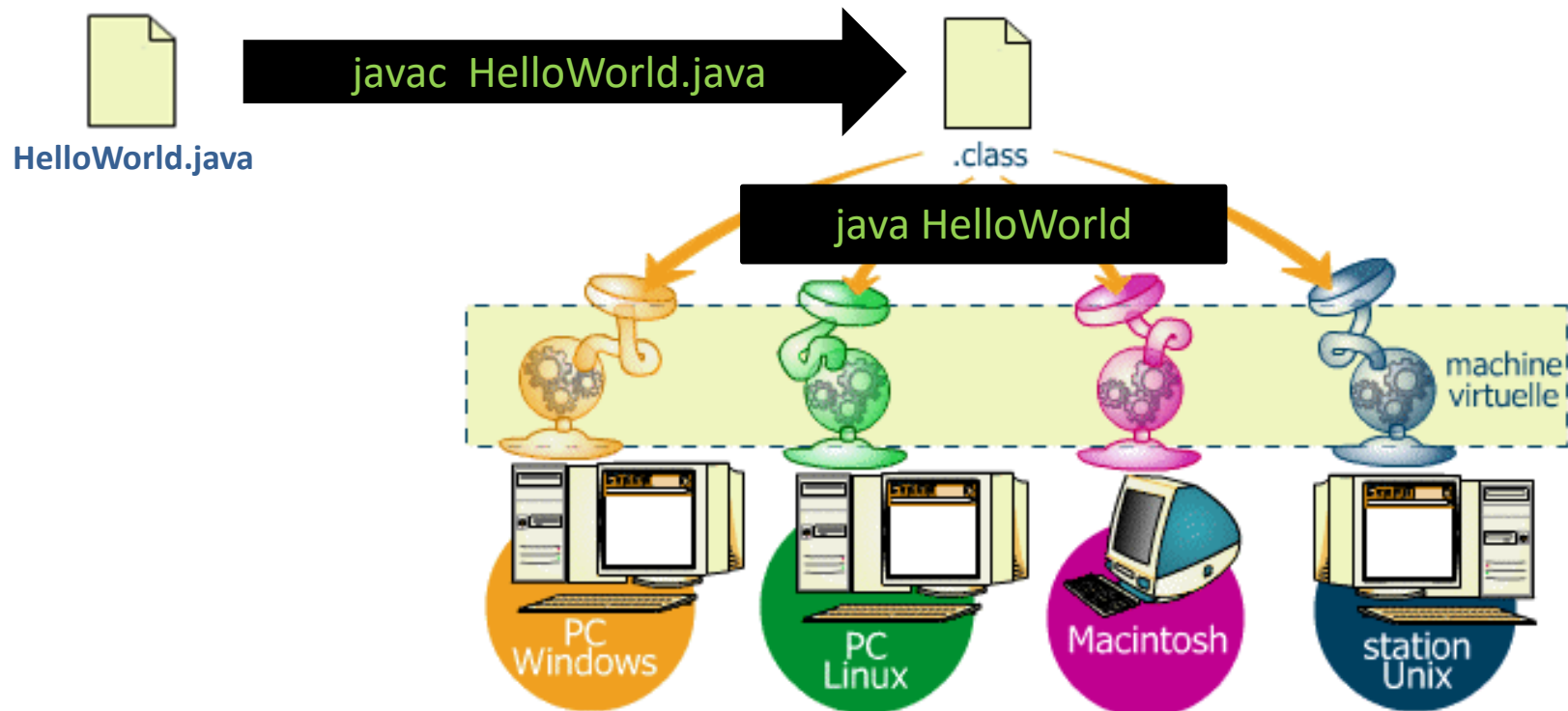
Access Modifier Non-access Modifier Method name paramètres

```
    public static void main(String[] args) {  
        System.out.println("Hello World !");  
    }  
}
```

Toute classe «marquée » public doit être écrite dans un fichier d'extension « java » qui porte le nom de la classe.

Dans un programme le point d'entrée est la méthode « main ».

Principe de la machine virtuelle (1)

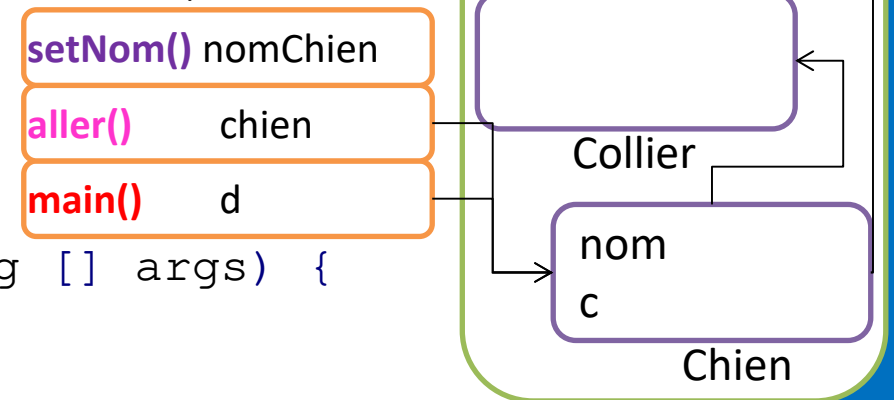


Principe de la machine virtuelle (2)

- Mémoire :
 - **Stack** : pile pour les variables locales aux méthodes
 - **Heap** : stockage des objets (avec leurs variables d'instance)

- Exemple :

```
class Chien {  
    Collier c; // var d'instance  
    String nom; // var d'instance  
    public static void main(String [] args) {  
        Chien d; // var locale: d  
        d = new Chien();  
        d.aller(d);  
        void aller(Chien chien) { // var locale: Chien  
            c = new Collier();  
            chien.setNom("Fox");  
            void setNom(String nomChien) { // var locale: nomChien  
                nom = nomChien;  
            }  
        }  
    }  
}  
class Collier { }
```



NOTIONS DE BASE



Les identificateurs

- Toutes les entités qui doivent être manipulées par le programmeur (variables, méthodes, classes, etc.) doivent avoir un nom pour les identifier : l'identificateur
- C'est une suite de caractères qui doit suivre les règles suivantes:
 - Commencer par une **lettre** ou **_** ou **\$**
 - Contient des lettres et des chiffres
 - Ne doit pas être un mot clé

Mots clés JAVA

abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default

do
Double
else
extends
final
finally
float
for
goto
if
implements
import

instanceof
int
interface
long
native
new
null
package
private
protected
public
return

short
static
super
switch
synchronized
this
throw
throws
transient
try
void
volatile
while

Types primitifs de JAVA (1)

- Entiers (par défaut *int*)

Type	Taille (octets)	Valeur minimale	Valeur maximale
byte	1	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	2	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	4	-2 147 483 648 (Integer.MIN_VALUE)	2 147 483 647 (Integer.MAX_VALUE)
long	8	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)

- Littéraux :
 - Entiers en base décimale : 10 -19 +122
 - Entiers en base hexadécimale : 0x1A 0xA
 - Entiers en base octale : 014
 - Entier long : 25L

Types primitifs de JAVA (2)

- Réels (par défaut *double*)

Type	Taille (octets)	Précision (chiffres significatifs)	Valeur absolue minimale	Valeur absolue maximale
float	4	7	1.40239846E-45 (Float.MIN_VALUE)	3.40282347E38 (Float.MAX_VALUE)
double	8	15	4.9406564584124654E-324 (Double.MIN_VALUE)	1.797693134862316E308 (Double.MAX_VALUE)

- Littéraux
 - Réels doubles : 12.43 -0.38 -.38
4. .27 4.25E4
4.25e+4
 - Réels flottantes : 12.43f

Types primitifs de JAVA (3)

- Caractères

- Codage

- UNICODE (2 octets)

- Déclaration

- char c1, c2 ;

- Littéraux

- 'a' 'E' 'e' '+' '\172' '\u1A'

Types primitifs de JAVA (4)

- Booléens
 - Déclaration
`boolean` ordonne ;
 - Exemple d'usage
`ordonne = n < p ;`
 - Littéraux
true
false

Constantes

- On utilise le mot clé *final*
- Exemple :
final int MAX = 20;
- Par convention le nom de la constante est en majuscule
- L'initialisation d'une variable final peut être différée du moment de déclaration

Priorité des opérateurs

- Mêmes opérateurs et mêmes priorités que le langage C
- Exercice :
 - Donnez les expressions équivalentes avec parenthèses
 - $a + b * c$
 - $a * b + c \% d$
 - $- c \% d$
 - $- a + c \% d$
 - $- a / - b + c$
 - $- a / - (b + c)$

Comparateurs et opérateurs logiques

- Opérateurs relationnels

Opérateur	Signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

- Opérateurs logiques

Opérateur	Signification
!	négation
&	et
^	ou exclusif
	ou inclusif
&&	et (avec court-circuit)
	ou inclusif (avec court-circuit)

Opérateurs d'incrémentation et de décrémentation

- Pré-incrémentation : `++<opérande>`
 - Pré-décrémentation : `--<opérande>`
 - Post-incrémentation : `<opérande>++`
 - Post-décrémentation : `<opérande>--`
 - Exemples:
 - Soit i une variable entière qui contient 5
- ```
n= ++i - 5; // i contient 6 et n contient 1
```
- ```
n= i++ - 5; // i contient 6 et n contient 0
```

Opérateurs d'affectation élargie

- +=
- -=
- *=
- /=
- %=
- |=
- ^=
- &=
- Exemple :
- `i = i+x ;` est équivalent à `i+=x;`

Expression

- Définition
 - Une combinaison d'opérateurs avec des opérandes
- Remarque
 - Évaluer une expression permet de calculer sa **valeur** et son **type**
- Exemples
 - i**+**x**
 - x**<**5**

Affichage sur la sortie standard (1)

- Méthodes de `java.io.PrintStream`
 - `print()` et `println()` permettent d'afficher une valeur primitive ou un objet :
 - Tout primitif ou objet en Java peut être transformé automatiquement en une chaîne de caractères
 - Exemple :
`System.out.print("x = "+x+" et y = "+y");`

Affichage sur la sortie standard (2)

- Méthodes de `java.io.PrintStream`
 - `format()` et `printf()` permettent d'afficher une donnée formatée :
 - Premier paramètre : une chaîne de caractère avec le formatage des valeurs (Exemples : `%d` pour les entier, `%f` pour les réel, `%n` pour le retour à la ligne)
 - Suite des paramètres : les valeurs à afficher
 - Exemple :
`System.out.printf("%09.3f%n", Math.PI);`

Lecture de l'entrée standard (2)

- Créer une instance de la classe **java.util.Scanner** qui permet de lire l'entrée standard :
Scanner **sc=new** Scanner(System.**in**);
- Méthodes
 - nextInt(), nextDouble(), etc.
 - Permet de récupérer un token pour chaque type de données (sauf char)
 - nextLine()
 - Retourne les caractères saisis de la position actuelle du buffer d'entré jusqu'à un retour à la ligne sous forme d'une chaîne de caractères
- Fermer un Scanner
 - close()

Structures conditionnelles (1)

- Syntaxe :

```
if (condition)
```

```
    instruction (ou bloc d'instructions)
```

```
[else
```

```
    instruction (ou bloc d'instructions) ]
```

- Remarque :

```
if (a<=b) if (b<=c) System.out.println  
    ("ordonne") ;
```

```
else System.out.println ("non ordonne") ;
```

=> Quelle interprétation ?

Structures conditionnelles (2)

- Exercice
 - Écrivez un programme de facturation avec remise. Il lit un prix hors taxes et calcule le prix TTC correspondant (avec un taux de TVA constant de 18,6%). Il établit ensuite une remise dont le taux dépend de la valeur TTC ainsi obtenue, à savoir :
 - 0 % pour un montant inférieur à 1 000 ,
 - 1 % pour un montant supérieur ou égal à 1 000 et inférieur à 2 000 ,
 - 3 % pour un montant supérieur ou égal à 2 000 et inférieur à 5 000,
 - 5 % pour un montant supérieur ou égal à 5 000.

Structures conditionnelles (3)

- Syntaxe

```
switch (expression) {  
    case constante_1 : [ instructions ]  
    case constante_2 : [ instructions ]  
    ...  
    case constante_n : [ instructions ]  
    [ default : instructions ]  
}
```

- Remarques

- Expression doit être de types : **byte**, **short**, **int** ou **char** ou énuméré (et le type **String** depuis la version 7)
- L'instruction **break** permet de sortir du bloc **switch**

Structures itératives (1)

- Syntaxe

```
do  
    instruction  
while (condition) ;
```

- Remarque

- Condition : expression booléenne

- Exemple

```
i=0;  
do {  
    System.out.println (i);  
    i++;  
}while (i<10);
```


Structures itératives (2)

- Syntaxe

```
while (condition)  
    instruction
```

- Remarque

- Condition : expression booléenne

- Exemple

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

Structures itératives (3)

- Syntaxe

```
for ([initialisations]; [condition];  
    [incrémentations])  
    instruction
```

- Remarque

- Initialisations : suite d'expressions séparées par des virgules
- Condition : condition d'arrêt
- Incrémentations : suite d'expressions séparées par des virgules

- Exemple

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

Instructions de branchement (1)

- Instruction **break**
 - Sans branchement : Sortir de la boucle
 - Avec branchement : Sortir de la boucle qui porte l'étiquette
- Exemple

```
répéter : while(...){ //répéter : étiquette de bloc
.....
while(...) {
.....
    break répéter ; // avec branchement (*)
.....
}
while(...) {
.....
    break; // sans branchement (#)
.....
}
..... // <-- ici (#)
}
..... // <-- ici (*)
```

Instructions de branchement (2)

- Instruction ***continue***
 - Sans branchement : Permet de passer prématurément à l'itération suivante
 - Avec branchement : Permet de passer prématurément vers l'étiquette d'une boucle
- Exemple

```
for (i=1 ; i<=5 ; i++) {  
    System.out.println ("debut tour "+i);  
    if (i<4) continue ;  
    System.out.println ("fin tour "+i);  
}
```

Instructions de branchement (2)

- Instruction ***continue***
 - Sans branchement : Permet de passer prématurément à l'itération suivante
 - Avec branchement : Permet de passer prématurément vers l'étiquette d'une boucle

- Exemple

```
externe : for (i=0; i<=2; i++)
```

```
    interne : for (j=0; j<=2; j++)
```

```
        if(i==1 && j==1) continue externe;
```

```
        else System.out.println("i="+i+"-j="+j);
```



LA PROGRAMMATION OBJET AVEC JAVA

Introduction

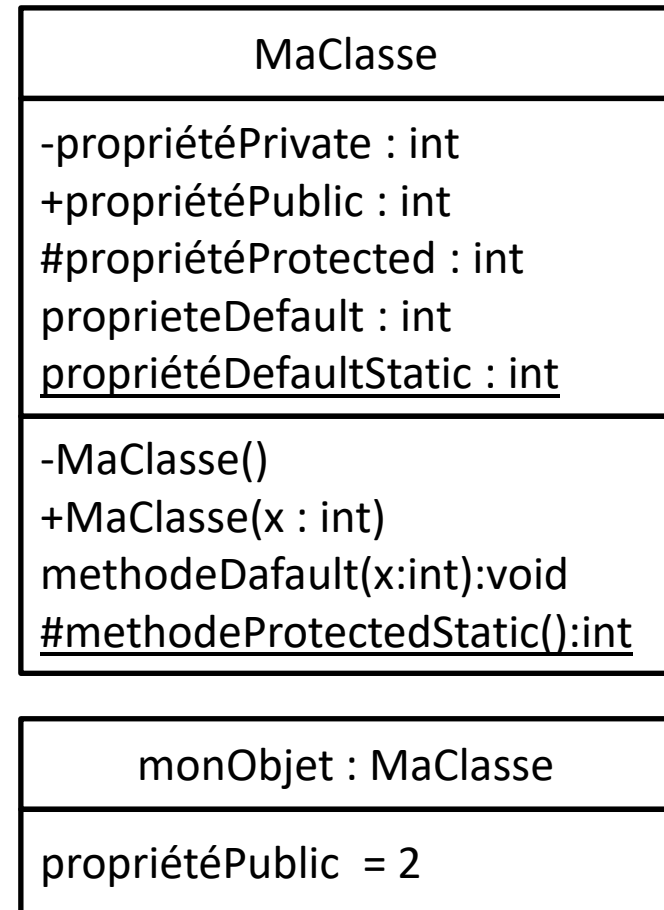
- Programmation orientée objet
 - Contribution à la fiabilité des logicielles
 - Facilitation de la réutilisation du code
- Concepts
 - Objet
 - Classe
 - Encapsulation
 - Héritage
 - Polymorphisme

Objet et encapsulation

- Un **objet** est l'association de données (état) et de traitements (comportement):
 - Structuration selon les structures de données
 - L'objet dispose d'un état et d'un comportement
- L'**encapsulation** consiste à cacher l'implémentation
 - Uniquement les signatures des méthodes sont visibles de l'extérieur

Classe (1)

- Le concept de classe est la généralisation du concept de type
 - Description de l'état et du comportement commun à un ensemble d'objets similaires
 - Les objets sont des instances de classes



Classe (2)

- Exemple de définition de classe

```
public class Point{
```

Déclaration d'une **classe**
accessible à tous nommée Point

```
private int x ; // abscisse
```

```
private int y ; // ordonnee
```

Déclaration de **variables**
d'**instances** non accessibles de
l'extérieur de la classe

```
public void initialise (int abs, int ord){
```

```
    x = abs ;
```

```
    y = ord ;
```

```
}
```

```
public void deplace (int dx, int dy){
```

```
    x += dx ;
```

```
    y += dy ;
```

```
}
```

Déclarations de **méthodes**
accessibles à tous

```
public void affiche (){
```

```
    System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
```

```
}
```

```
}
```

Classe (3)

- Utilisation des méthodes d'une classe
 - Créer une instance de cette classe : créer un objet
 - Appeler les méthodes de cet objet
- Exemple

```
public class TestPoint{  
    public static void main (String args[]) {  
        Point a ;           // Variable de type référence sur un objet Point  
        a = new Point() ;   // Construction d'un nouveau objet et assignation  
        a.initialise(3, 5) ; // Usage des méthodes de l'objet créé et  
        a.affiche() ;       // référencé par la variable a  
        a.deplace(2, 0) ;  
        a.affiche() ;  
  
        Point b = new Point() ; // deuxième objet instance de la classe  
        b.initialise (6, 8) ; b.affiche() ; // Point  
    }  
}
```

Classe (4)

- Mise en œuvre d'un programme
 - Chaque classe dans un fichier (.java) toutefois il est possible de mettre plusieurs classes dans un même fichier (condition : une seule doit être publique et le nom de fichier java doit porter le nom de cette classe).
 - Exemple :
 - Deux fichiers sources : *Point.java* et *TestPoint.java*
 - Puis compiler pour obtenir : *Point.class* et *TestPoint.class*
 - Puis exécuter (la classe qui contient la méthode *main()*) :
java TestPoint

Constructeur (1)

- Une méthode exécutée lors de la création d'un objet (instanciation de la classe)
- Contient les traitements jugés utiles pour le bon fonctionnement de l'objet
- Porte le nom de la classe, sans valeur de retour et qui peut disposer d'arguments
- Si aucun constructeur n'est spécifié, le compilateur crée un constructeur implicitement.

Constructeur (2)

- Exemple :
 - L'usage d'un objet Point nécessite l'appel de la méthode ***void initialise(int, int)*** qui permet d'initialiser l'abscisse et l'ordonnée du Point.
 - Problème :
 - Et si celui qui utilise l'objet oublie d'appeler en premier lieu cette méthode ?
 - Solution :
 - Utiliser un constructeur

Constructeur (3)

- Exemple :

```
public class Point {  
    private int x, y ;
```

```
    public Point (int abs, int ord){  
        x = abs ;  
        y = ord ;  
    }
```

```
    public void deplace (int dx, int dy){  
        x += dx ;  
        y += dy ;  
    }
```

```
    public void affiche (){  
        System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;  
    }  
}
```

Constructeur (4)

- Exemple (suite) :

```
public class TestPoint {
```

```
    public static void main(String[] args) {
```

```
        Point a ;
```

```
        a = new Point(3, 5) ;
```

```
        a.affiche() ;
```

```
        a.deplace(2, 0) ;
```

```
        a.affiche() ;
```

```
        Point b = new Point(6, 8) ;
```

```
        b.affiche() ;
```

```
    }
```

```
}
```


Initialisation d'un objet (1)

- Les variables d'instances peuvent être initialisées

- par défaut

- **boolean** à **false**
- **char** à **caractère de code nul**
- **Entier** (byte, short, int, long) à **0**
- **Flottant** (float, double) à **0.f** ou **0.**
- Objet à **null**

- explicitement lors de la déclaration

- par un constructeur



L'instruction : `A a = new A(...);`

Provoque :

1. l'initialisation *implicite* des variables
2. Éventuellement l'initialisation *explicite* des variables
3. L'exécution des instructions du *constructeur*

Initialisation d'un objet (2)

- Remarques :
 - Le mot clé ***final*** peut être utilisé pour la déclaration des variables d'instances
 - Les variables d'instances ***final*** peuvent être initialisées tardivement (au plus tard dans un constructeur)
 - On peut utiliser un bloc d'initialisation (technique non recommandée) : le bloc est exécuté après l'initialisation implicite puis selon sa position (avant ou après les initialisation explicites) et finalement le constructeur.

Initialisation d'un objet (3)

```
public class Initialisations {  
  
    int x;  
    int y = x*2;  
    {  
        x=5;  
        y=20;  
    }  
    public Initialisations(){  
        System.out.println  
            ("x="+x+" y="+y);  
        x=10;  
    }  
    public void affichage(){  
        System.out.println  
            ("x="+x+" y="+y);  
    }  
}
```

Résultat 1 ?

```
public class Initialisations {  
  
    int x;  
    int y = x*2;  
  
    public Initialisations(){  
        System.out.println  
            ("x="+x+" y="+y);  
        x=10;  
    }  
    {  
        x=5;  
        y=20;  
    }  
    public void affichage(){  
        System.out.println  
            ("x="+x+" y="+y);  
    }  
}
```

Résultat 2 ?

```
public class Initialisations {  
  
    {  
        x=5;  
        y=20;  
    }  
    int x;  
    int y = x*2;  
    public Initialisations(){  
        System.out.println  
            ("x="+x+" y="+y);  
        x=10;  
    }  
    public void affichage(){  
        System.out.println  
            ("x="+x+" y="+y);  
    }  
}
```

Résultat 3 ?

Méthodes d'un objet

- Constructeurs
- Méthodes d'accès (Eng. Accessor)
 - Nomenclature : **getXXXXX**
- Méthodes d'altération (Eng. Mutator)
 - Nomenclature : **setXXXXX**
- Exemples :

```
public void setCoordonnees(int x, int y){...};  
public int getAbscisse(){...};  
public int getOrdonnee(){...};
```
- Méthodes métier
 - Nom qui commence par une minuscule puis en "CamelCase"

Affectation et objets

- Affectation

- Exemple :

Point **a**, **b**, **c** ;

.....

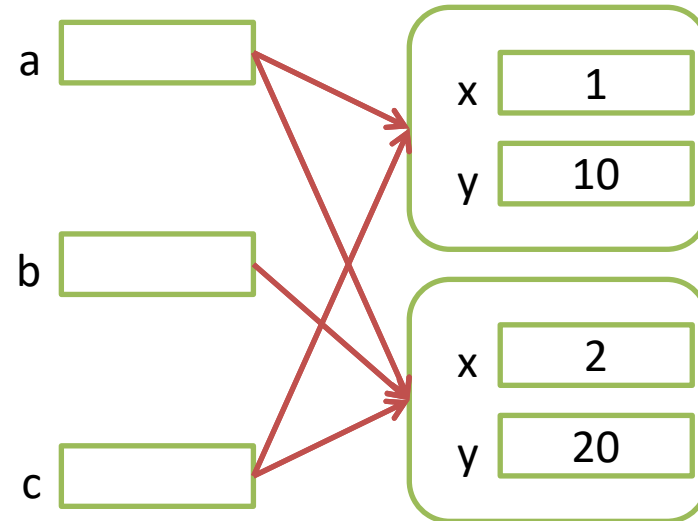
a = new Point (**1**, **10**) ;

b = new Point (**2**, **20**) ;

c = **a** ;

a = **b** ;

c = **b** ;



Écriture des Méthodes

- Valeur de retour :
 - Si la méthode retourne une valeur il faut :
 - Préciser le *type* de la valeur de retour au niveau de l'entête
 - Utiliser l'instruction *return* pour retourner la valeur
 - Exemple :

```
double distancePointOrigine(){  
    double d;  
    d=Math.sqrt(x*x+y*y);  
    return d;  
}
```

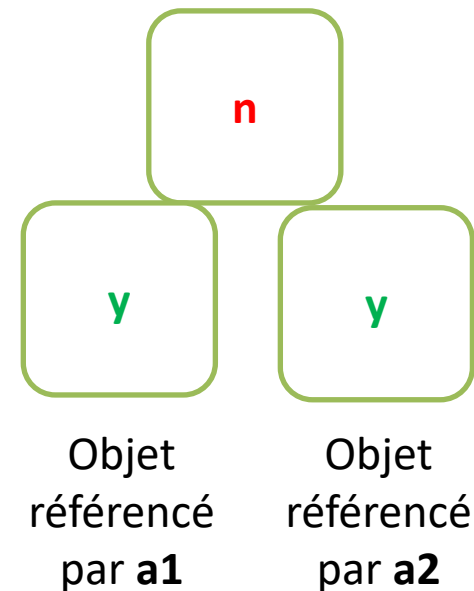
Variables de classe (1)

- Présentation :
 - C'est une variable qui, au lieu d'exister dans chaque instance de la classe, n'existe qu'en un seul exemplaire pour toutes les instances de la classe.
 - On l'appelle aussi variable statique

- Déclaration :

- Exemple :

```
class B{  
    static int n;  
    float y;  
    ...  
}  
class C{  
    B a1 = new B(), a2 = new B();  
    ...  
}
```



Variables de classe (2)

- Exercice
 - Créez une classe nommée **Objet** qui contient une variable statique de type entier qui contiendra le nombre d'instances déjà créées de cette classe (qui doit s'incrémenter à chaque création d'une instance).
 - Créez une classe nommée **TestObjet** qui contient une méthode **main()** permettant de créer trois objets de type **Objet** et d'afficher pour chacun le numéro d'instance créée.

Variables de classe (3)

- Remarque :
 - L'initialisation d'une variable de classe doit se faire avant l'appel au constructeur. En effet, cette variable peut être utilisée sans même qu'une instance de classe n'est créée et donc sans appel au constructeur.
 - Exemple :
 - Si **A** est une classe qui comporte une variable de classe **x** de type entier, alors on peut avoir le code suivant :

```
class B{  
    int y = A.x;  
    ...  
}
```

Méthode de classe

- Présentation
 - Méthode indépendante des instances de la classe
 - Exemples : `main()`, `Math.sqrt()`, ...
 - Déclarée avec le mot clé ***static***
 - Ne peut agir que sur des variables de classe
 - Exercice :
 - Écrivez une méthode dans la classe `Objet` permettant d'accéder au nombre d'objets de cette classe.

Bloc d'initialisation Statique

- Principe :
 - Il n'agit que sur des variables de classe
- Exemple :

```
class A{  
    private static int t[5] ;  
    ...  
    static { .....  
        for (int i=0 ; i<5 ; i++) t[i] = i ;  
    }  
    ...  
}
```

Surcharge de méthodes (1)

- Nomenclature
 - Surcharge ou Surdéfinition (overloading)
- Principe
 - Plusieurs méthodes peuvent porter le même nom mais avec impérativement une différence au niveau des paramètres formels (nombre, type ou ordre)
 - Le type de retour n'intervient pas dans le surcharge des méthodes

Surcharge de méthodes (2)

- Exemple :

```
class Point{
    private int x, y ;
    // constructeur
    public Point (int abs, int ord){
        x = abs ; y = ord ;
    }
    // deplace (int, int)
    public void deplace (int dx, int dy){
        x += dx ; y += dy ;
    }
    // deplace (int)
    public void deplace (int dx){
        x += dx ;
    }
    // deplace (short)
    public void deplace (short dx){
        x += dx ;
    }
}
```

```
public class TestSurcharge{
    public static void main (String args[]){
        Point a = new Point (1, 2) ;
        // appelle deplace (int, int)
        a.deplace (1, 3) ;
        // appelle deplace (int)
        a.deplace (2) ;
        short p = 3 ;
        // appelle deplace (short)
        a.deplace (p);
        byte b = 2 ;
        /* Quelle méthode ? */
        a.deplace (b) ;
    }
}
```

Surcharge de méthodes (3)

- Exercice :
 - Définissez une classe **Point** qui comporte un surcharge de la méthode constructeur. Elle doit permettre de construire :
 - Un objet Point dont l'abscisse et l'ordonnée sont égaux à zéro
 - Un objet Point dont l'abscisse et l'ordonnée sont initialisés à une valeur donnée en paramètre
 - Un objet Point dont l'abscisse et l'ordonnée sont initialisés avec deux valeurs données en paramètre
 - Un objet Point dont l'abscisse et l'ordonnée sont identiques à celles d'un autre objet Point donné en paramètre
 - Définissez une classe **TestPoint** qui permet de faire le test unitaire de la classe **Point**

Passage de paramètres (1)

- Java fait **toujours** un passage de paramètres **par valeur**
- Ceci implique que :
 - les valeurs dont le type est primitif sont copiées localement dans les paramètres formels.
 - les valeurs dont le type est un objet sont copiées (référence vers un objet) mais pas l'objet.

Passage de paramètres (2)

```
public class Point {  
  
    private int x , y;  
  
    public int getX() {  
        return x;  
    }  
    public void setX(int abs) {  
        x= abs;  
    }  
    public int getY() {  
        return y;  
    }  
    public void setY(int ord) {  
        x= ord;  
    }  
    public Point(int abs, int ord){  
        x= abs;  
        y=ord;  
    }  
}
```

```
    public void permuterVO(Point pt){  
        int temp;  
        temp = pt.getX();  
        pt.setX() = pt.getY();  
        pt.setY() = temp;  
    }  
  
    public void permuterVP (int abs, int ord){  
        int temp;  
        temp = abs;  
        abs = ord;  
        ord = temp;  
    }  
}
```


Passage de paramètres (3)

```
public class TestPassageParametres {  
    public static void main(String[] args) {  
        Point a = new Point (1, 3) ;  
        Point b = new Point (1, 3) ;  
        a.permuterVO(a);  
        b.permuterVP(b.getX(), b.getY());  
        System.out.println ("Permutation paramètre objet: x = " +  
a.getX() + ", y = " + a.getY()) ;  
        System.out.println ("Permutation paramètres primitifs: x = " +  
b.getX() + ", y = " + b.getY()) ;  
    }  
}
```

Résultat ?

Autoréférence

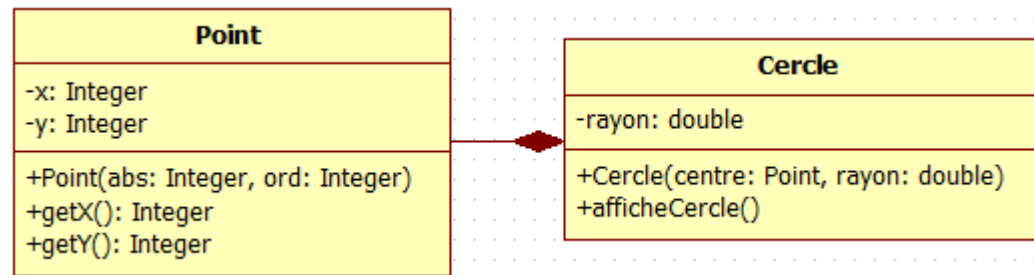
- Pour qu'un objet fait référence à lui-même on utilise le mot clé **this**
- Exemple :

```
public class Point {  
    private int x, y;  
    public Point (int x, int y){  
        x = x;  
        y = y;  
    }  
    public Point (int z){  
        ; // appel du constructeur Point(int, int)  
    }  
    ...  
}
```

Objet membre (1)

- C'est une variable d'instance d'un objet qui est de type objet
- Exemple :
 - Classe cercle qui est définie par un center et un rayon
 - Le centre est un objet Point
 - Le rayon est un réel double

- Exercice :

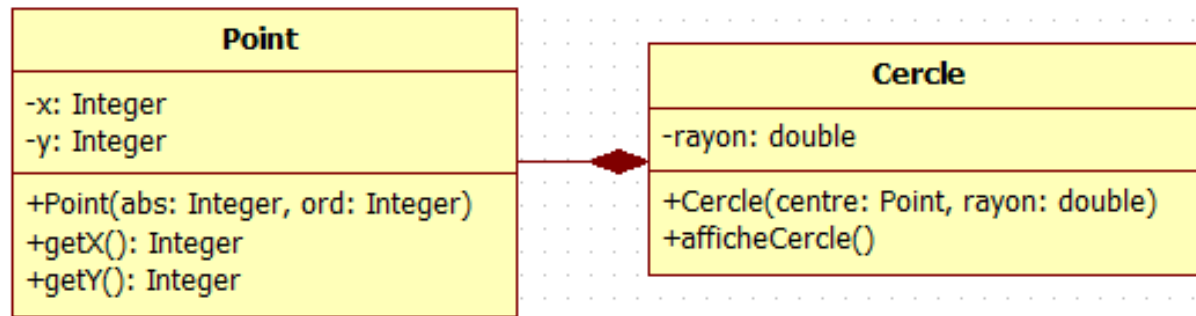


– Donnez le code de la classe cercle

Objet membre (2)

- Solution :

```
public class Cercle {  
    private Point centre;  
    private double rayon;
```



```
    public Cercle(Point centre, double rayon){  
        this.centre = centre;  
        this.rayon= rayon;  
    }
```

```
    public void afficheCercle(){  
        System.out.println("Coordonnées centre : "+ centre.getX() + " , "+  
        centre.getY());  
        System.out.println("Valeur du rayon : "+ rayon);  
    }  
}
```

Classe interne (1)

- C'est une classe défini dans une autre classe
 - Un objet de la classe interne est toujours **associé** à un objet de classe externe (qui lui a donné naissance)
 - Un objet de la classe interne a **accès aux variables et méthodes** (même privés) de l'objet de la classe externe qui lui a donné naissance
 - Un objet de la classe externe a **accès aux variables et méthodes** (même privés) de l'objet de la classe interne auquel il a donné naissance

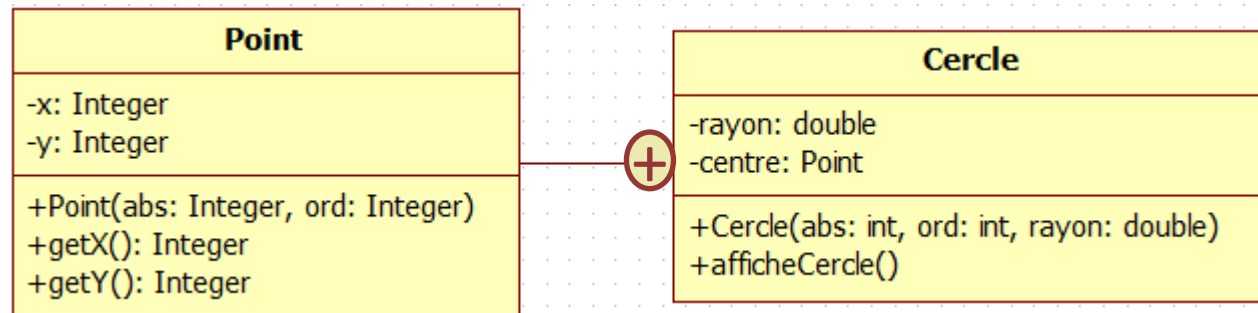
Classe interne (2)

- Exemple

```
public class Cercle {
    private double rayon;
    private Point centre;
    public Cercle(int abs, int ord,
        double rayon){
        centre = new Point(abs, ord);
        this.rayon= rayon;
    }
    public void afficheCercle(){
        System.out.println ("Centre :
        "+ centre.x+ ", "+ centre.y);
```

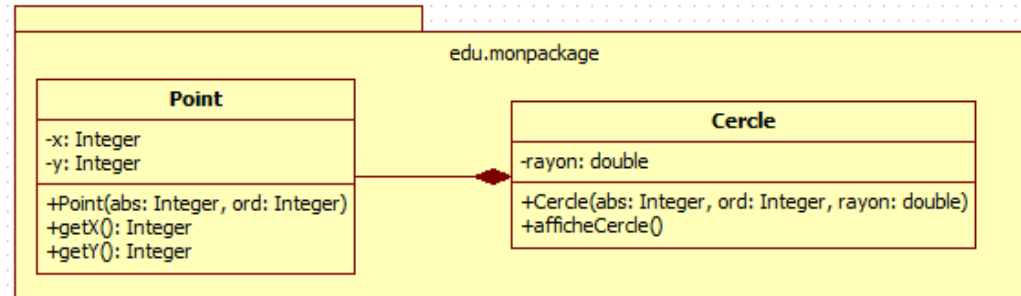
```
System.out.println ("Rayon : "+
rayon);
```

```
}
class Point {
    private int x , y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
    ...
}
```



Paquetage (1)

- Regroupement **logique** d'un ensemble de classe



- Porte un identificateur (par convention en miniscule),
- Conventionnellement il doit avoir un root parmi : com, edu, gov, mil, net, org ou des lettres d'identification du pays
 - Exemple : **edu.monpackage**
- Pour indiquer qu'une classe appartient à un package :
package <nom de package>;
- Si rien n'est indiqué la classe appartient au package par défaut (default package)

Paquetage (2)

- Pour faire référence à une classe qui appartient à un autre package :
 - 1^{ère} méthode :
`<nom package>.<nom classe>`
 - Exemple :
`edu.monpackage.Point p = new edu.monpackage.Point();`
 - 2^{ième} méthode :
`import <nom paquetage>.<Nom classe OU *>;`
 - Exemple :
`import edu.monpackage.Point;`
`import edu.monpackage.*;`

Paquetage (3)

- Exercice :
 - Créez deux classes publiques A et B appartenant chacune à un package différent. Puis dans la classe **A** fait appel dans une méthode nommée **a()** à une méthode publique **b()** de la classe **B**. la méthode **b()** doit afficher le message “Hello package !”.

Modificateurs d'accès (1)

- Objectif :
 - Fixer la **visibilité** d'une classe ou d'un membre d'une classe par rapport aux autres classes.
 - Autoriser ou non l'**accès** à une classe
 - Instancier la classe
 - Hériter de la classe
 - Accéder aux membre de la classe
 - Modificateurs d'accès pour les classes :
 - **public** : visible à toutes les classes
 - **default** : visible uniquement aux classes du même package

Modificateurs d'accès (2)

- Modificateurs d'accès pour les membres (variables et méthodes) d'une classe :
 - **public**
 - **protected**
 - **default**
 - **private**
- Remarque : si une classe n'est pas visible alors quelque soit le type du modificateur d'accès des membres ils ne seront pas visibles.

Modificateurs d'accès (3)

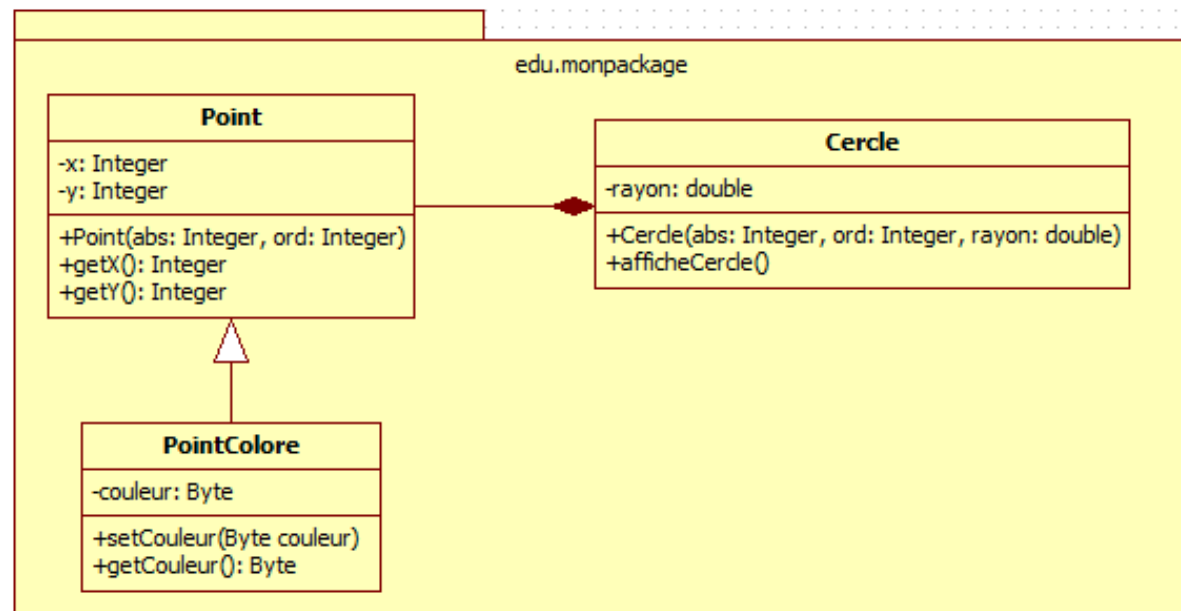
		Même package		Autre package	
	Même Classe	Sous-classe	Autre classe	Sous-classe	Autre Classe
private	X				
default (package)	X	X	X		
protected	X	X	X	X (seulement par héritage)	
public	X	X	X	X	X

Par accès : créer un objet et accéder à un de ses membres

Par héritage : utiliser un membre hérité

Héritage (1)

- Le concept d'héritage consiste à réutiliser ce qui a été défini pour une classe (classe de base/super-classe/classe mère) tout en le spécialisant (classe dérivée/sous-classe/ classe fille)



Héritage (2)

- Exemple :
 - Classe **PointCouleur** qui est un **Point** qui dispose d'une couleur :

```
package edu.monpaquetage;  
class PointCouleur extends Point {  
    private byte couleur ;  
    public PointCouleur (int x, int y, byte couleur){  
        super(x, y);  
        this.couleur = couleur ;  
    }  
    public void setCouleur (byte couleur){  
        this.couleur = couleur ;  
    }  
    public byte getCouleur(){  
        return couleur;  
    }  
}
```

Héritage (3)

- Exercice :
 - Ecrivez une classe **TestPointCouleur** qui instancie **PointCouleur** et affiche ses coordonnées et son couleur.

```
package edu.monpaquetage;  
public class TestPointCouleur{  
    public static void main (String args[]){  
        PointCouleur pc = new PointCouleur(1,5, (byte)3) ;  
        System.out.printf("Abscisse : %d, Ordonnée : %d,  
Couleur : %d", pc.getX(), pc.getY(), pc.getCouleur());  
    }  
}
```

Héritage (4)

- Règle d'accès
 - Une sous-classe n'hérite pas les membres privés de sa super-classe
- Constructeur
 - Le constructeur de la sous classe doit prendre en charge l'intégralité de la construction de l'objet
 - Pour appeler le constructeur de la classe mère il faut l'invoquer via le mot clé **super** suivi des éventuels arguments (**super()** doit être la première instruction dans le constructeur)
 - Si la sous-classe ne possède pas de constructeur la classe de base doit disposer d'un constructeur publique sans arguments (ou aucun constructeur)

Héritage (6)

- Supposons que :

```
class B extends A { . . . . . }
```

- La création d'un objet de type *B* se déroule en 6 étapes.
 1. Allocation mémoire pour un objet de type B
 2. Initialisation par défaut de tous les attributs de B
 3. Initialisation explicite, s'il y a lieu, des attributs hérités de A éventuellement, exécution des blocs d'initialisation de A
 4. Exécution du corps du constructeur de A.
 5. Initialisation explicite, s'il y a lieu, des attributs de B ; éventuellement, exécution des blocs d'initialisation de B
 6. Exécution du corps du constructeur de B.

Redéfinition de méthodes (1)

- Redéfinition de méthodes (Overriding) :
 - **Redéfinir** le comportement d'une **méthode héritée** en la spécialisant tout en gardant la **même signature** (dont le type de la valeur de retour)
 - Exemple :
 - Supposant qu'on veut définir une méthode publique ***afficher()*** au niveau de la classe ***Point*** qui affiche les propriétés d'un point
 - On souhaite que l'affichage dans la sous-classe ***PointCouleur*** affiche en plus la ***couleur*** d'un point

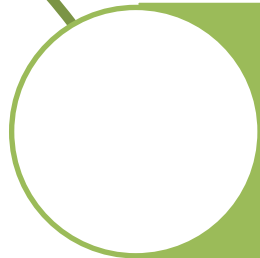
Redéfinition de méthodes (2)

- Exemple :

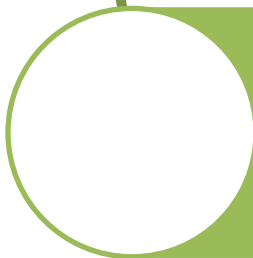
```
public class Point {  
    ...  
    public void affiche () {  
        System.out.println("Abscisse : "+x+" Ordonnee :  
        "+y");  
    }  
    ...  
}  
class PointColore extends Point {  
    ...  
    public void affiche () {  
        super.affiche();  
        System.out.print(" Couleur : "+couleur) ;  
    }  
    ...  
}
```

Redéfinition de méthodes (3)

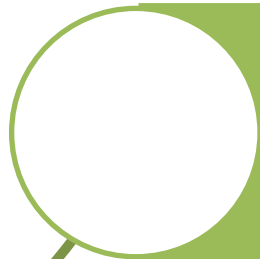
- Règles à suivre pour une redéfinition correcte



les valeurs de retour des deux méthodes doivent être exactement de même type (ou, depuis le JDK 5.0, être covariantes),



le droit d'accès de la méthode de la classe dérivée ne doit pas être moins élevé que celui de la classe de base,



la clause *throws* au niveau de la redéfinition ne doit pas mentionner des exceptions non mentionnées dans la clause *throws* de la méthode de la super classe

Surcharge de méthodes (1)

- Surcharge de méthodes (Overloading) :
 - La **surcharge** d'une **méthode héritée** consiste à écrire une méthode qui porte le même nom mais avec des arguments différents
 - Exemple :

```
class A {  
    ...  
    public void f (int n) { ..... }  
    public void f (float x) { ..... }  
}  
class B extends A {  
    ...  
    public void f (int n) { ..... } // redéfinition  
    public void f (double y) { ..... } // surcharge  
}
```

Surcharge de méthodes (2)

- Exemple :

```
class A {  
    public void f (int n) { ..... }  
    .....  
}  
class B extends A {  
    public void f (float x) { ..... }  
    .....  
}  
A a ; B b ;  
int n ; float x ;  
.....  
a.f(n) ; // appelle f (int) de A  
a.f(x) ; // erreur de compilation : conversion interdite de float à int  
b.f(n) ; // appelle f (int) de A  
b.f(x) ; // appelle f(float) de B
```

Polymorphisme (1)

- Principe :
 - Permet de manipuler des objets sans connaître (tout à fait) le type
- Polymorphisme en Java
 - Compatibilité par affectation entre une variable qui référence une super-classe et un objet dont le type peut être une sous-classe
 - Liaison dynamique des méthodes (choix de la méthode à exécuter à l'exécution et non à la compilation)

Polymorphisme (2)

- Exemple :
 - On peut déclarer une variable de type **Point** et qui fait référence à un objet de type **PointCouleur** puisque c'est un Point (héritage)
 - Ensuite on peut invoquer la méthode ***afficher()*** et l'objet va réagir selon sa vraie nature

Point **p**;

p = new Point (3, 5) ;

p.affiche () ; //méthode affiche de la classe Point

p = new PointCouleur (4, 8, (byte) 2) ;

p.affiche () ; //méthode affiche de la classe PointCouleur

Polymorphisme (3)

- Exemple :

```
class A{
1 public void f (float x) { ..... }
    .....
}
class B extends A{
2 public void f (float x) { ..... } // redéfinition de f
3 public void f (int n) { ..... }  // surcharge de f
    .....
}
A a = new A(...);
B b = new B(...);
int n;
a.f(n);    // 1, 2 ou 3 ?
b.f(n);    // 1, 2 ou 3 ?
a = b;
a.f(n);    // 1, 2 ou 3 ?
```

Classe Object (1)

- Présentation :
 - C'est la classe dont hérite ***implicitement*** toute classe qui n'hérite pas explicitement d'une autre classe.
- Conséquences :
 - Toute classe hérite les méthodes publiques de la classe Object :
 - public String toString()
 - public boolean equals (Object)
 - ...
 - Toute objet peut être affecté à une variable de type Object

Classe Object (2)

- Exemple :

```
Point p = new Point (...);
```

```
PointColore pc = new PointColore (...);
```

```
Fleur f = new Fleur (...);
```

```
Object o ;
```

```
...
```

```
o = f; // OK
```

```
o = p; // OK
```

```
o = pc; // OK
```

```
o.deplace(); // Erreur de compilation
```

```
((Point)o).deplace(); // OK en compilation
```

```
Point p1 = (Point) o; // OK : idem ci-dessus, création d'une
```

```
p1.deplace(); // référence intermédiaire dans p1
```

Classe Object (3)

- Méthode ***toString*** :
 - Retourne une chaîne de caractères qui contient :
 - Nom de la classe
 - HashCode en hexadécimal (précédée de @)
 - Exemple :

```
public class Test{
    public static void main (String args[]){
        Point a = new Point (1, 2) , b = new Point (5, 6) ;
        System.out.println ("a = " + a.toString()) ;
        System.out.println ("b = " + b.toString()) ;
    }
}
```

Résultat :
a = Point@fc17aedf
b = Point@fc1baedf

Classe Object (4)

- Méthode ***equals*** :
 - Retourne **true** si les deux objets ont la même référence
 - Exemple :

```
class Point{
    ...
    @Override
    public boolean equals (Object p) { ... }
}
class Test{
    public static void main(){
        Point a = new Point (1, 2) ;
        Point b = new Point (1, 2) ;
        Object o1 = new Point (1, 2) ;
        Object o2 = new Point (1, 2) ;
        if (a.equals(b) == true)    System.out.println("a = b");
        else                      System.out.println("a != b");
        if (o1.equals(o2) == true) System.out.println("o1 = o2");
        else                      System.out.println("o1 != o2");
    }
}
```

Classe Object (5)

- Exercice :

Donnez le code source de la classe **Point** qui exploite les méthodes **toString()** (permettant d'afficher les coordonnées d'un Point) et **equals()** permettant de retourner vrai si deux points sont égaux.

Modificateur « final » (1/3)

- Une méthode déclarée final ne peut pas être redéfinie dans une sous-classe
- Exemple :

```
class MaClasse{  
    ...  
    final void maMethodeFinal () {...}  
    ...  
}
```

Modificateur « final » (2/3)

- Une classe déclarée final ne peut plus être dérivée
- Exemple :

```
final class MaClasseFinal {  
    ...  
}
```


Modificateur « final » (3/3)

- Une variable d'instance déclarée final ne peut plus être changée une fois qu'elle a été initialisée.
- Elle doit être initialisée explicitement (dans la déclaration, dans un bloc d'initialisation ou au niveau du constructeur).
- Exemple :

```
class MaClasse {  
    final int x = 10;  
    ...  
}
```

Classe abstraite (1)

- Présentation :
 - C'est une classe qui peut contenir des méthodes abstraites.
 - Une méthode est abstraite si on ne donne pas sa définition (les détails d'implémentation)
- Conséquences :
 - On ne peut pas instancier une classe abstraite
 - Une méthode abstraite doit obligatoirement être publique
 - Une sous-classe ne doit pas obligatoirement définir une ou toutes les méthodes abstraites de la super-classe

Classe abstraite (2)

- Syntaxe

```
abstract class A {  
    public void f() {                // f est définie  
        ...  
    }  
    public abstract void g(int n) ; // g n'est pas définie  
    ...  
}
```

Classe abstraite (3)

- Intérêt
 - Permet d'imposer des fonctionnalités qu'on souhaite avoir dans toutes les sous-classes instanciables
 - On impose aux développeurs des sous-classes de suivre les signatures des méthodes abstraites et de proposer leurs implémentations

Interface (1)

- Présentation :
 - Elle définit des signatures de méthodes et des constantes
 - Une classe peut implémenter plusieurs interfaces
 - Une interface peut hériter d'une ou de plusieurs interfaces
 - On peut déclarer des variables dont le type est une interface

Interface (2)

- Syntaxe :

// l'interface est soit public soit accès paquetage (default)

public interface ExempleInterface{

// déclaration d'une constante (static final facultatifs)

int MAXI = 100 ;

// en-tête d'une méthode f (public abstract facultatifs)

void f(int n) ;

// en-tête d'une méthode g (public abstract facultatifs)

void g() ;

}

Interface (3)

- Syntaxe d'implémentation:

```
public interface I1{  
    void f() ;  
}
```

```
public interface I2{  
    int h() ;  
}
```

```
public interface I3 extends I2{  
    int k() ;  
}
```

```
class A implements I1, I3 {
```

```
    /* A doit obligatoirement définir les méthodes f, h et k prévues dans I1  
    et I3 qui hérite de I2 */
```

```
    ...
```

```
}
```

Garantie qu'offre une classe
d'implémenter les
fonctionnalités définies par une
interface

Interface (4)

- Nouveautés à partir de Java 8 :
 - Il est possible de définir des méthodes statiques dans une interface

```
static void direBonjour() {  
    System.out.println("Bonjour !");  
}
```
 - Il est possible de définir le comportement par défaut (**default**) d'une méthode dans une interface

```
default void direBonsoir() {  
    System.out.println("Bonsoir !");  
}
```
 - En cas de conflit entre Interface et classe mère la classe mère le remporte
 - En cas de conflit entre deux interface la classe doit redéfinir la méthode

Interface (5)

- Soit le code suivant :

```
interface I1{
    static void direBonjour() {
        System.out.println("Bonjour !");
    }
    default void direBonsoir() {
        System.out.println("Bonsoir !");
    }
    default void direSalut() {
        System.out.println("Salut !");
    }
}
interface I2 {
    default void direSalut() {
        System.out.println("Salut !!!");
    }
}
class MaClasse {
    public void direBonsoir() {
        System.out.println("Bonsoir !!!");
    }
}
```

```
}
}
public class TestInterfaceJava8
extends MaClasse implements I1, I2
{
    ...
}
```

Donner le code permettant d'exécuter les trois méthodes de l'interface **I1**, la méthode de l'interface **I2** et la méthode de la classe **MaClasse**

Classes enveloppes (1)

- Contexte :
 - Les **objets** et les variables de type **primitif** ne se comportent pas exactement de la même façon :
 - Affectation : Adresse vs. Valeur
 - Compatibilité de type : Héritage vs. Hiérarchie de type
 - Polymorphisme : Uniquement les objets
 - Collections : Uniquement pour les objets
- Présentation :
 - Les classes enveloppes (wrappers) permettent de manipuler les types primitifs comme des objets

Classes enveloppes (2)

- Présentation (suite) :
 - Exemples de classes enveloppes :
 - **Boolean, Character, Integer, Double, ...**
 - Elles sont finales et les six classes numériques dérivent de la classe **Number**
- Exemple :
`Integer x = new Integer(12);`

Classes enveloppes (3)

Méthodes

– Constructeur

- accepte un argument du type primitif correspondant

– **xxxValue()** avec xxx représente le nom du type primitif :

- retourne la valeur du type primitif correspondant
- Exemples : intValue(), floatValue(), etc.

– **equals()** :

- Effectue la comparaison en profondeur des objets

Classes enveloppes (4)

Emballage et déballage

— Présentation :

- Mécanisme de simplification de manipulation des classes enveloppes introduit depuis le **JDK 5.0**
- **Emballage** (Boxing) : on peut affecter une valeur primitive à une classe d'enveloppe sans passer par le *constructeur*
- **Déballage** (unboxing) : on peut avoir les valeurs encapsulés dans les classe enveloppes sans appel à la méthode *xxxValue*

Classes enveloppes (5)

- Exemples :

Integer nObj = 12 // au lieu de := new Integer (12)

Double xObj = 5.25 ; // au lieu de := new Double (5.25)

.....

int n = nObj ; // au lieu de := nObj.intValue()

double x = xObj // au lieu de = xObj.doubleValue()

nObj1 = nObj2 + 2 ; /* nObj2 est converti en int auquel
on ajoute 2, le résultat est converti en Integer */

nObj1++ ; /* nObj1 est converti en int, auquel
on ajoute 1 le résultat est converti en Integer */



LES EXCEPTIONS

Introduction

Motivations

- Besoin d'écrire des programmes robustes
- Besoin d'écrire un code lisible (Traiter les erreurs potentiels dans le code source produit un code compliqué et illisible)

Solution Java : Exceptions

- Code efficace et bien organisé
- Détection des erreurs sans passer par le test des valeurs de retour.
- Séparation entre le code susceptible de générer les erreurs du code qui traite les erreurs
- Un même traitement peut être associé à plusieurs types d'erreurs potentielles
- Possibilité de récupérer l'erreur à plusieurs niveaux d'une application (propagation dans la pile d'appels)

Introduction (2)

Définition Exception

- Une « condition exceptionnelle » qui altère le déroulement normal d'un programme.
- Un signal indiquant que quelque chose d'exceptionnel (comme une erreur) s'est produit, et qui interrompt le flot d'exécution normal du programme

Causes

- Une ressource ou une condition nécessaire pour exécuter normalement un programme n'est pas présente.

Sources

- Problèmes matériels (Réseau, disque, etc.)
- Epuisement de ressources (RAM, Connexion à une BD, etc.)
- Erreurs de Programmation (Division par zéro, dépassement de la taille d'un tableau, etc.)
- etc.

Gestion d'exceptions (1)

Déroulement

- Lorsqu'une exception se produit (thrown) le code responsable de son traitement (exception handler) la capte (catches) pour la traiter.

Syntaxe de gestion d'exception

```
try{  
    // Code à risque : il peut déclencher une exception  
}  
catch (TypeDException e){  
    // Code pour gérer l'exception si elle se produit  
}  
// On peut avoir plusieurs catch pour un même code à risque
```

Gestion d'exceptions (2)

Exemple

- Le constructeur d'un point n'accepte que des coordonnées positives, sinon une exception **ErrConstructeurException** est déclenchée

```
public class TestException {  
    public static void main (String args[]){  
        try{  
            Point a = new Point (-1, 4) ;  
            System.out.println (a);  
        }  
        catch (ErrConstructeurException e){  
            System.out.println ("L'une des coordonnées est négative !") ;  
        }  
    }  
}
```

Marquer les instructions qui risquent de déclencher une exception

Gestionnaire de l'exception **ErrConstructeurException**

Gestion d'exceptions (3)

- Le bloc **try** est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- Si une exception est levée :
 - Les instructions dans le bloc **try** qui suivent l'instruction qui a déclenché l'exception ne seront pas exécutées.
 - les clauses **catch** sont examinées l'une après l'autre dans le but d'en trouver une qui traite cette classe d'exceptions (ou une superclasse).
 - Les clauses **catch** doivent donc traiter les exceptions de la plus spécifique à la plus générale.
 - Si une clause **catch** convenant à cette exception a été trouvée, le bloc associé est exécuté

Gestion d'exceptions (4)

- Dans un bloc **catch** :
 - On peut arrêter l'exécution : **System.exit(int)**
 - On peut retourner une valeur (avec **return**) et donc continuer l'exécution à partir du code qui a appelé la méthode qui contient le bloc catch
 - On peut re-déclencher l'exception (avec **throw**) pour la méthode appelante
 - On peut déclencher une nouvelle exception (avec **throw new**) pour la méthode appelante
 - On peut continuer l'exécution après le dernier bloc **catch ... {}** de la clause **try{ ...}** en cours

Gestion d'exceptions (5)

Comment savoir si l'exécution d'une méthode peut déclencher une exception ?

- La signature de la méthode

Composantes d'une signature

- Nom de la méthode
- Valeur de retour
- Paramètres
- **Exceptions** qu'elle peut déclencher

Exemple

```
void maMethode() throws PremiereException,  
DeuxiemeException{  
    ...  
}
```

Gestion d'exceptions (6)

- Exemple :

```
11 //lecture d'un fichier texte
12 InputStream ips=new FileInputStream("d:\\f.txt");
13
14
15
16
17
```

Unhandled exception type **FileNotFoundException**

2 quick fixes available:

- [Add throws declaration](#)
- [Surround with try/catch](#)

Press 'F2' for focus

← → ↺ 🏠 <https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html>

FileInputStream

```
public FileInputStream(String name)
    throws FileNotFoundException
```

Gestion d'exceptions (7)

Gestion d'un code à risque

- Gérer l'exception :

```
void LireFichier(){  
    //...  
    //lecture d'un fichier texte  
    try{  
        InputStream ips=new FileInputStream("d:\\f.txt");  
        //...  
    }  
    catch(FileNotFoundException e){  
        System.out.println("Erreur de lecture du fichier !");  
    }  
    // ...  
}
```

- Remonter l'exception :

```
void LireFichier() throws FileNotFoundException{  
    //...  
    //lecture d'un fichier texte  
    InputStream ips=new FileInputStream("d:\\f.txt");  
    //...  
}
```


Remonter une exception (1)

- Clause « **throws** » :
 - Toute méthode susceptible de déclencher une exception qu'elle ne traite pas localement doit mentionner son type dans une clause ***throws*** figurant dans son en-tête.
 - Ainsi l'entête de la méthode permet de savoir exactement à quelles exceptions on est susceptible d'être confronté.

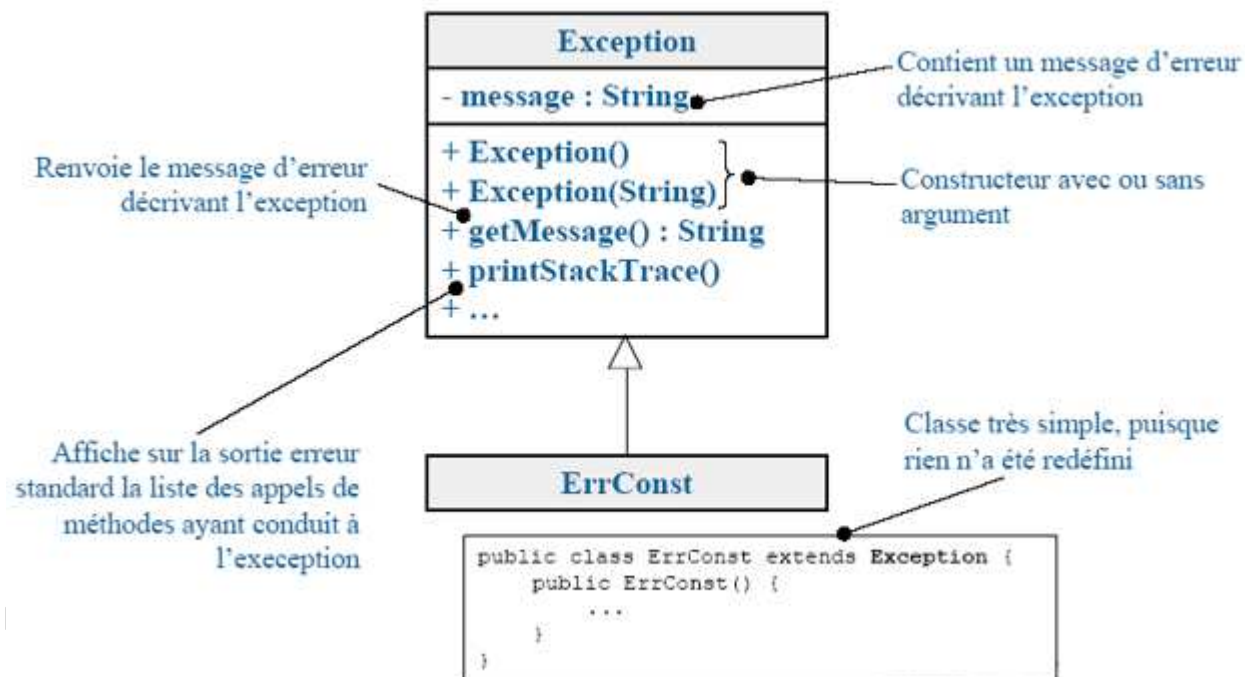
Remonter une exception (2)

- Si elles ne sont pas immédiatement capturées par un bloc ***catch***, les exceptions se propagent en remontant la pile d'appels des méthodes, jusqu'à être traitées.
- Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode ***main()***, ce qui pousse l'interpréteur Java à afficher un message d'erreur et à s'arrêter.
- L'interpréteur Java affiche un message identifiant : ***l'exception***, la ***méthode*** qui l'a causée, la ***ligne*** correspondante dans le code.

Créer une Exception

Objectifs

- Il faut créer une classe qui dérive de la classe **Exception** (Directement ou Indirectement)
- Il faut ajouter un code qui déclenche une instance de cette exception via le mot clé **throw** si détection d'une situation qui empêche l'exécution normale du programme.



Créer une Exception

Exemple

```
class ErrConstructeurException extends Exception {  
}
```

Indique que la méthode est susceptible de déclencher une exception de type **ErrConstructeurException**

```
class Point{  
    public Point(int x, int y) throws ErrConstructeurException {  
        if ( (x<0) || (y<0))  
            throw new ErrConstructeurException ();  
        this.x = x ;  
        this.y = y ;  
    }  
    ...  
}
```

Déclencher une exception de type **ErrConstructeurException**

Transmission d'informations au gestionnaire d'exception (1)

1^{ère} façon : Via le constructeur

Exemple

```
class ErrConstructeurException extends Exception {  
    private int abs, ord ;  
    ErrConstructeurException(int abs, int ord){  
        this.abs = abs ; this.ord = ord ;  
    }  
    ...  
}  
  
class Point{  
    public Point(int x, int y) throws ErrConstructeurException {  
        if ( (x<0) || (y<0) ) throw new ErrConstructeurException (x, y) ;  
        ...  
    }  
    ...  
}
```

L'objet **ErrConstructeurException** reçoit les coordonnées qui ont déclenché l'exception

Envoie des coordonnées qui ont déclenché l'exception

Transmission d'informations au gestionnaire d'exception (2)

1^{ère} façon (suite)

```
public class TestException{  
    public static void main (String args[]){  
        try{  
            Point a = new Point (-3, 5) ;  
            System.out.println(a);  
        }  
        catch (ErrConstructeurException e){  
            System.out.println ("Erreur construction Point") ;  
            System.out.println("Coord. erronees: "+e.getAbs() +" "  
                               "+e.getOrd());  
        }  
    }  
}
```

Usage des informations passées
à l'objet **ErrConstructeur**

Transmission d'informations au gestionnaire d'exception (3)

2^{ème} façon

- Via la méthode ***getMessage()*** qui retourne un message passé au constructeur de l'Exception

Exemple

```
class ErrConstructeurException extends Exception {  
    ErrConstructeurException (String message){  
        super (message);  
    }  
}  
class Point{  
    public Point(int x, int y) throws ErrConstructeurException {  
        if ( (x<0) || (y<0)) throw new ErrConstructeurException  
        ("Coord. erronees: "+x+" "+y) ;  
        ...  
    }  
    ...  
}
```

Appel au constructeur de la classe mère
Exception

Message relatif à l'exception

Transmission d'informations au gestionnaire d'exception (4)

2^{ème} façon (suite)

```
public class TestException{  
    public static void main (String args[]){  
        try{  
            Point a = new Point (-3, 5) ;  
            a.affiche() ;  
        }  
        catch (ErrConstructeurException e){  
            System.out.println(e.getMessage() );  
            System.exit (-1) ;  
        }  
    }  
}
```

Récupérer le message relatif à
l'exception

Exercice

```
class Point{
    private int x, y ;
    public Point(int x, int y) throws ErrConst{
        if ( (x<=0) || (y<=0)) throw new
        ErrConst() ;
        this.x = x ;
        this.y = y ;
    }
    public void f() throws ErrBidon{
        try{
            Point p = new Point (-3, 2) ;
        }
        catch (ErrConst e){
            System.out.println ("dans catch de f") ;
            // on lance une nouvelle exception
            throw new ErrBidon() ; }
        System.out.println ("apres catch de f") ;
    }
}
```

```
class ErrConst extends Exception{ }
class ErrBidon extends Exception{ }
public class TestException{
    public static void main (String args[]){
        try{
            Point a = new Point (1, 4) ;
            a.f() ; }
        catch (ErrConst e){
            System.out.println (" dans catch
ErrConst de main") ; }
        catch (ErrBidon e){
            System.out.println ("dans catch
ErrBidon de main") ; }
        System.out.println ("Après catch de
main") ;
    }
}
```

- Donnez les messages affichés par ce programme Java

Bloc finally (1)

- Un bloc **finally** permet au programmeur de définir un ensemble d'instructions qui est toujours exécuté, ***que l'exception soit levée ou non, capturée ou non.***
- La seule instruction qui peut empêcher **finally** d'être exécuté est **System.exit(int)**.
- Le bloc **finally** se situe après le dernier catch d'un bloc **try**

Bloc finally (2)

Exemple

```
class Point{
    private int x, y ;
    public Point(int x, int y) throws ErrConst{
        if ( (x<=0) || (y<=0)) throw new ErrConst();
        this.x = x; this.y = y ;
    }
    public void f() throws ErrBidon{
        try{
            Point p = new Point (-3, 2) ;
        }
        catch (ErrConst e){
            System.out.println ("dans catch de f") ;
            throw new ErrBidon() ; }
        finally{
            System.out.println ("dans finally de f") ;
        }
        System.out.println ("apres catch de f") ;
    }
}
```

```
class ErrConst extends Exception{ }
class ErrBidon extends Exception{ }
public class TestException{
    public static void main (String args[]){
        try{
            Point a = new Point (1, 4) ;
            a.f() ;
        }
        catch (ErrConst e){
            System.out.println (" dans catch
ErrConst de main") ; }
        catch (ErrBidon e){
            System.out.println ("dans catch
ErrBidon de main") ; }
        System.out.println ("Après catch de main") ;
    }
}
```



Donnez les messages affichés par ce programme Java **sans** et **avec** finally

Gestion des ressources (1/3)

Ressource

- Objet qui doit être fermé (libéré) une fois qu'on n'a plus besoin dans le programme
- Exemples : Fichier, Connexion JDBC, Connexion Socket, etc.

Avant Java 7

- Libération des ressources utilisées dans un bloc **try/catch** dans **finally**

A partir de Java 7

- Libération automatique des ressources une fois le bloc **try/catch** est exécuté

Gestion des ressources (2/3)

- Exemple avec Java 6 :

```
public class Java6ResourceManagement {
    public static void main(String[] args) {
        BufferedReader br = null;
        FileReader fr = null;
        try {
            fr = new FileReader("C:\\ex.txt");
            br = new BufferedReader(fr);
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (br != null) br.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

- Exemple avec Java 7 :

```
public class Java7ResourceManagement {
    public static void main(String[]
args) {
        try (BufferedReader br = new
BufferedReader(new FileReader(
"C:\\ex.txt"))) {
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Gestion des ressources (3/3)

- Remarques :
 - Code lisible et facile à écrire
 - Gestion automatique des ressources
 - Moins de lignes de code
 - Possibilité de gérer plusieurs ressources (séparation par ";") avec libération des ressources dans le sens inverse de leur allocation.

Multi catch

Présentation

- Possibilité d'associer plusieurs types d'exceptions à un seul bloc catch à partir de **Java 7**

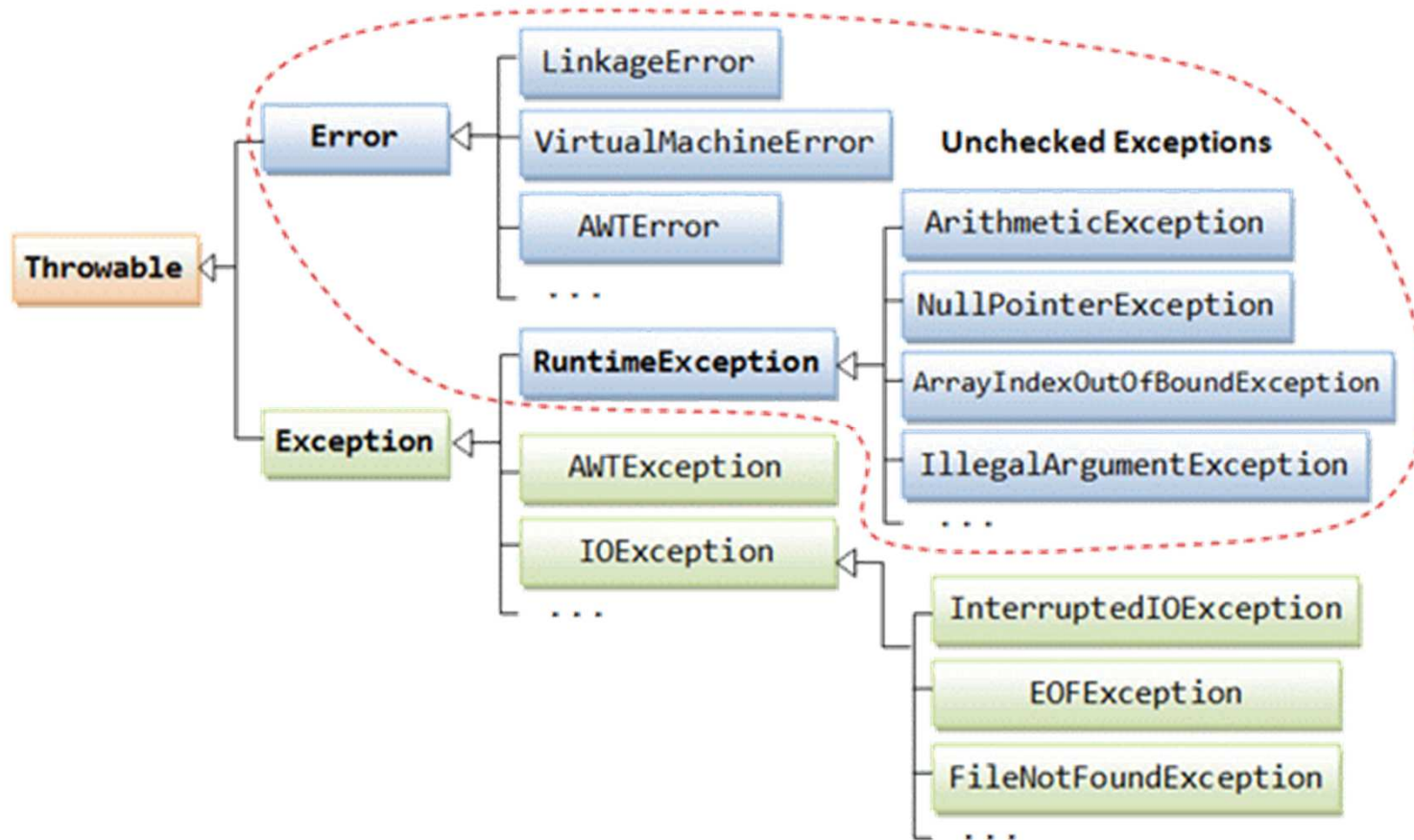
Syntaxe

```
catch (Exception1 | ... | ExceptionN e) {  
    ...  
}
```

Objectif

- Avoir un code concis et lisible

Vue d'ensemble



Source : http://www.ntu.edu.sg/home/ehchua/programming/java/J5a_ExceptionAssert.html

Vue d'ensemble

- Java fournit de nombreuses classes qui héritent de la classe Exception
- Ces exceptions se classes en deux catégories :
 - Exceptions explicites (ou sous contrôle – **checked-**)
 - Exemple de source : erreurs de lecture ou d'écriture
 - Exceptions implicites (ou hors contrôle – **unchecked-**)
 - Exemples de sources : division par zéro, débordement d'indice de tableau, etc.
- Il est obligatoire de gérer uniquement les exceptions sous contrôle

Vue d'ensemble

Sans exceptions

```
erreurType lireFichier() {
    int codeErreur = 0;
    // Ouvrir le fichier
    if (isFileIsOpen()) {
        // Détermine la longueur du fichier
        if (getFileSize()) {
            // Vérification de l'allocation de la mémoire
            if (getEnoughMemory()) {
                // Lire le fichier en mémoire
                if (readFailed()) {
                    codeErreur = -1;
                }
            } else {
                codeErreur = -2;
            }
        } else {
            codeErreur = -3;
        }

        // Fermeture du fichier
        if (closeTheFileFailed()) {
            codeErreur = - 4;
        }
    } else {
        codeErreur = - 5;
    }
}
```

La gestion des erreurs
devient très difficile

Difficile de gérer les
retours de fonctions

Le code devient de plus
en plus conséquent

Vue d'ensemble

Avec exceptions

Le mécanisme d'exception permet

- La concision
- La lisibilité

```
void lireFichier() {  
    try {  
        // Ouvrir le fichier  
        // Détermine la longueur du fichier  
        // Vérification de l'allocation de la mémoire  
        // Lire le fichier en mémoire  
        // Fermer le fichier  
    } catch (FileOpenFailed) {  
        ...  
    } catch (FileSizeFailed) {  
        ...  
    } catch (MemoryAllocFailed) {  
        ...  
    } catch (FileReadFailed) {  
        ...  
    } catch (FileCloseFailed) {  
        ...  
    }  
}
```



- Gestion d'exceptions
- Remonter une exception
- Créer une exception
- Transmission d'informations
- Bloc finally
- Gestion des ressources
- Multi catch
- Vue d'ensemble

TABLEAUX

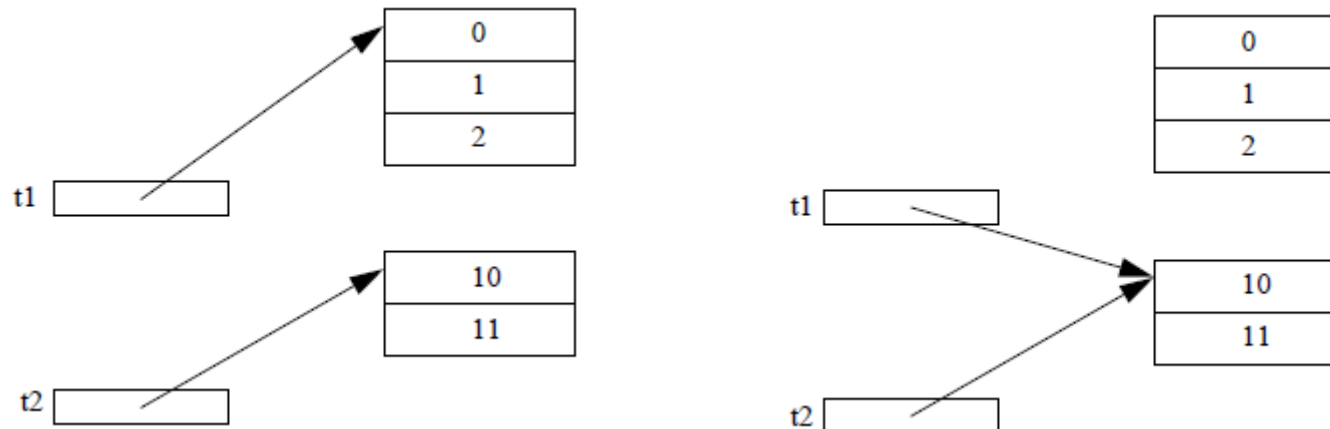


Tableaux (1)

- Déclaration d'une référence à un tableau
 - Exemples : `int t[];` ou `int[] t;`
- Création d'un nouveau tableau
 - Exemples :
 - `t = new int[5]; // initialisation implicite`
 - `int n=2, p=3; int tab[]={1,n,n+p,2*p,12};`
- Accès à l'élément d'un tableau
 - Exemple : `t[2];`

Tableaux (2)

- Affectation de tableaux



- Longueur d'un tableau

– Exemples :

```
int t[] = new int[5];  
System.out.println(t.length); //affiche 5
```

Tableaux (3)

- Exemple d'un tableau d'objets

```
public class TabPoint{  
    public static void main (String args[]) {  
        Point[] tp = new Point[3] ;  
        tp[0] = new Point (1, 2) ;  
        tp[1] = new Point (4, 5) ;  
        tp[2] = new Point (8, 9) ;  
        for (int i=0 ; i<tp.length ; i++)  
            tp[i].affiche() ;  
    }  
}
```


Tableaux (4)

- Boucle de consultation (depuis JDK 5.0)

- Syntaxe :

- `for(type variable:tableau) {...}`

- Exemple

```
public class TabPoint{  
    public static void main (String args[]) {  
        Point[] tp = new Point[3] ;  
        tp[0] = new Point (1, 2) ;  
        tp[1] = new Point (4, 5) ;  
        tp[2] = new Point (8, 9) ;  
        for (.....)  
            .....  
    }  
}
```

Tableaux (5)

- Passage d'un tableau en paramètre
 - Passage d'une **référence** à un tableau
- Valeur de retour de type tableau
 - Retourne une **référence** à un tableau

Tableau (6)

- Classe **java.util.Arrays** :
 - Offre des méthodes statiques :
 - Pour trier : **sort**(tableau), ...
 - Pour rechercher : **binarySearch**(tableau, valeur), ...
 - Etc.
 - Remarques :
 - On ne peut chercher que dans un tableau trié !
 - Le tri doit se faire sur des éléments comparables !

Tableau (7)

- Exercice :
 - Donner un code qui permet de créer un tableau d'entiers
 - D'afficher son contenu
 - De le trier
 - De rechercher une valeur dedans

Tableau (8)

- Remarque :
 - Et si on demande à `Arrays.sort()` de trier un tableau de Points ? Selon quelle logique la méthode va effectuer le tri ???

Tableau (9)

- On ne peut trier que des choses comparables
- Il faut implémenter l'interface `java.util.Comparable <T>` :
 - Comporte la méthode `int compareTo(T objet)` :
 - Retourne un entier **négatif** si l'objet courant est plus petit que l'objet en paramètre, **zéro** si égalité et **positif** si l'objet courant est plus grand

Tableau (10)

- Exercice:
 - Créez la classe **Point** qui implémente l'interface *Comparable* et qui comporte :
 - Deux variables d'instance entier : **abscisse** et **ordonnée**
 - Un constructeur qui initialise l'abscisse et l'ordonnée
 - Deux accesseurs pour les variables d'instances
 - Une redéfinition des méthodes **toString()** et **equals()**
 - Une implémentation de **compareTo()** basée sur la comparaison de la distance euclidienne par rapport à l'origine du repère
 - Créez la classe **TestPoint** qui crée un tableau de points, l'affiche, le trie et l'affiche de nouveau

Tableau (11)

- Remarque :
 - Et si on a plusieurs critères de tri ?
- Exemple :
 - L'objet est un livre
 - On peut trier les livres dans un tableau selon :
 - Leurs titres
 - Leurs prix
 - Leurs nombre de pages
 - Etc.

Tableau (12)

- Pour chaque critère de tri il faut créer une classe qui implémente l'interface `java.util.Comparator<T>`
- Elle contient la méthode :
 - `public int compare(T o1, T o2)`
- Utilisation :
 - `Arrays.sort(T[] tableau, Comparator c)`

Tableau de tableaux (1)

- Java ne dispose pas de la notion de **tableau à plusieurs indices** mais dispose de la notion de **tableau de tableaux**

- Déclaration

– Exemples :

```
int t [] [] ;
```

```
int [] t [] ;
```

```
int [] [] t ;
```

- Initialisation (Exemple 1)

```
– int t [] [] = { new int [3], new int [2] } ;
```

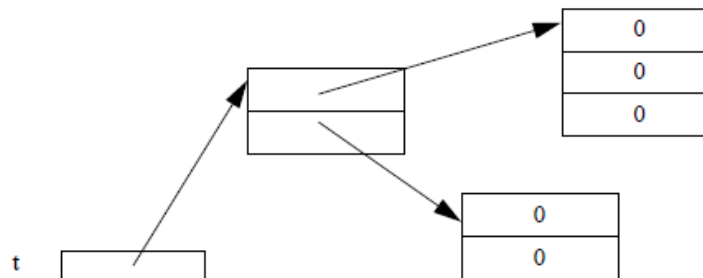


Tableau de tableaux (2)

- Initialisation (Exemple 2)

```
int t[][] = new int [2] [] ;  
// création d'un tableau de 2 tableaux d'entiers  
int [] t1 = new int [3] ;  
// t1 = référence à un tableau de 3 entiers  
int [] t2 = new int [2] ;  
// t2 = référence à un tableau de 2 entiers  
t[0] = t1 ; t[1] = t2 ;  
// on range ces deux références dans t
```

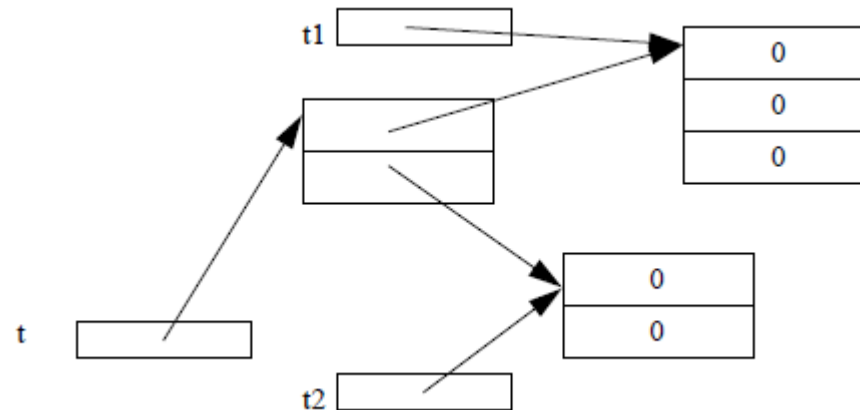


Tableau de tableaux (3)

- Exercice :
 - Soit la classe TestTableauDeTableaux suivante :

```
public class TestTableauDeTableaux {  
    public static void main(String[] args) {  
        int t[][]={{1, 2, 3},{11, 12},{21, 22, 23, 24}};  
        System.out.println ("====> t avant raz : ") ;  
        Util.affiche(t) ;  
        Util.raz(t) ;  
        System.out.println ("====> t apres raz : ") ;  
        Util.affiche(t) ;  
    }  
}
```
 - Donnez le code de la classe **Util** qui comporte deux méthodes :
 - **void raz(int[][])** qui met les valeurs d'un tableau de tableaux à zéro
 - **void affiche(int[][])** qui affiche les valeurs de chaque tableau du tableau t

CHAINES DE CARACTÈRES ET TYPES ÉNUMÉRÉS



Chaînes de caractères (1)

- Déclaration d'une référence à une chaîne
 - `String ch;`
- Initialisation avec une constante :
 - `ch = "Bonjour";`
- Constructeurs de la classe String :
 - `String ch1 = new String ();`
`//ch1 fait référence à une chaîne vide`
 - `String ch2 = new String("hello");`
`//ch2 contient la référence à une chaîne`
`//contenant "hello"`
 - `String ch3 = new String(ch2);`
`// ch3 contient la référence à une chaîne`
`// copie de ch2, donc contenant "hello"`

Chaînes de caractères (2)

- Quelques méthodes de la classe String
 - Longueur d'une chaîne
 - `ch.length()` ;
 - Comparaison de chaînes
 - `ch1.equals(ch2)` ;
 - Comparaison de chaînes sans sensibilité à la casse
 - `ch1.equalsIgnoreCase(ch2)` ;
 - Conversion en majuscule ou en minuscule
 - `ch.toLowerCase()` ;
 - `ch.toUpperCase()` ;

Chaînes de caractères (3)

- Quelques méthodes de la classe String (suite)
 - Suppression des espaces du début et du fin
 - `String ch.trim();`
 - Comparaison de deux chaînes
 - `int ch.compareTo(String Ch2)`
 - Comparaison de deux chaînes en ignorant la casse
 - `int ch.compareToIgnoreCase(String Ch2)`
 - Concaténation de deux chaînes
 - `String ch.concat(String Ch2)`

Chaînes de caractères (4)

- Quelques méthodes de la classe String (suite)
 - Extraire une sous chaîne à partir d'une position
 - `String ch.substring(int pos);`
 - Extraire une sous-chaîne à partir d'une position en indiquant la position à partir de laquelle on s'arrête
 - `String ch.substring(int pos, int pos2);`
 - Récupérer un caractère à une position donnée
 - `char ch.charAt(int pos);`

Chaînes de caractères (5)

- Quelques méthodes de la classe String (suite)
 - Trouver l'emplacement du premier caractère d'une sous-chaîne dans une chaîne ou -1
 - `int ch.indexOf(String ch2);`
 - Trouver l'emplacement du premier caractère d'une sous-chaîne dans une chaîne à partir de la position données ou -1
 - `int ch.indexOf(String ch2, int pos);`

Chaînes de caractères (4)

- Arguments de la méthode main()

- Exemple :

```
public class ArgMain{  
    public static void main (String args[]){  
        int nbArgs = args.length ;  
        if(nbArgs==0) System.out.println("pas d'arguments");  
        else{  
            for (int i=0 ; i<nbArgs ; i++)  
                System.out.println("argument "+(i+1)+"="+args[i]);  
        }  
    }  
}
```

- Exemple d'appel : `java ArgMain test essai`

- Résultat affiché

- argument 1 = test
 - argument 2 = essai

Classe StringBuilder (1)

- Motivation :
 - Certains programmes manipulant intensivement des chaînes, la perte de temps qui en résulte peut devenir gênante
 - La modification d'une chaîne crée une nouvelle chaîne : les chaînes de caractères sont immutables !
- Présentation
 - Chaîne de caractères modifiable depuis Java 5
 - C'est l'équivalent du **StringBuffer** qui est *thread safe* (donc moins performante)

Classe StringBuilder (2)

- Quelques méthodes :
 - ***setCharAt()*** : modification d'un caractère de rang donné
 - ***charAt()*** : accès à un caractère de rang donné
 - ***append()*** : ajout d'une chaîne en fin (*accepte des arguments de tout type primitif et de type String*)
 - ***insert()*** : insertion d'une chaîne à un emplacement donné
 - ***replace()*** : remplacement d'une partie par une chaîne donnée
 - ***toString()*** : conversion de *StringBuilder* en *String*

Classe StringBuilder (3)

- Exercice :

```
public class TestStringBuilder {  
    public static void main(String args[]) {  
        String ch = "la java";  
        StringBuilder chBuf = new StringBuilder(ch);  
        System.out.println(chBuf);  
        chBuf.setCharAt(3, 'J');  
        System.out.println(chBuf);  
        chBuf.setCharAt(1, 'e');  
        System.out.println(chBuf);  
        chBuf.append(" 2");  
        System.out.println(chBuf);  
        chBuf.insert(3, "langage ");  
        System.out.println(chBuf);  
    }  
}
```

Donnez le résultat de l'exécution.

Type énuméré (1)

Présentation

- Introduit par le JDK 5.0
- Permet de déclarer une variable dont les valeurs sont restreints à un nombre prédéfini d'objets.
- Toutes les énumérations héritent de la classe `java.lang.Enum` et **encapsulent** des objets qui partagent les éventuelles méthodes définies dans l'énumération
- Chaque **objet** de l'énumération est **public** et **static**

Exemple

- Déclaration

```
enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,  
            DIMANCHE }
```

ou

```
enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,  
            DIMANCHE };
```

- Utilisation

```
Jour j;  
j = Jour.MERCREDI ;
```

Type énuméré (2)

Modificateurs d'accès autorisés d'une énumération :

- public et default

Création :

- Dans un fichier séparé (.java)
- Dans un fichier contenant d'autres classes (avec le modificateur d'accès default)
- Dans une classe (comme les classes internes)

Type énuméré (3)

- Quelques opérations :
 - Egalité : `==` ou `equals()`
 - Comparaison : `compareTo()`
 - selon l'ordre de déclaration (retourne un entier positif – définie après la valeur donnée en paramètre- ou négatif ou nul)
 - Ordre : `ordinal()`
 - retourne un entier (0 pour la première constante)
 - Conversion en une chaîne : `toString()`
 - Conversion des valeurs en un tableau : `values()`

Types énumérés (4)

- Il est possible de définir des constructeurs (implicitement **privés**) et des méthodes
- Exemple :

```
enum Pays {  
    TUNISIE("tn"), FRANCE("fr"), ALLEMAGNE("aL");  
    private String abrev;  
  
    private Pays(String a){  
        this.abrev = a;  
    }  
    public String getAbreviation() {  
        return this.abrev;  
    }  
}
```

Types énumérés (5)

```
public class TestEnumeration {  
    static Pays p;  
    public static void main(String[] args) {  
        // Afficher la liste des pays via values()  
        for(Pays p:Pays.values())  
            System.out.println(p);  
        // Utiliser des méthodes et ordinal()  
        for(Pays p:Pays.values())  
            System.out.println(p.ordinal()+"-"+p+"("+p.getAbreviation()+")");  
        // Utiliser valueOf()  
        p = Pays.valueOf("TUNISIE");  
        System.out.println(tunisie);  
        // Utiliser dans switch  
        switch (p) {  
            Case TUNISIE: System.out.println("Mon pays"); break;  
            default: System.out.println("Ce n'est pas mon pays"); }  
    }  
}
```

Types énumérés (6)

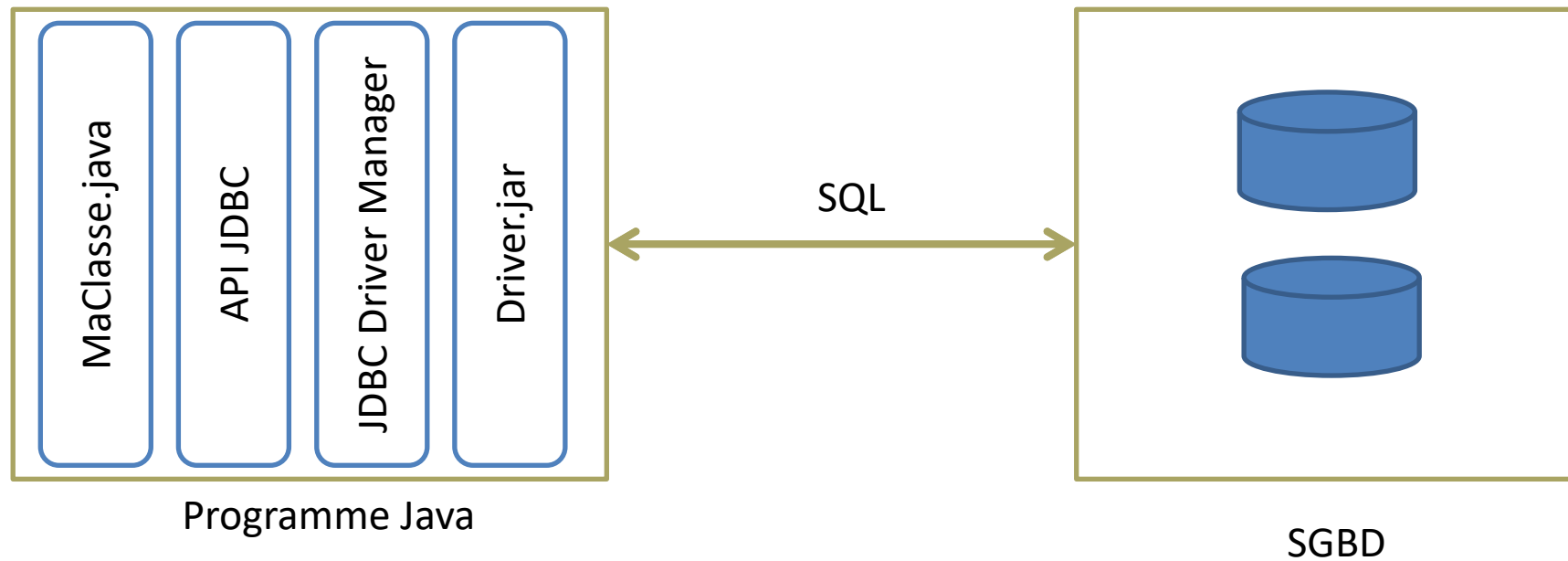
- **Exercice :**
 - Définissez un type énuméré nommé ***Mois*** permettant de représenter les douze mois de l'année, en utilisant les noms usuels (***janvier, fevrier, mars...***) tout en associant à chacun le nombre de jours correspondants (on ne tiendra pas compte des années bissextiles) et son nom en Anglais.
 - Écrivez un petit programme affichant ces différents noms avec le nombre de jours correspondants :
 - Mois 1 : janvier (January) - 31 jours**
 - Mois 2 : fevrier (February) - 28 jours**
 -**
 - Mois 12 : decembre (December) - 31 jours**



COMMUNICATION AVEC LES BD

Introduction

- Objectif
 - Interagir avec une BD



JDBC

- **Java DataBase Connectivity**
- Intergiciel (middleware) permettant aux programmes Java d'utiliser une source de données.
- API de la plateforme Java SE
- Fonctionnalités :
 - Établir une connexion avec un SGBD
 - Accéder à une BD
 - Exécuter des instructions CRUD
 - Supporter les transactions

Étapes à suivre

- **Étape 1 :**
 - Charger le *Driver* du SGBD
- **Étape 2 :**
 - Établir une *connexion* avec la BD
- **Étape 3 :**
 - Création d'un *Statement* ou d'un *PreparedStatement* pour exécuter l'ordre SQL
- **Étape 4 :**
 - Gérer les *résultats* d'un ordre SQL
- **Étape 5 :**
 - *Libération* des ressources

Charger le Driver du SGBD

```
/* Chargement du driver JDBC */  
try {  
    Class.forName( "<nom du driver>" );  
} catch ( ClassNotFoundException e ) {...}
```

- Exemple avec MySQL

```
/* Chargement du driver JDBC pour MySQL */  
try {  
    Class.forName( "com.mysql.jdbc.Driver" );  
} catch ( ClassNotFoundException e ) {  
    System.err.println("Echec du chargement du driver");  
}
```

Étapes à suivre

- **Étape 1** : Charger le Driver du SGBD
- **Étape 2** : Établir une connexion avec la BD
- **Étape 3** : Création d'un *Statement* ou d'un *PreparedStatement* pour exécuter l'ordre SQL
- **Étape 4** : Gérer les résultats d'un ordre SQL
- **Étape 5** : Libération des ressources

Établir une connexion avec une BD

```
private Connection connexion = null;
...
try {
    connexion = DriverManager.getConnection( "<URL>",
    "<USER>", "<PSW>" );
} catch ( SQLException e ) {...}
```

- Exemple avec MySQL

```
private Connection connexion = null;
...
try {
    connexion =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/maB
    aseDeDonnees", "root", "" );
} catch ( SQLException e ) { ... }
```

Étapes à suivre

- **Étape 1** : Charger le Driver du SGBD
- **Étape 2** : Établir une connexion avec la BD
- **Étape 3** : Création d'un *Statement* ou d'un *PreparedStatement* pour exécuter l'ordre SQL
- **Étape 4** : Gérer les résultats d'un ordre SQL
- **Étape 5** : Libération des ressources

Statement

- L'instance de *java.sql.**Connection*** dispose d'une méthode permettant de créer un objet *java.sql.**Statement*** :
 - Statement **createStatement()**
- Chaque objet *java.sql.**Statement*** dispose de méthode permettant d'exécuter un ordre SQL statique et retourner un ensemble de résultats :
 - ResultSet **executeQuery(String)**

Statement

- Exemple :



The screenshot shows a database management interface with a breadcrumb path: localhost > livres > livre. The 'livre' table is selected and highlighted with a red circle. Below the breadcrumb, there is a toolbar with icons for 'Afficher', 'Structure', 'SQL', 'Rechercher', 'Insérer', 'Exporter', 'Importer', and a 'plus' dropdown. Below the toolbar is a table with the following columns: #, Colonne, Type, Interclassement, Attributs, Null, Défaut, Extra, and Action. The table contains four rows of data for the 'livre' table.

#	Colonne	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
1	id_livre	int(11)			Non	Aucune	AUTO_INCREMENT	Modifier Supprimer plus ▼
2	titre	varchar(50) utf8_bin			Non	Aucune		Modifier Supprimer plus ▼
3	prix	float			Non	Aucune		Modifier Supprimer plus ▼
4	id_categorie	int(11)			Non	Aucune		Modifier Supprimer plus ▼

```
String requete="SELECT titre FROM livre";
Statement st=null;
ResultSet rs=null;
...
try{
    st=connexion.createStatement();
    rs = st.executeQuery(requete);
}catch(SQLException e){ ... }
```

PreparedStatement

- L'instance de *java.sql.**Connection*** dispose d'une méthode permettant de créer un objet de type PreparedStatement :
 - PreparedStatement **prepareStatement**(String)
- Chaque Objet implémentant cette interface dispose de méthodes permettant d'exécuter un ordre SQL précompilé, paramétré et protégé :
 - **executeQuery()**, **executeUpdate()**, etc.

PreparedStatement

- Exemples :

```
PreparedStatement ps = null; ResultSet rs=null;
...
ps = connexion.prepareStatement("SELECT titre, prix FROM
livre WHERE id_livre= ? OR titre like ?");
ps.setInt(1, param_id_livre); ps.setString(2, param_titre);
rs=ps.executeQuery();
...
```

```
PreparedStatement ps = null;
...
ps = connexion.prepareStatement("INSERT INTO Livre(titre,
prix, id_categorie) values(?,?,?)");
/* initialisation des paramètres de la requête */
ps.setString(1, titre); ps.setDouble(2, prix);
ps.setInt(3, id_categorie);
ps.executeUpdate();
```


Étapes à suivre

- **Étape 1** : Charger le Driver du SGBD
- **Étape 2** : Établir une connexion avec la BD
- **Étape 3** : Création d'un *Statement* ou d'un *PreparedStatement* pour exécuter l'ordre SQL
- **Étape 4** : Gérer les résultats d'un ordre SQL
- **Étape 5** : Libération des ressources

ResultSet

- Le résultat d'une requête est stocké dans un objet qui implémente l'interface *java.sql.ResultSet*
- L'interface **ResultSet** définit des méthodes permettant de traiter les résultats obtenus
- Exemple :

```
List<Livre> livres = new ArrayList<Livre>();  
/* resultat : un ResultSet retourné par une requête */  
while ( resultat.next() ) {  
    Livre livre=new Livre();  
    livre.setId_livre(resultat.getInt( "id_livre" ));  
    livre.setTitre(resultat.getString( "titre" ));  
    livre.setPrix(resultat.getDouble( 3 )); // "prix"  
    livre.setId_categorie(resultat.getInt("id_categorie" ));  
    livres.add(livre);  
}
```

ResultSet

- Il est possible de modifier les données du ResultSet via des méthode de la forme :
 - `updateXxxx(String, Xxx)`
 - Exemple : `updateDouble ("prix", 50.5);`
- Pour annuler les modifications par les méthodes `updateXxxx(String, Xxx)` on peut utiliser la méthode :
 - `cancelRowUpdates()`
- Pour propager les modifications vers la BD il faut valider les changement via la méthode :
 - `updateRow()`

Transactions

- `Java.sql.Connection.setAutoCommit(boolean)`
 - Permet de rendre la validation des commandes SQL automatique (true - comportement par défaut-) ou pas (false).
- `Java.sql.Connection.commit()`*
 - Permet de valider les changement depuis le dernier Commit/Rollback effectué
- `Java.sql.Connection.rollback()`*
 - Permet d'annuler les changement depuis le dernier Commit/Rollback effectué

(*) Si l'auto-commit est à false

Transactions

- Exemple :

```
public void transactionEntreComptes() {  
    try {  
        // Désactivation de l'autocommit  
        connexion.setAutoCommit(false);  
        // Retirer la somme du premier compte  
        s = connexion.createStatement();  
        s.executeUpdate("UPDATE Compte SET solde=solde-  
100 WHERE id_compte = 1");  
        // Verser la somme dans le deuxième compte  
        s.executeUpdate("UPDATE Compte SET  
solde=solde+100 WHERE id_compte = 2");  
        // Validation  
        connexion.commit();  
    } catch (SQLException e) {  
        System.err.println(e.getMessage());  
    }  
}
```

Étapes à suivre

- **Étape 1** : Charger le Driver du SGBD
- **Étape 2** : Établir une connexion avec la BD
- **Étape 3** : Création d'un *Statement* ou d'un *PreparedStatement* pour exécuter l'ordre SQL
- **Étape 4** : Gérer les résultats d'un ordre SQL
- **Étape 5** : Libération des ressources

Libération des ressources (1/2)

```
if ( resultSet != null ) {  
    try { resultSet.close();  
    } catch ( SQLException e ) {  
        /* Traiter les erreurs éventuelles ici. */  
    }  
if ( statement != null ) {  
    try { statement.close();  
    } catch ( SQLException e ) {  
        /* Traiter les erreurs éventuelles ici. */  
    }  
if ( connexion != null ) {  
    try { connexion.close();  
    } catch ( SQLException e ) {  
        /* Traiter les erreurs éventuelles ici. */  
    }  
}
```

Libération des ressources (2/2)

- A partir du Java 7 :

```
try (Connection c=  
    DriverManager.getConnection(URL, USER, PSW);  
    Statement s= connection.createStatement()) {  
    ...  
}  
catch(SQLException e) {  
    e.printStackTrace();  
}
```


Résumé

- **Ajouter le driver** du SGBD au classpath du projet
- **Charger le driver**
- Établir une **connexion** avec la BD
- Récupérer un objet permettant l'exécution des requêtes SQL (**Statement** ou **PreparedStatement**)
- Exécuter la **requête** et récupérer le **résultat**
- **Traiter** les données retournées
- **Libérer** les ressources



PROGRAMMATION GÉNÉRIQUE

Introduction

- Problème :
 - Même code qui s'applique à des Types différents
- Exemple :

MonEntier
- valeur : Integer
+ MonEntier(valeur : Integer)
+ getValeur() : Integer
+ setValeur(valeur : Integer)

MaChaîne
- valeur : String
+ MonPoint(valeur :) : String
+ getValeur() : String
+ setValeur(valeur : String)

MonPoint
- valeur : Point
+ MonPoint(valeur : Point)
+ getValeur() : Point
+ setValeur(valeur : Point)

Introduction

- Première solution :
 - Généraliser le code

- Exemple :

MonObject
- valeur : Object
+ MonObjet(valeur : Object)
+ getValeur() : Object
+ setValeur(valeur : Object)

- Problème :
 - Contrôler les types des objets

Introduction

- Objectif :
 - Écrire un code source unique utilisable avec des objets ou des variables de types quelconques
- Mécanismes
 - Héritage : tous les objets héritent de la classe Object
 - Paramètres de types : type fixé au moment de l'instanciation (depuis JDK 5.0)
- Application :
 - Classes et méthodes

Classe générique (1)

- Exemple : définition d'une classe génériques

```
class Couple <T> {  
    private T x, y; // les deux éléments du couple  
    public Couple(T premier, T second) {  
        x = premier;  
        y = second;  
    }  
    public String toString()  
    {  
        // x et y convertis automatiquement par toString  
        return ("Valeur 1 :"+x+"- Valeur 2 :"+y);  
    }  
    T getPremier() {  
        return x;  
    }  
}
```

Paramètre de type : Indique que la classe est générique et que dans la définition **T** indique un type quelconque

Classe générique (2)

- Exemple (suite) : utilisation

```
public class TestCouple {  
    public static void main(String args[]) {  
        Couple<Integer> ci = new Couple<Integer>(3, 5);  
        System.out.println(ci);  
        Couple<Double> cd = new Couple<Double>(2.0, 12.0);  
        System.out.println(cd);  
        Double p = cd.getPremier();  
        System.out.println("1er élément du couple = " + p);  
    }  
}
```

Classe générique (3)

- Exercice :
 - Donnez le code source de la classe couple et de la classe qui la teste sans utiliser un type générique :

```
class Couple <T> {  
    private T x, y;  
    public Couple(T premier, T second) {  
        x = premier;  
        y = second;  
    }  
    public String toString()  
    {  
        return ("Valeur 1 :"+x+"- Valeur 2 :"+y);  
    }  
    T getPremier() {  
        return x;  
    }  
}
```


Classe générique (4)

Classe générique à deux paramètres

- Exemple :

```
public class CoupleMixte <T, U> {  
    private T x;  
    private U y;  
    public CoupleMixte(T premier, U second) {  
        x = premier;  
        y = second;  
    }  
    public T getPremier() {  
        return x;  
    }  
    public void affiche() {  
        System.out.println("1er valeur:"+x+"-2ième valeur:"+y);  
    }  
}
```

Classe générique (5)

Limitations

- Interdit d'instancier un objet (ou un tableau d'objets) d'un type paramétré au sein d'une classe générique
- Exemple :

```
public class CoupleMixte<T, U> {  
    private T x;  
    ...  
    public CoupleMixte() {  
        x=new T();  
        ...  
    }  
    ...  
}
```

Classe générique (6)

Limitations (suite)

- Interdit de créer une classe générique qui hérite de ***Throwable*** (donc de ***Exception*** ou de ***Error***)
- Interdit de lever une exception à l'aide d'un objet d'une classe générique
 - Exemple : *throw (Couple <Integer>)*
- Interdit d'intercepter une exception en se basant sur un objet d'une classe générique
- Un champs statique ne peut pas être d'un type paramétré

Classe générique (7)

Remarques

- On peut imposer que le type générique doit hériter d'une **classe** ou implémenter une **interface**
 - Exemple : *class Couple <T **extends** Numeric>*
 - Exemple : *class Couple <T **extends** Comparable>*
- On peut combiner les deux
 - Exemple : *class Couple <T **extends** Numeric **&** Comparable>*

Exercice

- Écrire une classe générique *Triplet* permettant de manipuler des triplets d'objets d'un même type. On la dotera :
 - d'un constructeur à trois arguments (les objets constituant le triplet),
 - de trois méthodes d'accès *getPremier()*, *getSecond()* et *getTroisieme()*, permettant d'obtenir la référence de l'un des éléments du triplet,
 - Et elle redéfinit la méthode *toString()*.
- Écrire une classe pour tester la classe générique

Méthode générique (1)

Présentation

- Une méthode générique ne fixe pas le type d'au moins un paramètre
- Elle peut exister dans une classe non générique

Méthode générique (2)

- Exemple :

```
/* Méthode statique permettant de tirer au
hasard un élément d'un tableau fourni en
argument, de type quelconque. */
static <T> T hasard(T[] valeurs) {
    int n = valeurs.length;
    if (n == 0)
        return null;
    int i = (int) (n * Math.random());
    return valeurs[i];
}
```

Méthode générique (3)

- Exemple (suite) :

```
public static void main(String args[]) {  
    Integer[] tabI={ 1, 5, 8, 4, 9 };  
    System.out.println("hasard sur tabI = " +  
        TestMethodeGenerique.hasard(tabI));  
    String[] tabS={"bonjour","salut","hello"};  
    System.out.println("hasard sur tabS = " +  
        TestMethodeGenerique.hasard(tabS));  
}
```


Méthode générique (4)

- Exercice :
 - Ecrire une méthode générique déterminant le plus grand élément d'un tableau.
 - La comparaison des éléments se fait en utilisant l'ordre induit par la méthode ***compareTo()*** de la classe des éléments du tableau.

Méthode générique (5)

Wild Card : ?

- On peut imposer que le type générique utilisé dans une méthode est un sous type d'un Type et qu'il est utilisé en **consultation** :
 - Exemple : *public void maMethode(<? **extends** Point> x){...}*
- On peut imposer que le type générique utilisé dans une méthode est un super type d'un type :
 - Exemple : *public void maMethode(<? **super** PointColore> x){...}*

Arguments variables (1)

- Il est possible de définir une méthode dont le nombre d'arguments est variable (depuis JDK 5.0)
- Les arguments représentés par l'ellipse (...) sont traités comme un tableau
 - Exemple :

```
class TestEllipse{
    static int somme (int ... valeurs){
        int s = 0 ;
        for (int v : valeurs) s += v ;
        return s ;
    }
}

public class TestEllipse {
    public static void main(String[] args) {
        int[] t= {2,8};
        System.out.println(TestEllipse.somme(t));
        System.out.println(TestEllipse.somme(1,2,3));
        System.out.println(TestEllipse.somme());
    }
}
```

Arguments variables (2)

- Quelques règles
 - Dans une méthode comportant plusieurs paramètres **un seul** peut être un ellipse et il doit figurer en **fin de liste**.
 - Il n'est pas possible de **surcharger** une méthode dont un paramètre est un ellipse par une méthode dans le paramètre ellipse est remplacé par un tableau de même type

- Exemple

```
void test(int x, int ...v){...}
```

```
void test(int x, int[] t){...}
```

Arguments variables (3)

- Remarque

- Il est possible de surcharger des méthodes comme suit :

```
void test(int x, int ...v){...}
```

```
void test(int x, int y){...}
```

- Lors d'un appel de la fonction test, Java ignore tout d'abord les méthodes à ellipse
- Seulement si la recherche n'a pas abouti alors on fait intervenir les méthodes à ellipse

Arguments variables (4)

- Exercice
 - Soient les deux méthodes suivantes :
 1. `void test(int x, int ...v){...}`
 2. `void test(int x, int y){...}`
 - Donnez le numéro de la méthode appelée :

```
int i, j, k;  
test(i, j);  
test(i);  
test(i, j, k);
```
 - Soient les deux méthodes suivantes :
 1. `void f(int ...v){...};`
 2. `void f(double x, double y){...};`
 - Donnez le numéro de la méthode appelée :

```
int i, j;  
f(i, j);
```



COLLECTIONS

Introduction (1)

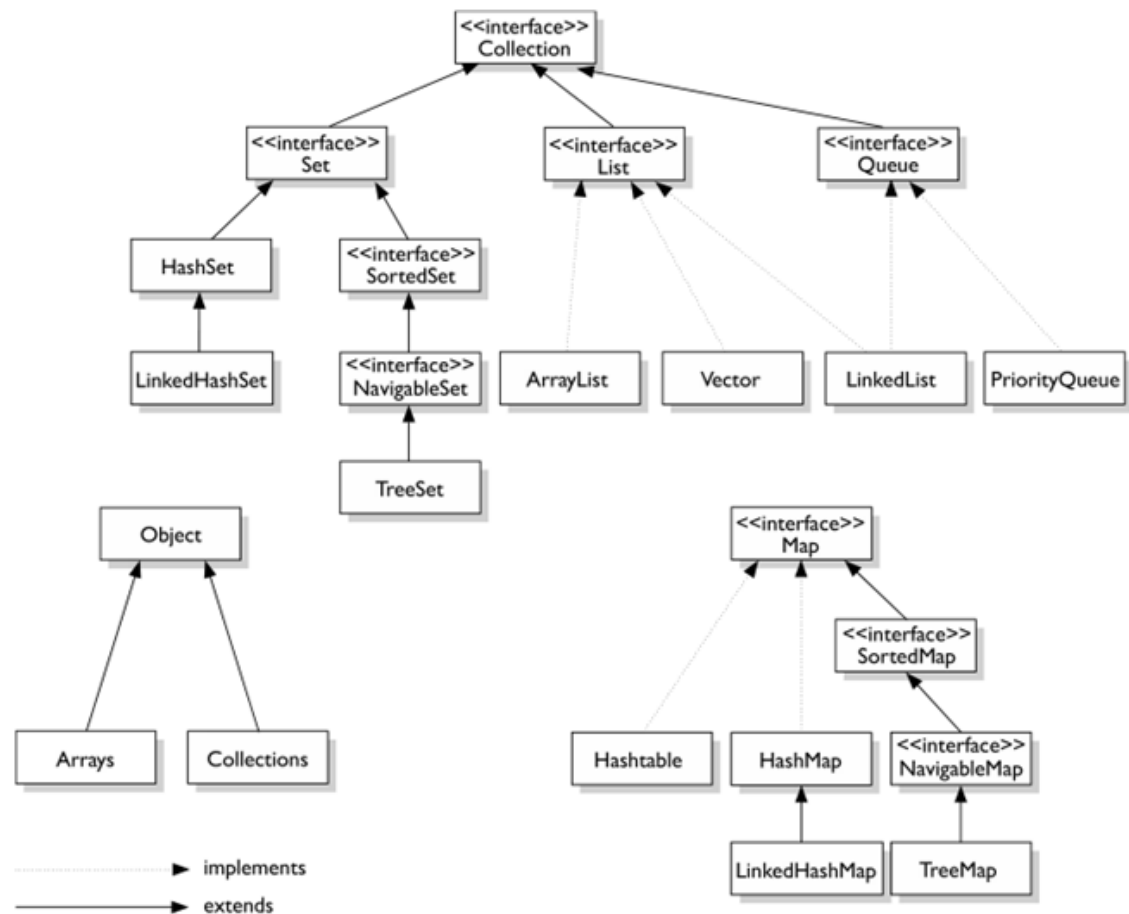
- Objectifs :
 - Mettre plusieurs objets dans une même structure de données sans avoir des restrictions sur la taille (vs. Tableaux)
 - Bénéficier de structures de données variées pour répondre aux besoins des programmeurs

Introduction (2)

- Types de collections (java.util) :

- List
- Set
- Queue
- Map

Classes et interfaces
génériques depuis Java 5 !



Introduction (3)

Classe	List	Set	Map	Ordonnée	Triée
ArrayList	X			OUI (index)	NON
LinkedList	X			OUI (index)	NON
Vector	X			OUI (index)	NON
HashSet		X		NON	NON
TreeSet		X		OUI	OUI
LinkedHashSet		X		OUI (ordre d'insertion ou du dernier accès)	NON
HashMap			X	NON	NON
TreeMap			X	OUI	OUI
LinkedHashMap			X	OUI (ordre d'insertion)	NON
Hashtable			X	NON	NON

Listes (1)

- Liste d'objets dont chaque élément dispose d'un index.
- Collection ordonnée mais pas triée
- Quelques méthodes de l'interface List :

boolean **add**(element)

boolean **add**(index, element)

boolean **contains**(object)

E **get**(index)

int **indexOf**(object)

Iterator **iterator**()

E **remove**(index)

boolean **remove**(object)

int **size**()

Object[] **toArray**()

Listes (2)

- Classes qui implémentent l'interface
 - **ArrayList**
 - Adéquate pour un **parcours rapide**, mais non adéquate aux modifications fréquentes
 - **Vector**
 - Version **synchronisée** (thread safe) de ArrayList, donc moins performante
 - **LinkedList**
 - Liste doublement chaînée, qui permet la **modification rapide** des éléments mais avec un parcours plus lent

Listes (3)

– Exemple

```
public class TestArrayList {  
    public static void main(String[] args) {  
        List<Point> points= new ArrayList<Point>();  
        Point p = new Point(3,3);  
        points.add(p);  
        points.add(new Point(1,1));  
        points.add(new Point(2,2));  
        //Parcours de la liste  
        for(Point p : points)  
            System.out.println(p.getAbscisse());  
        //Autres manipulations  
        System.out.println("Taille : "+points.size());  
        System.out.println("Deuxième point : "+points.get(1));  
        System.out.println("Index de p : "+points.indexOf(p));  
        points.remove(1);  
        Object[] t= points.toArray();  
        for(Object o : t)  
            System.out.print(((Point)o).getAbscisse());  
    }  
}
```

Ensembles (1)

- Il s'agit d'un ensemble d'objets sans redondance
- Classes implémentant l'interface **Set**
 - **HashSet** : Ensemble non ordonné (pas d'ordre de parcours) et non trié, dont les performances dépendent de l'implémentation de ***hashCode()***
 - **LinkedHashSet** : Ensemble ordonné (selon l'ordre d'insertion) mais non trié
 - **TreeSet** : Ensemble ordonné et trié dans l'ordre ascendant

Ensembles (2)

- Quelques méthodes de l'interface Set :

boolean **add**(element)

boolean **contains**(object)

Iterator **iterator**()

boolean **remove**(object)

int **size**()

Object[] **toArray**()

Ensembles (3)

- L'unicité des éléments dans un ensemble dépend de l'implémentation de la méthode : ***equals()***
- *Il faut rappeler que equals() doit être :*
 - *Réflexive*
 - *Symétrique*
 - *Transitive*
 - *Consistante*

Ensembles (4)

- Classe **HashSet**
 - Non ordonné :
 - On ne peut pas prédire l'ordre des éléments lors d'un parcours de l'ensemble
 - Performance :
 - Dépend de l'implémentation de la méthode ***hashCode()***

Ensembles (5)

- **HashCode**

- Idée :

- produire une valeur (Hash Code) pour chaque Objet qui va permettre de le classer dans un sous ensembles identifié par cette valeur.

- Intérêt :

- Lors de la recherche ou de l'insertion d'un Objet, il n'est plus besoin de parcourir tout l'ensemble, mais de vérifier le sous-ensemble identifié par le Hash Code de cet Objet.

Ensembles (6)

- ***hashCode()***

- Méthode de la classe **Object** tel que :

- Deux objets égaux doivent avoir un même Hash Code
 - La valeur retournée ne doit pas être modifiée, durant l'exécution du programme, tant qu'il n'y a pas eu de modifications qui influencent le calcul de l'égalité

- Signature :

- public int hashCode()*

Ensembles (7)

- Implémentation :
 - Idéalement la méthode ***hashCode()*** doit produire des entiers permettant de distribuer équitablement un ensemble d'objets en des sous-ensembles
 - Exemple inefficace mais légal :

```
public int hashCode() { return 5; }
```

Ensembles (8)

- Soit la classe Point :

```
class Point{
    private int abs, ord;
    Point(int x, int y){
        this.abs=x;
        this.ord=y;}
    public int getAbs() { return abs; }
    public int getOrd() { return ord; }
    @Override
    public String toString(){
        return "Point (" + this.abs + ", " + this.ord + ")";
    }
}
```

- Donnez le code de la classe **TestHashSet** qui crée un **HashSet** avec deux points ayant les même coordonnées et un troisième différent, puis on parcourt l'ensemble pour afficher son contenu

Ensembles (9)

- Donnez le résultat de l'exécution.
- Redéfinissez la méthode ***equals()*** afin de considérer que deux objets ayant les même coordonnées sont égaux
- Donnez le résultat de l'exécution après modifications
- Expliquez ce résultat et fait les modifications nécessaires pour avoir un résultat cohérent

Ensembles (10)

- Donnez le code source d'un programme Java équivalent, mais qui utilise un **TreeSet** pour stocker les points.
- Donnez le résultat de l'exécution

Tableaux associatifs (1)

- Ensemble de couples <clé, valeur>, tel que la clé est unique au sein d'un Map.
- Classes implémentant l'interface Map :
 - **HashMap** : non ordonné et non trié, rapide en modification
 - **Hashtable** : comme le HashMap mais avec des méthodes *synchronisées* et l'interdiction d'avoir **Null** comme clé ou comme valeur.
 - **LinkedHashMap** : *ordonné* mais non trié, parcours rapide
 - **TreeMap** : ordonné et *trié* (selon les clés)

Tableaux associatifs (2)

- Pour le bon fonctionnement d'une table associative la clé doit redéfinir les méthodes **equals()** et **hashCode()**
- Quelques méthodes de l'interface **Map** :
 - put**(key, value)
 - object **get**(key)
 - remove**(key)
 - boolean **containsKey**(key)
 - boolean **containsValue**(value)
 - Set **keySet**()
 - Set<Map.Entry> **entrySet**()
 - int **size**()

Tableaux associatifs (3)

- Exercice :
 - Donnez le code source d'une classe qui contient un **HashMap** de couples de chaînes de caractères. La clé est un nom de compte et la valeur est le mot de passe qui lui correspond.
 - La classe offre une méthode qui permet de vérifier si un compte est valide ou pas en lui passant un nom de compte et un mot de passe (retourne un booléen).
 - La classe contient la méthode **main()** qui remplit le **HashMap** avec 6 entrées dont deux avec la même clé. Ensuite, elle affiche le contenu du **HashMap**. Finalement, elle vérifie la validité de trois comptes : un avec des paramètres correctes, un avec un faux mot de passe, et un avec un faux nom de compte

Files d'attente (1)

- Permet de gérer les files d'attente entre un producteur et un consommateur.
- Généralement se sont des files d'attente de type FIFO ou LIFO
- Exemple de Classe implémentant l'interface **Queue** :
 - **PriorityQueue** : file d'attente triée selon la priorité (la valeur la plus petite est plus prioritaire que la valeur la plus grande)

Files d'attente (2)

- Quelques méthodes de l'interface Queue :
 - Ajouter un élément :
 - void **add** throws IllegalStateException (T t)
 - boolean **offer**(T t)
 - Retirer le premier élément :
 - E **remove()** throws NoSuchElementException ()
 - E **poll()**
 - Consulter le premier élément :
 - E **element()** throws NoSuchElementException ()
 - E **peek()**

Classe Collections

- Offre des méthodes statiques qui opèrent sur les collections
- Exemples de méthodes :

Collections.sort(liste*);

Collections.sort(liste, comparator)

Collections.binarySearch(liste*, valeur)

Collections.binarySearch(liste, valeur, comparator)

Collections.max(collection*)

Collections.min(collection*)

* Les objets doivent être comparables

Itérateurs (1)

- Présentation :
 - Objets qui permettent de parcourir les éléments d'une collection qui implémente l'interface Collection (depuis Java 5 on peut utiliser la boucle for each)
 - Pour obtenir un itérateur sur une collection ou pour réinitialiser l'itérateur : appeler la méthode ***iterator()*** de la collection
 - Deux types :
 - Monodirectionnels
 - Bidirectionnels

Itérateurs (2)

- Itérateur monodirectionnel :
 - Chaque classe collection dispose d'une méthode nommée ***iterator()*** fournissant un itérateur monodirectionnel (implémentant l'interface ***Iterator <E>***)
 - Indique la position courante (un élément de la collection ou la fin de la collection)
 - La méthode ***next()*** fournit l'objet en cours et avance
 - La méthode ***hasNext()*** permet de détecter la fin de la collection
 - La méthode ***add(element)*** : ajoute un élément à la position courante et déplace la position courante à l'élément suivant
 - La méthode ***set(element)*** : modifie l'élément courant (retourné par ***next()***)

Itérateurs (3)

- Itérateur bidirectionnel :
 - Permet le parcours de **certaines** collections dans les deux sens (tels que : listes chaînées, vecteurs dynamiques, etc.)
 - Dispose des méthodes **next()**, **hasNext()** et **remove()**.
 - Méthodes spécifiques
 - **previous()** : pointe vers le prédécesseur et retourne sa valeur
 - **hasPrevious()** : retourne vrai s'il y 'en a un prédécesseur

Itérateurs (4)

- Exemple :

```
class MaListeChaine{  
    private LinkedList<Integer> l = new LinkedList<Integer>();  
    void initialisation(){  
        for(int i=0; i<10; i++)  
            l.add(i);  
    }  
    void affichage(){  
        Iterator<Integer> iter=l.iterator();  
        while(iter.hasNext())  
            System.out.println(iter.next());  
    }  
    void affichageInverse(){  
        ListIterator<Integer> iter = l.ListIterator(l.size());  
        while(iter.hasPrevious())  
            System.out.println(iter.previous());  
    }  
}
```

En rouge : méthodes de collection
En vert : méthodes d'un itérateur
monodirectionnel
En bleu : méthodes d'un itérateur
bidirectionnel

Exercice

- Créez une classe **Livre** qui représente un livre ayant un titre, un prix et un nombre de pages.
- Créez la classe **TestLivre** qui permet de créer une collection de 6 livres. On doit afficher les livres triés selon l'ordre alphabétique des titres, puis selon le prix, puis selon le nombre de pages.



SIMPLIFICATION DU CODE

Introduction

Problème

- Certaines classes et méthodes ne sont utilisable qu'une seule fois
- Taille du code source important avec un nombre important de classes et de méthodes

Objectif

- Simplifier et minimiser la taille du code source

Solutions

- Classes anonymes
- Interfaces fonctionnelles
- Lambdas
- Références de méthodes

Classe Anonyme (1)

Objectif

- Offrir des fonctionnalités qui ne seront pas réutilisés

Exemple (création du cadre de la situation)

```
interface Salutation {  
    String getSalutation();  
}  
class Bonjour implements Salutation {  
    public String getSalutation() {return  
        "Bonjour";}  
}  
// Classes Bonsoir et BonneNuit aussi implémentent Salutations
```

Classe Anonyme (2)

Exemple (Usage des instances de type Salutation)

```
public class Personne {  
    ...  
    void saluer(Salutation s) {  
        System.out.println(  
            this.prenom+ " " +  
            this.nom +" : "+  
            s.getSalutation());  
    }  
}
```

Classe Anonyme (3)

Exemple (suite : test)

```
public class Main {  
    public static void main(String[] args)  
    {  
        Personne p = new Personne();  
        p.saluer(new Bonjour());  
        p.saluer(new Bonsoir());  
        p.saluer(new BonneNuit());  
    }  
}
```

Objectif : ajouter une nouvelle Salutation qui sera uniquement utilisée dans la méthode main()

Classe Anonyme (4)

Exemple (Usage d'une classe anonyme)

```
public class Main {  
    public static void main(String[] args) {  
        Personne p = new Personne();  
        p.saluer(new Bonjour());  
        // Classe anonyme de type Salutation  
        p.saluer(new Salutation() {  
            @Override  
            public String getSalutation() {  
                return " !السلام عليكم ";  
            }  
        });  
    }  
}
```

Remarques : ne peut pas être abstraite ou statique et ne peut pas contenir de constructeur

Interface fonctionnelle

Présentation

- Une interface qui peut traduire une fonctionnalité dont elle précise la signature introduite dans **Java 8**
- Appelée SAM (Single Abstract Method)
- Annotée par **@FunctionalInterface**
- Contient une seule méthode abstraite (peut contenir des méthodes par défaut)

Exemple

```
@FunctionalInterface
interface Salutation {
    String getSalutation();
}
```

Lambda (1)

Présentation

- Permet de redéfinir la méthode abstraite d'une classe fonctionnelle sans passer par la création d'une classe anonyme
- Permet d'avoir un code plus lisible et permet de donner une référence à la méthode

Exemple

```
public class Main {  
    public static void main(String[] args) {  
        Personne p = new Personne();  
        p.saluer(new Bonjour());  
        // Expression Lambda pour une nouvelle salutation  
        Salutation slt = () -> "السلام عليكم";  
        p.saluer(slt);  
    }  
}  
  
// Classe anonyme équivalente  
p.saluer(new Salutation() {  
    @Override  
    public String getSalutation() {  
        return "السلام عليكم";  
    }  
});
```

Lambda (2)

Syntaxe

() -> instruction;

- Si l'instruction est une expression alors sa valeur est retournée

(paramètres effectifs) -> instruction;

- Si l'instruction est une expression alors sa valeur est retournée

**(paramètres effectifs) -> {
 instructions;
 return ... ;
};**

- Permet d'exécuter plusieurs instructions et retourner éventuellement une valeur

Package java.util.function (1)

Présentation

- Introduit à partir de **Java 8**
- Contient des interfaces fonctionnelles permettant d'éviter la création d'interfaces fonctionnelles pour les tâches courants
- Les interfaces offertes peuvent contenir des méthodes par défaut
- Exemples :
 - **Function** <T, R> qui offre **R apply(T t)**
 - **Predicate** <T> qui offre **boolean test(T t)**
 - **Consumer** <T> qui offre **void accept(T t)**
 - **Supplier** <T> qui offre **T get()**
 - **BinaryOperation** <T> qui offre **T apply(T t1, T t2)**
 - Il existe des dérivés plus spécifiques : **IntFunction**, **IntSupplier**, etc.

Package java.util.function (2)

Exemple

Donnez le code d'une Lambda permettant de générer l'acronyme d'une personne à partir d'un objet `Personne` ayant des accesseurs pour lire le Nom et le Prénom.

```
public class Main {  
    public static void main(String[] args) {  
        Personne p = new Personne("Farhat", "Ramzi");  
        /* Expression Lambda pour retourner l'acronyme  
d'une personne */  
        Function<Personne, String> acronyme = (per) ->  
            per.getPrenom().charAt(0)+  
            "."+  
            per.getNom().charAt(0)+  
            ".";  
        System.out.println(acronyme.apply(p));  
    }  
}
```

Référence de méthode

Principe

Définir une référence pour une méthode d'une Interface ou d'une classe (méthode statique, méthode membre ou constructeur).

Syntaxe

<Interface ou Classe> :: <nom de la méthode>

Exemple

```
/* Référence de méthode pour println() de  
System.out */  
Consumer<String> e = System.out::println;  
e.accept("Acronyme : "+acronyme.apply(p));
```



LES THREADS

Introduction

- Idée de base :
 - Système ***multi-tâche*** : exécution concurrente (simultanée d'un point de vue utilisateur) de plusieurs tâches (processus).
- Originalité de Java :
 - Programme ***multi-threads*** : le programme est constitué de plusieurs threads qui s'exécutent en concurrence et qui peuvent communiquer entre eux.

Introduction

- Un **Thread** est un objet Java.
- Un **thread** d'exécution est un processus individuel qui a sa propre pile d'exécution.
- La méthode `main()` tourne dans un thread appelé **thread main**
- L'**ordonnancement** des threads est assuré par la JVM
- Deux types de threads : **utilisateur** (la fin de leur exécution donne fin au programme) et **daemon**

Créer un Thread

- Définir et instancier un Thread : deux approches
 - Hériter de la classe **Thread**
 - Implémenter l'interface **Runnable**
- Première approche : instancier la classe **java.lang.Thread**
- Classe qui comporte des méthodes dont :
 - public void **start()**
 - C'est la méthode qui permet de créer un nouveau thread avec sa propre pile d'exécution
 - public void **run()**
 - C'est la première méthode exécuté dans un thread

Hériter de la classe Thread (1)

```
class Ecrit extends Thread {
    private String texte;
    private int nb;
    private long attente;

    public Ecrit(String texte, int nb,
        long attente) {
        this.texte = texte;
        this.nb = nb;
        this.attente = attente;
    }

    public void run() {
        try {
            for (int i = 0; i < nb; i++){
                System.out.print(texte);
                Thread.sleep(attente);
            }
        } catch (InterruptedException e){
        } // imposé par sleep
    }
}
```

```
public class TestThreads {

    public static void main(String
        args[]) {
        Ecrit e1=new Ecrit("A", 10, 5);
        Ecrit e2=new Ecrit("B", 12, 10);
        Ecrit e3=new Ecrit("C", 5, 15);
        e1.start();
        e2.start();
        e3.start();
    }
}
```

Permet de lancer l'exécution du thread : opérations nécessaires auprès du SE et de la JVM puis lancer **run()**

Permet de mettre en sommeil le thread pour un certain nombre de millisecondes : utiliser ici **optionnellement** pour garantir l'exécution des autres threads

Hériter de la classe Thread (2)

- **Remarques :**

- Si on appelle directement la méthode **run()** le programme fonctionnera mais on aura une exécution séquentielle dans le thread principal.
- La méthode **start()** doit être appelée une seule fois pour un objet thread donné au risque de déclencher l'exception **IllegalThreadStateException**.
- La méthode **sleep()** est une méthode statique de la classe Thread qui met en état de sommeil le thread courant.

Implémenter l'interface Runnable (1)

- Intérêt :
 - Utilisable au cas où la classe hérite déjà d'une autre classe (pas d'héritage multiple en Java)
- Implémentation :
 - **Runnable** comporte une seule méthode qui doit être implémentée : **run()**
- Création d'un Thread
 - Les instances de la classe qui implémentent **Runnable** ne sont pas de type **Thread** !
 - Il faut utiliser le constructeur **Thread(Runnable r)**

Implémenter l'interface Runnable (2)

Exercice : Donnez un code équivalent en utilisant l'interface Runnable

```
class Ecrit extends Thread {  
    private String texte;  
    private int nb;  
    private long attente;  
  
    public Ecrit(String texte, int nb,  
long attente) {  
        this.texte = texte;  
        this.nb = nb;  
        this.attente = attente;  
    }  
    public void run() {  
        try {  
            for (int i = 0; i < nb; i++){  
                System.out.print(texte);  
                Thread.sleep(attente);  
            }  
        } catch (InterruptedException e){  
        }  
    }  
}  
  
public class TestThreads {  
  
    public static void main(String  
args[]) {  
        Ecrit e1=new Ecrit("A ",10,5);  
        Ecrit e2=new Ecrit("B " 12 10);  
        Ecrit e3=new Ecrit("C", 5, 15);  
        e1.start();  
        e2.start();  
        e3.start();  
    }  
}
```

Interruption d'un thread (1)

- Problème :
 - Dans certain cas la méthode **run()** n'a pas de fin programmée (boucle infinie)
 - Besoin d'interrompre le thread
- Méthodes :
 - Demander à un thread **t** d'arrêter l'exécution :
t.interrupt()
 - Vérifier dans la méthode **run()** d'un thread s'il y a une demande d'arrêt :
static boolean interrupted()

Interruption d'un thread (2)

- Exercice :
 - Donnez le code de la classe **EcrireSansFin** qui permet d'afficher infiniment des * sur l'écran. Cette classe va servir pour créer un thread.
 - Donnez le code de la classe **TestEcrireSansFin** qui permet de créer et lancer un thread à base de la classe **EcrireSansFin** puis qui l'arrête après 500 ms.

Thread démon (1)

- Chaque programme s'arrête lorsque le dernier thread se termine.
- Deux types de thread:
 - Thread utilisateur
 - Thread démon (daemon en anglais)
- Particularité d'un thread démon : si à un moment donné, les seuls threads en cours d'exécution d'un même programme sont des démons, ces derniers sont arrêtés brutalement et le programme se termine.

Thread démon (2)

- Tout thread prend par défaut le même type que le thread qui l'a créé (main est un thread utilisateur)
- Cration d'un thread démon :
 - t. ***setDaemon(true)*** avant d'appeler la méthode ***t.start()***
- Exception ***InvalidThreadStateException*** se produit si :
 - on fait appel à ***setDaemon()*** après ***start()***
 - on appelle plusieurs fois ***setDaemon()***

Coordination de threads (1)

- Les threads appartiennent à un même programme :
 - concurrence sur les ressources dont le processeur
- Objectif :
 - Favoriser l'exécution de certains threads
- Solution :
 - Utiliser la méthode **setPriority(int)** pour fixer la priorité d'un thread (1 plus prioritaire 10 moins prioritaire) avant de le démarrer
 - Chaque thread hérite la priorité du thread qui la créé
 - Priorité du thread main est par défaut 5

Coordination de threads (2)

- Les threads appartiennent à un même programme :
 - concurrence sur les ressources dont le processeur
- Objectif :
 - Ne pas monopoliser le processeur
- Solution :
 - Utiliser la méthode statique **yield()** de la classe Thread pour passer de l'état « Running » à l'état « Runnable »
 - Permet de rentrer en concurrence avec les thread de même priorité sur l'usage du processeur

Coordination de threads (3)

- Les threads appartiennent à un même programme :
 - Modification concurrente de l'état des objets
- Risque :
 - Incohérence de données
- Objectif :
 - synchronisation
- Solution
 - Utilisation du modificateur (non access modifier) **synchronized**

Coordination de threads (4)

- **Exercice :**
 - Créez la classe **CompteBancaire** qui contient un attribut privé **solde** entier initialisé à 50, un **accesseur** pour récupérer la valeur du solde, et la méthode **retirer(int)** qui permet de retirer la somme donnée en paramètre du solde.

Coordination de threads (4)

- **Exercice (suite) :**
 - Créez la classe **GestionCompte** qui implémente **Runnable**. Elle contient :
 - un attribut de type **CompteBancaire**,
 - une méthode privée **faireUnRetrait(int somme)** : si la somme est suffisante écrire « nom_thread va retirer », puis mettre le thread en état de sommeil pour 500 ms, puis retirer la somme du compte, puis écrire le « nom_thread a retiré »; sinon afficher « solde insuffisant pour nom_thread ».
 - Une méthode **run()** : faire un retrait de 10, puis si le solde est négatif afficher « Compte à découvert : valeur_solde ». Répéter ceci 5 fois.
 - Une méthode **main(String[] args)** : Créer une instance de la classe, l'utiliser pour créer deux threads, appeler l'un « marie » et l'autre « femme » et démarrer les deux threads.
 - Nommer un thread : **setName(String)**
 - Nom d'un thread : **Thread.currentThread().getName()**

Coordination de threads (4)

- **Problème ?**

- Entre la vérification de la solvabilité du compte et le retrait du compte dans la méthode **faireUnRetrait(int)** un autre thread peut changer l'état du compte.

- **Solution ?**

- Empêcher l'utilisation concurrente de la méthode **faireUnRetrait(int)**

Coordination de threads (5)

- Pour rendre l'exécution d'une méthode atomique (si un thread n'est plus à l'état « Exécution » alors qu'il a commencé l'exécution de la méthode, aucun autre thread ne peut exécuter la méthode) il faut marquer la méthode comme « synchronisée ».

- Exemple :

```
private synchronized faireUnRetrait(int somme){  
    ...  
}
```

Coordination de threads (6)

- **Remarques :**

- On ne peut utiliser **synchronized** que sur des **méthodes** ou sur des **blocks**
- Chaque objet dispose d'un seul **verrou** qui est acquis par le thread qui exécute une de ses méthodes synchronisées (ou blocks synchronisés).
- Si le verrou d'un objet est acquis par un thread, aucun autre thread ne peut exécuter une des méthodes synchronisées (ou blocks synchronisés) de l'objet
- Le verrou est libéré lorsque l'exécution de la méthode synchronisée (ou block synchronisé) se termine
- Chaque classe également dispose d'un verrou pour les méthodes statiques synchronisées

Coordination de threads (7)

- Exemple de block synchronisé :

```
Class MaClasse{  
    void maMethode(){  
        System.out.println("je ne suis pas synchronisé");  
        synchronized(this){  
            System.out.println("je suis synchronisé");  
        }  
    }  
}
```

- Exercice :
 - Réécrivez le code source de la méthode **faireUnRetrait(int)** en utilisant une méthode synchronisée, puis un block synchronisé

Coordination de threads (8)

- Les threads appartiennent à un même programme : possibilité de dépendance
 - Cas de producteur/consommateur.
- Objectif :
 - synchronisation
- Solution
 - Attendre un thread jusqu'il envoie une notification

Coordination de threads (9)

- Chaque objet hérite de la classe Object dispose des méthodes suivantes (qui doivent être appelées d'un contexte synchronisé) :
 - void **wait()** throws InterruptedException
 - Mettre le thread courant dans l'état « attente » jusqu'à avoir une notification de la part de l'objet utilisé pour l'appel de wait(). Le thread, qui doit disposer d'un verrou sur l'objet, cède le verrou avant d'entrer dans l'état d'attente.
 - void **notify()** throws IllegalStateException
 - void **notifyAll()** throws IllegalStateException
 - Permettent d'indiquer au(x) thread(s) en attente de passer à l'état « Prêt »

Coordination de threads (10)

- Exemple :

```
class GestionTableau {  
    private String[] tab = new String[10];  
    private int index = 0;  
    synchronized void ajouter(String s) {  
        tab[index] = s;  
        index++;  
        notify();  
    }  
    synchronized String getPremierElement() {  
        while(index == 0) {  
            try {  
                wait();  
            } catch (InterruptedException ie) {  
                ie.printStackTrace();  
            }  
        }  
        return tab[0];  
    }  
}
```

Un seul thread peut
ajouter à la fois

Notifier un thread qui
attend pour cette objet

Récupérer le premier
élément

Entrer dans la boucle si le tableau
est vide

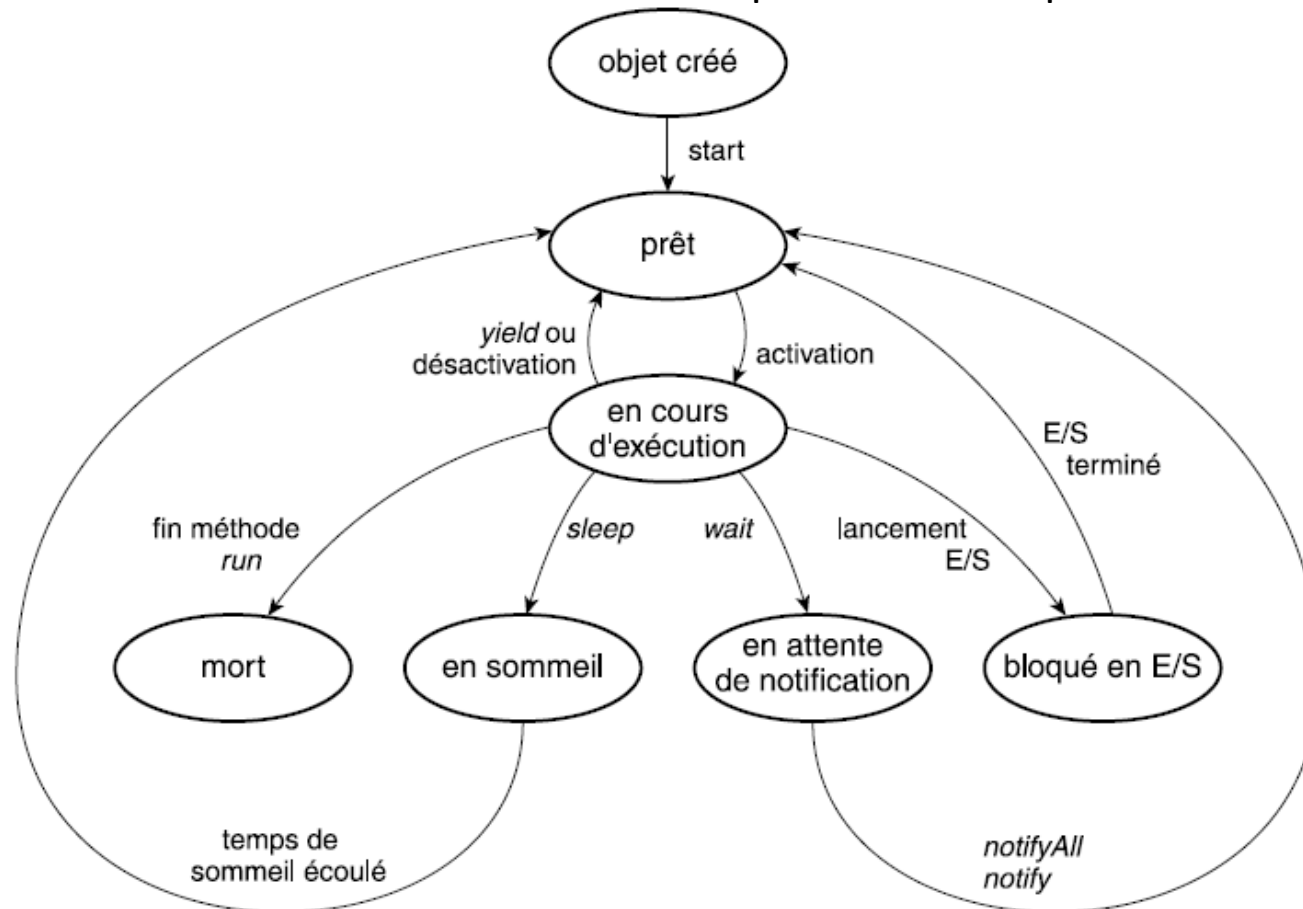
Libérer le verrou et
attendre une notification
d'un autre thread utilisant
l'objet courant

}

Différents états d'un thread

Exercice :

Donnez les états d'un Thread et les événements permettant de passer d'un état à un autre





BASES DE LA PROGRAMMATION GRAPHIQUE

Création d'une fenêtre

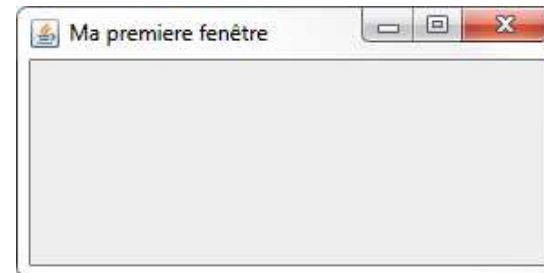
Classe JFrame

- Permet de créer une fenêtre graphique
- Appartient au paquetage javax.swing
- Constructeur par défaut : `JFrame()`
- Quelques méthodes
 - **setTitle** (String titre) // titre
 - **setSize** (int width, int height) // taille en nombre de pixels
 - **setDefaultCloseOperation**(JFrame.EXIT_ON_CLOSE)
// active la fermeture de la fenêtre
 - **setLocationRelativeTo**(null) // fenêtre au centre de l'écran
 - **setVisible** (boolean visibility) // rendre la fenêtre visible

Création d'une fenêtre (2)

- Exemple (Instanciación de **JFrame**) :

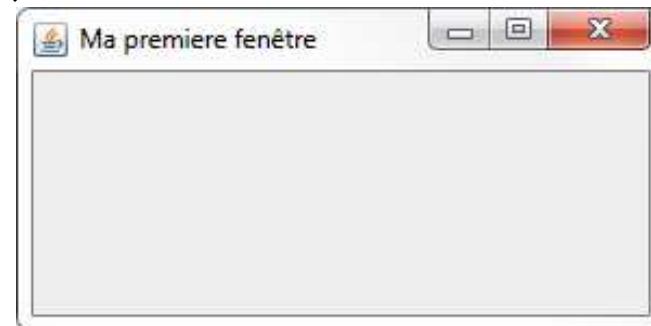
```
import javax.swing.*;
public class TestPremièreFenêtre {
    public static void main(String args[]) {
        JFrame f = new JFrame();
        // Taille de la fenêtre
        f.setSize(300, 150);
        // Titre de la fenêtre
        f.setTitle("Ma premiere fenêtre");
        // Rendre la fenêtre visible
        f.setVisible(true);
    }
}
```



Création d'une fenêtre (3)

- Exemple (héritage de JFrame) :

```
public class Fenetre extends JFrame {  
    Fenetre() {  
        this.setTitle("Ma premiere fenetre");  
        this.setSize(300, 150);  
        // Positionner la fenetre au centre  
        this.setLocationRelativeTo(null);  
        // Terminer le thread si on appui sur X  
        this.setDefaultCloseOperation(  
JFrame.EXIT_ON_CLOSE);  
        this.setVisible(true);  
    }  
}
```



Création d'une fenêtre (4)

Autres paramètres

- Positionner la fenêtre par rapport au coin supérieur gauche de l'écran selon un nombre de pixel horizontalement et verticalement

setLocation(int x, int y)

- Permettre ou interdire le redimensionnement de la fenêtre

setResizable(boolean b)

- Garder la fenêtre au premier plan

setAlwaysOnTop(boolean b)

- Afficher sans contour ni boutons

setUndecorated(boolean b)

Composantes graphiques

Deux types de composantes

- Conteneurs : composantes susceptibles de contenir d'autres éléments
 - Exemples : JFrame, JPanel, etc.
- Contrôles : composantes atomiques
 - Exemples : cases à cocher, boutons radios, étiquettes, les champs de texte, etc.

Meubler une fenêtre

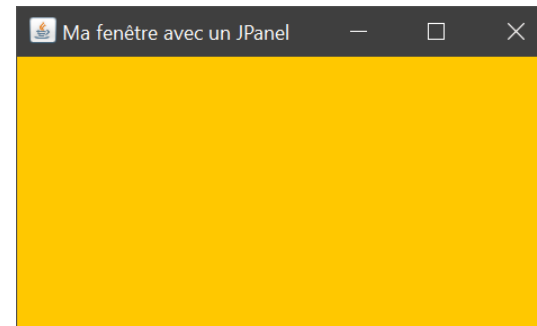
Plusieurs démarches

- Démarche 1 :
 - Utiliser un objet **JPanel** comme conteneur de la fenêtre
 - Créer un **JPanel**
 - Utiliser la méthode **setContentPane()** pour définir le JPanel comme conteneur de la fenêtre
 - Définir un **gestionnaire de mise en page** (ou utiliser celui par défaut du conteneur) pour agencer les composants
 - Créer des **composants** (boutons, listes déroulantes, etc.)
 - Ajouter les composants

Meubler une fenêtre (2)

Exemple

```
public class FenetreAvecContenu extends JFrame{
    FenetreAvecContenu(){
        this.setTitle("Ma fenêtre avec un JPanel");
        this.setSize(500, 300);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Instanciación d'un objet JPanel
        JPanel pannel = new JPanel();
        //Définition de sa couleur de fond
        pannel.setBackground(Color.ORANGE);
        // Définir le JPanel comme le contenu du JFrame
        this.setContentPane(pannel);
        this.setVisible(true);
    }
}
```



Meubler une fenêtre (3)

- On peut à tout instant dans un conteneur (JFrame ou JPanel) :
 - **Ajouter** un nouveau composant : *add()*
 - **Supprimer** un composant : *remove()*
 - **Désactiver** un composant : *setEnabled(false)*
 - **Réactiver** un composant : *setEnabled(true)*
- Forcer la prise en compte des changements
 - *revalidate ()* pour le composant
 - *validate()* pour le conteneur

Meubler une fenêtre (4)

Plusieurs démarches

- Démarche 2 :
 - Utiliser le conteneur de la fenêtre
 - Récupérer le conteneur de la fenêtre via **getContentPane()**
 - Définir un **gestionnaire de mise en page** (ou utiliser celui par défaut du conteneur) pour agencer les composants
 - Créer des **composants** (boutons, listes déroulantes, etc.)
 - Ajouter les composants

Meubler une fenêtre (5)

- Exemple d'ajout d'un bouton à une fenêtre :

```
class MaFenêtre extends JFrame{  
    MaFenêtre(){  
        this.setTitle("Tester un bouton");  
        this.setSize(400,400);  
        this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);  
        this.setVisible(true);  
  
        JButton b = new JButton("Test");  
        Container c = this.getContentPane();  
        c.add(b);  
    }  
}
```

Texte à afficher sur le bouton

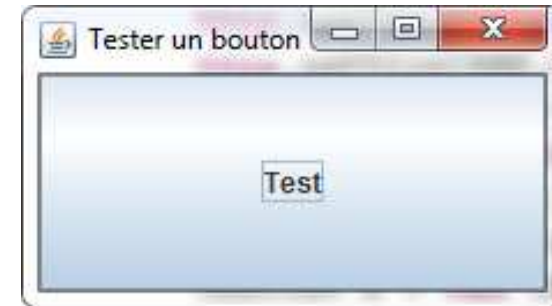
Ajout du bouton

Référence au contenu de la fenêtre.
Il faut importer java.awt.Container

Gestionnaire de mise en page

Remarque

- le bouton occupe toute la fenêtre !



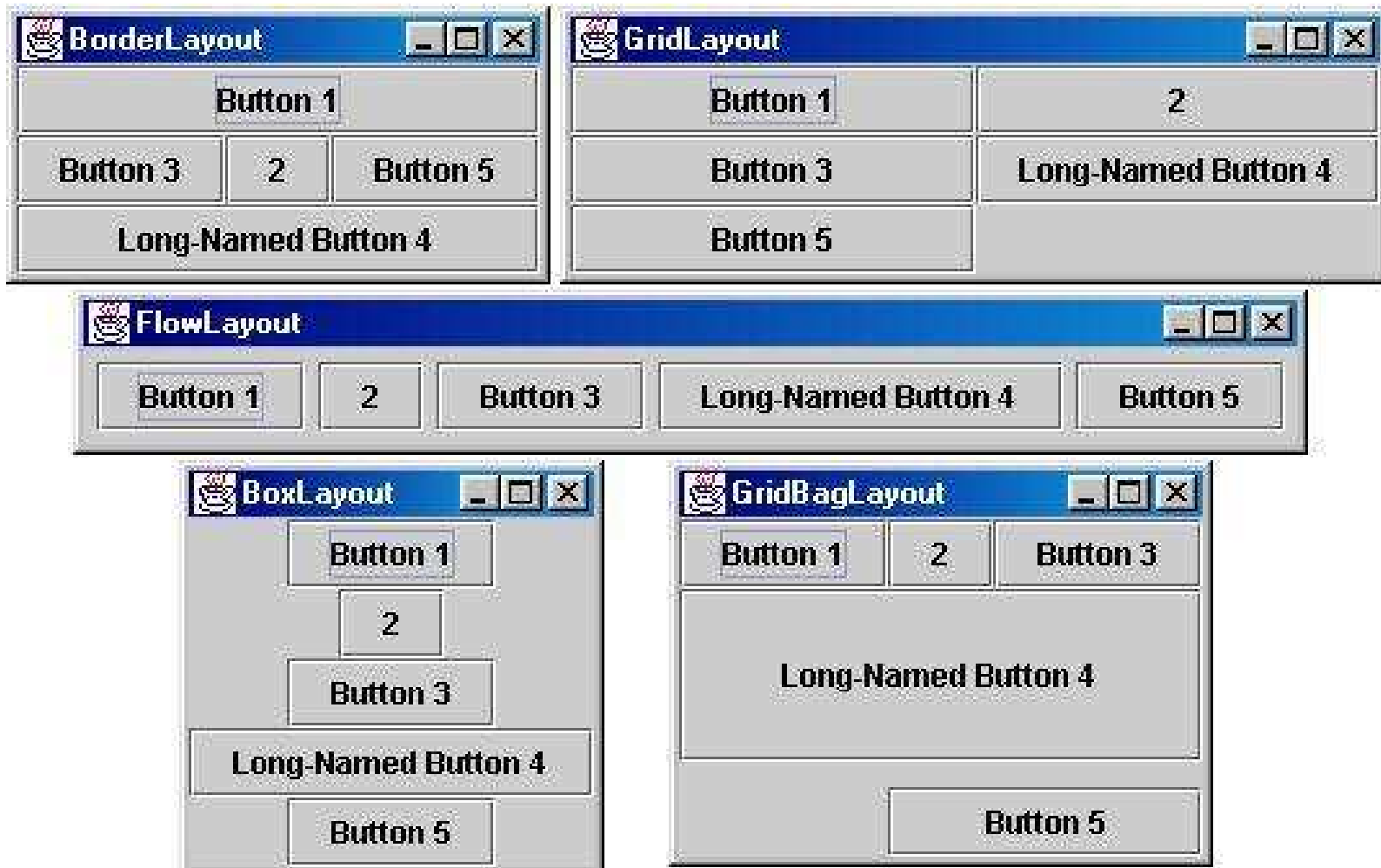
Explications

- La disposition des composants dans une fenêtre est gérée par un gestionnaire de mise en forme (Layout Manager)
- Par défaut le gestionnaire d'un **JFrame** est une instance de la classe **BorderLayout** dont le comportement par défaut fait que le bouton occupe toute la fenêtre.

Gestionnaires de mise en page

- BorderLayout, FlowLayout, CardLayout, GridLayout, BoxLayout, GridBagLayout et GroupLayout

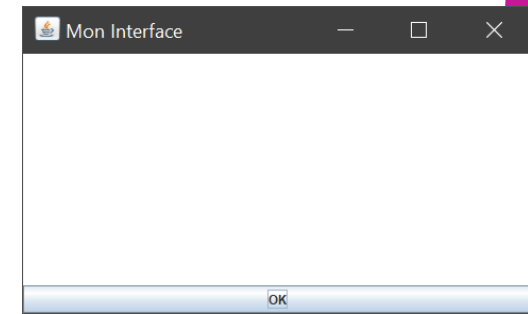
Gestionnaire de mise en page (2)



Gestionnaire de mise en page (3)

Exemple de gestion du contenu avec BorderLayout

```
public class TestInterfaceBorderLayout {  
    // Déclaration des variables  
    ...  
    void dessinerFenetre() {  
        fenetre = new JFrame(); // Création fenêtre  
        fenetre.setSize(500, 300);  
        fenetre.setTitle("Mon Interface");  
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        fenetre.setLocationRelativeTo(null);  
        conteneur = fenetre.getContentPane(); // Type Container  
        conteneur.setLayout(new BorderLayout()); //Utilisation BorderLayout  
        monBouton = new JButton("OK"); //Création compostants  
        panneau = new JPanel();  
        panneau.setBackground(Color.WHITE);  
        conteneur.add(monBouton, BorderLayout.SOUTH); //Positionnement  
        conteneur.add(panneau, BorderLayout.CENTER);  
        fenetre.setVisible(true); // Affichage fenêtre  
    }  
}
```



Programmation événementielle

Principe

- Associer un traitement si un événement se produit sur une composante graphique

Plusieurs démarches

– Exemple :

- Créer une classe qui implémente l'interface **ActionListener**
- Implémenter dans la classe la méthode **actionPerformed()** qui va déclencher le traitement si un événement se produit
- Associer cette classe comme Gestionnaire d'événements à un composant via la méthode **addActionListener()**

Interface ActionListener (1)

- Ajouter un traitement si le bouton est actionner :

```
class MaFenêtre extends JFrame{
    MaFenêtre(){
        ...
        JButton b = new JButton("Test");
        b.addActionListener(new EcouteurAction(b));
        ...
    }
}

class EcouteurAction implements ActionListener{
    private JButton b;
    EcouteurAction(JButton b){this.b = b;}
    public void actionPerformed(ActionEvent arg0) {
        b.setVisible(false);
    }
}
```

Ajouter d'un écouteur d'action sur le bouton

L'unique méthode à implémenter (d'où l'absence d'une classe ActionListenerAdapter)

Interface ActionListener (2)

- Exemple avec deux boutons :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestBoutons {
    public static void main(String
args[]){
        new MaFenêtre();}
}

class MaFenêtre extends JFrame
implements ActionListener{
    private JButton b1, b2;

    MaFenêtre(){
        this.setTitle("2boutons");
        this.setSize(200,80);
```



```
this.setDefaultCloseOperation(
DISPOSE_ON_CLOSE);
this.setVisible(true);
b1 = new JButton("OK");
b2 = new JButton("KO");
this.getContentPane().add(b1);
this.getContentPane().add(b2);
this.getContentPane().setLayou
t(new FlowLayout());
b1.addActionListener(this);
b2.addActionListener(this);
}

public void
actionPerformed(ActionEvent e){
    if(e.getSource()==b1)
        System.out.println("OK");
    else
        System.out.println("Annuler");
}
```

Retourne la référence de l'objet
source de l'événement

Interface ActionListener (3)

Exercice

- Donnez le code source d'un programme Java permettant d'afficher une fenêtre comme celle-ci :



- Le premier bouton permet d'afficher un message et le deuxième de réinitialiser l'interface :



Liste déroulante

Création

- Classe : **JComboBox <T>**
- Ajouter des éléments dans la liste déroulante :
 - **Méthode 1** : passer un tableau d'éléments (exemple un tableau de chaînes de caractères) comme paramètre du constructeur
 - **Méthode 2** : utiliser la méthode **addItem()**
- Récupérer l'élément sélectionné
 - **getSelectedItem()**
- Ajouter un gestionnaire d'événements
 - **addActionListener()**

Liste déroulante (2)

Exercice

- Créez une application qui offre une liste déroulante de pays et qui affiche le nom du pays sélectionné



Cases à cocher (1)

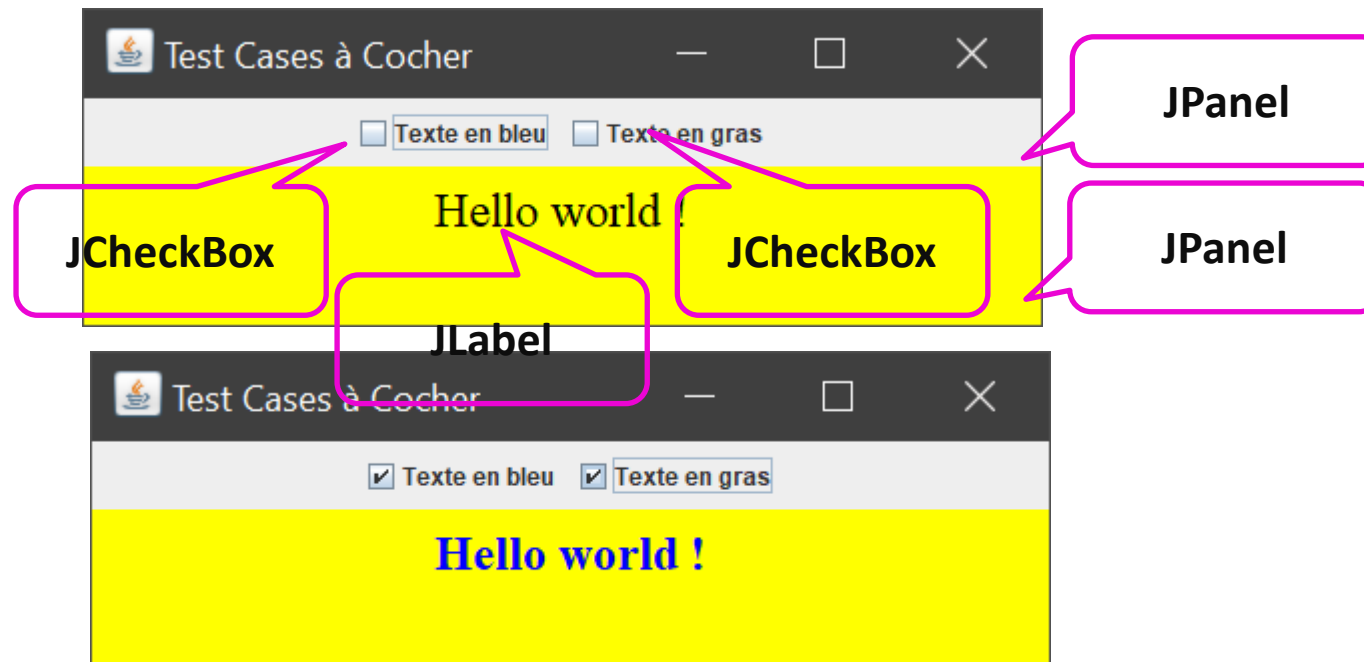
Création

- Classe : **JCheckBox**
- Fixer le texte à afficher avec la case à cocher
 - Donner le texte comme paramètre du constructeur :
JCheckBox(String)
- Vérifier si l'élément est sélectionné ou non
 - **boolean isSelected()**
- Ajouter un gestionnaire d'événements
 - **addActionListener()**

Cases à cocher (2)

Exercice

- Créez une application qui offre des cases à cocher pour modifier le style d'un texte afficher



Boutons Radio (1)

Création

– Bouton radio

- Donner le texte comme paramètre du constructeur : **JRadioButton(String)**
- Cocher un bouton radio : **setSelected(true)**
- Vérifier l'état d'un bouton radio : **isSelected()**
- Ajouter un gestionnaire d'événements : **addActionListener()**

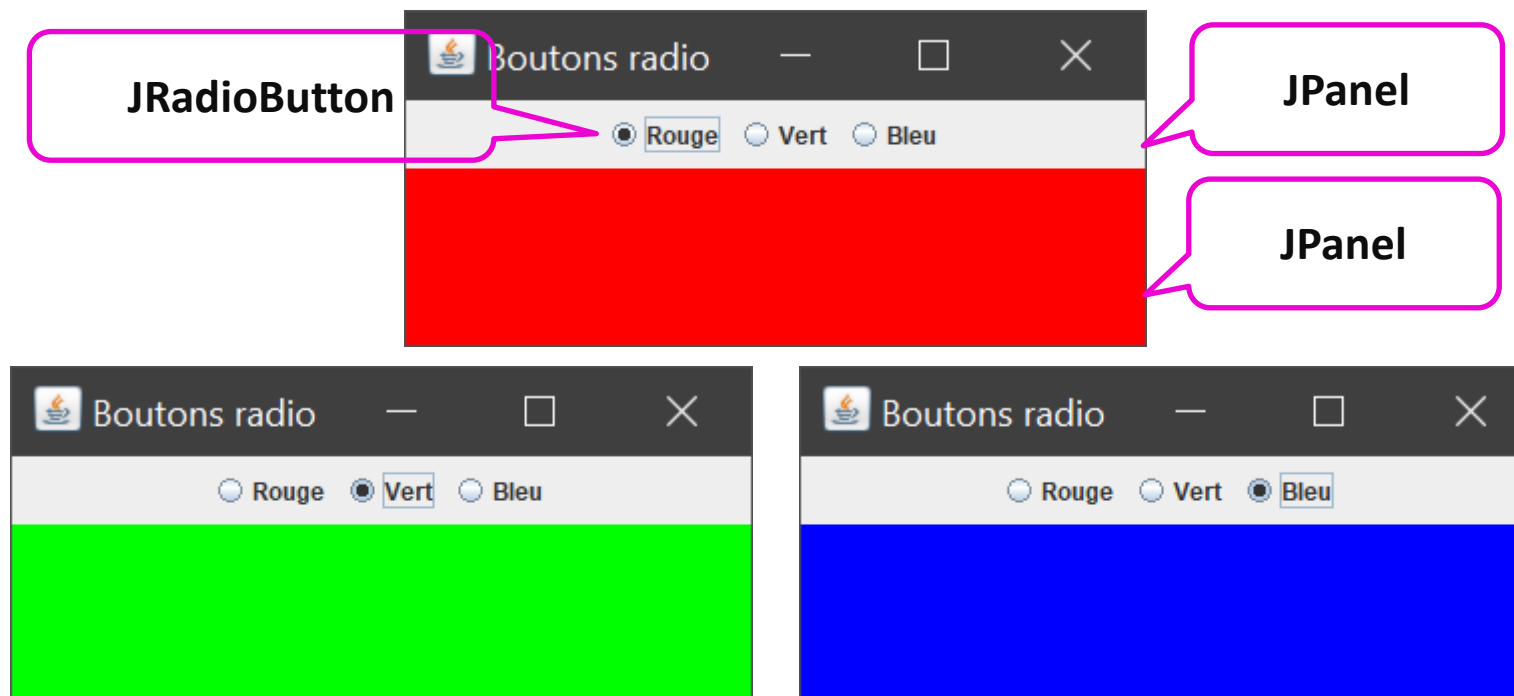
– Ajouter le bouton radio au groupe de boutons

- Créer un groupe de boutons de type **ButtonGroup**
- Ajouter les bouton dont la sélection est exclusive au groupe via la méthode **add()**

Boutons Radio (2)

Exercice

- Créez une application qui offre des boutons radio pour choisir la couleur d'arrière plan d'un panneau



Champ texte (1)

Création

- Instance de :
 - **TextField**
- Spécifier un texte par défaut
 - **TextField("Texte par défaut")**
- Spécifier la taille du champ texte :
 - **setPreferredSize(new Dimension(150, 30));**
- Spécifier la police de caractères (Spécifier la famille de police, la décoration du texte et la taille du texte)
 - **setFont(new Font("Arial", Font.BOLD, 14))**
- Spécifier la couleur du texte
 - **setForeground(couleur)**

Champ texte (2)

Exercice

- Créez une application qui offre un champ texte permettant d'indiquer un nom. Si on valide le nom un message avec le nom donné dans le champ texte



Champ texte (3)

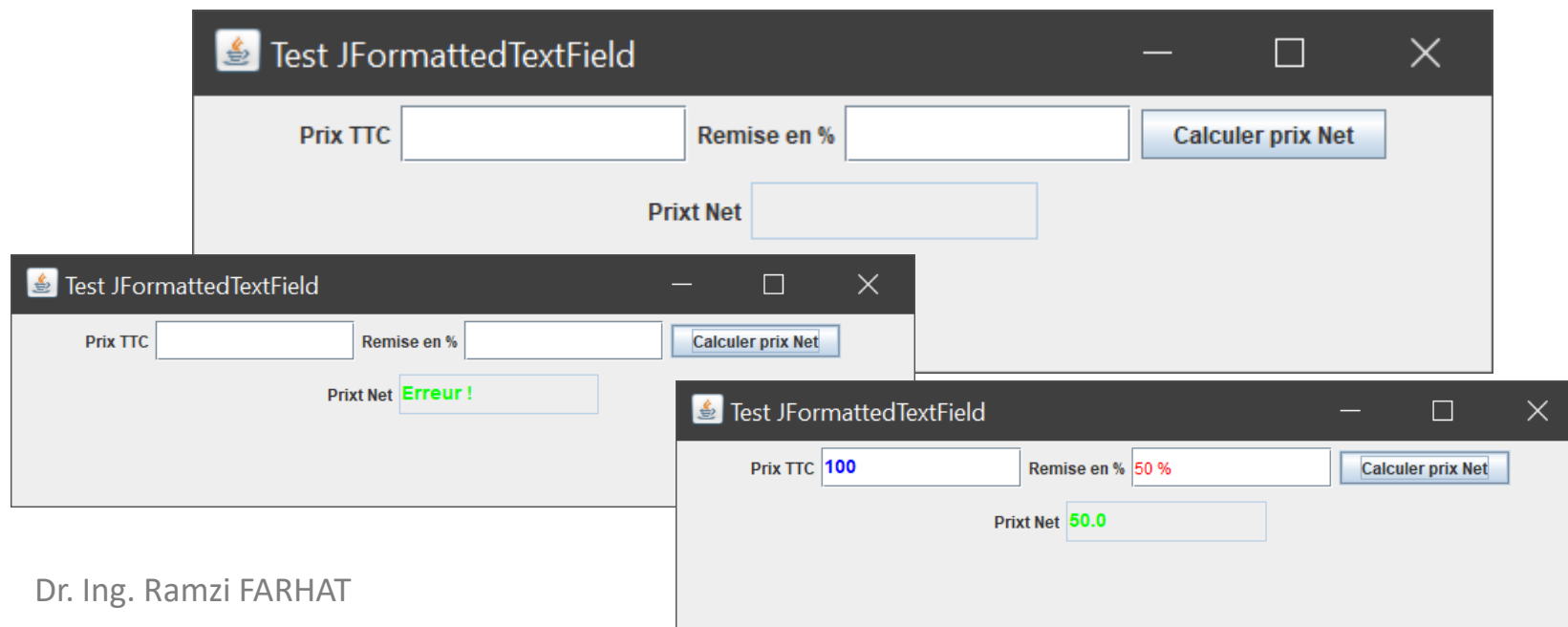
Champ texte avec contraintes

- Instance de :
 - **JFormattedTextField**
- Objectif
 - Accepter uniquement une valeur qui répond à un format spécifique (Entier, date, etc.)
- Formats :
 - NumberFormat avec : `getIntegerInstance()`, `getPercentInstance()`, `getNumberInstance()`
 - DateFormat avec : `getTimeInstance()` et `getDateInstance()`
 - MessageFormat
 - MaskFormatter
- Exemple de création :
 - `new JFormattedTextField(NumberFormat.getPercentInstance());`

Champ texte (4)

Exercice

- Créez une application qui permet de saisir un prix TTC et une remise (en pourcentage) et qui permet de calculer le prix Net.



Interface MouseListener (1)

- Présentation
 - Permet d'écouter et de réagir aux événements de la souris
 - Fait partie du package : `java.awt.event`
 - Comporte cinq méthodes
 - `public void mouseClicked(MouseEvent ev){ ...}`
 - `public void mousePressed (MouseEvent ev) {...}`
 - `public void mouseReleased(MouseEvent ev) {...}`
 - `public void mouseEntered (MouseEvent ev) {...}`
 - `public void mouseExited (MouseEvent ev) {...}`

Interface MouseListener (2)

- Etape 1 : Définir la classe MaFenêtre

```
class MaFenêtre extends JFrame{  
    MaFenêtre(){  
        this.setTitle("Ma deuxième fenêtre");  
        this.setSize(400,400);  
        this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);  
        this.setVisible(true);  
        // Ajouter un gestionnaire pour la souris  
        this.addMouseListener(new EcouteurSouris());  
    }  
}
```

Cette classe est à définir.
Elle doit implémenter l'interface ***MouseListener***
afin de définir ce qu'il faut faire pour chaque
action de la souris.

Interface MouseListener (3)

- Etape 2 : Définir la classe GestionnaireSouris

```
class EcouteurSouris implements MouseListener{  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Clic");  
    }  
    public void mouseEntered(MouseEvent e) {  
        System.out.println("Pointeur sur la fenêtre");  
    }  
    public void mouseExited(MouseEvent e) {  
        System.out.println("Pointeur hors de la  
fenêtre");  
    }  
    public void mousePressed(MouseEvent e) {}  
    public void mouseReleased(MouseEvent e) {}  
}
```

Interface MouseListener (4)

- Etape 3 : Tester la fenêtre

```
import javax.swing.*;
// Nécessaire pour utiliser JFrame
import java.awt.event.*;
// Nécessaire pour utiliser MouseListener

public class TestEvenementSouris {
    public static void main(String args[]){
        new MaFenêtre();
    }
}
```

Classe MouseAdapter (1)

- Objectif :
 - Éviter d'implémenter toutes les méthodes de l'interface ***MouseListener***
- Principe :
 - Elle offre une implémentation par défaut de toutes les méthodes
 - Il suffit d'hériter de cette classe et de redéfinir que les méthodes nécessaires

Classe MouseAdapter (2)

- Exemple :

```
class EcouteurSouris extends MouseAdapter{  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Clic");  
    }  
    public void mouseEntered(MouseEvent e) {  
        System.out.println("Pointeur sur la fenêtre");  
    }  
    public void mouseExited(MouseEvent e) {  
        System.out.println("Pointeur hors de la  
fenêtre");  
    }  
}
```

Gestion des événements

- Récapitulation :
 - Chaque événement a :
 - une source qui déclenche l'événement (dans les exemples la souris)
 - Un écouteur (dans les exemples une classe dérivée de `MouseAdapter` ou implémentant `MouseListener`)
 - Si la catégorie de la source est Xxx :
 - On associe un écouteur de type `XxxEvent`
 - On dispose d'une méthode `addXxxListener`
 - On doit Implémenter l'interface `XxxListener` (ou hériter de la classe `XxxAdapter` si plusieurs méthodes existent)

Mot de fin

