

JAVA : Introduction

Référence : <http://deptinfo.unice.fr/~grin>

« Write once, run everywhere »



Plan du cours

- **JAVA de base**
- **Notion d'objet**
- **Notion de classe**
- **Héritage**
- **Polymorphisme**
- **Classe abstraite et interface**
- **Exception**
- **Interface graphique**

Principales propriétés de Java

- Langage orienté objet, à classes (les objets sont décrits/regroupés dans des classes)
- de syntaxe proche du langage C
- fourni avec le JDK (Java Development Kit) :
 - outils de développement
 - ensemble de paquetages très riches et très variés
- portable grâce à l'exécution par une machine virtuelle JVM

Principales propriétés de Java

- multi-tâches (*thread*)
- sûr
 - fortement typé
 - nombreuses vérifications au chargement des classes et durant leurs exécution
- adapté à Internet
 - chargement de classes en cours d'exécution (le plus souvent par le réseau : *applet*)
 - facilités pour distribuer les traitements entre plusieurs machines (sockets, Corba, EJB)

Premier programme JAVA

- Le « profil » d'une méthode est donné par son entête de définition ; celui de main() doit être : `public static void main(String[] args)`

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

La classe Helloworld est **public**

Compilation d'un code source

- Un code source ne peut être exécuté directement par un ordinateur
- Il faut traduire ce code source dans un langage que l'ordinateur (le processeur de l'ordinateur) peut comprendre (langage *natif*)
- Un compilateur est un programme qui effectue cette traduction

Compilation en Java → *bytecode*

- En Java, le code source n'est pas traduit directement dans le langage de l'ordinateur
- Il est d'abord traduit en un langage appelé « *bytecode* », langage d'une machine virtuelle(JVM ; *Java Virtual Machine*) définie par *Sun*
- Ce langage est indépendant de l'ordinateur qui va exécuter le programme

La compilation
fournit du
bytecode

Programme écrit en Java

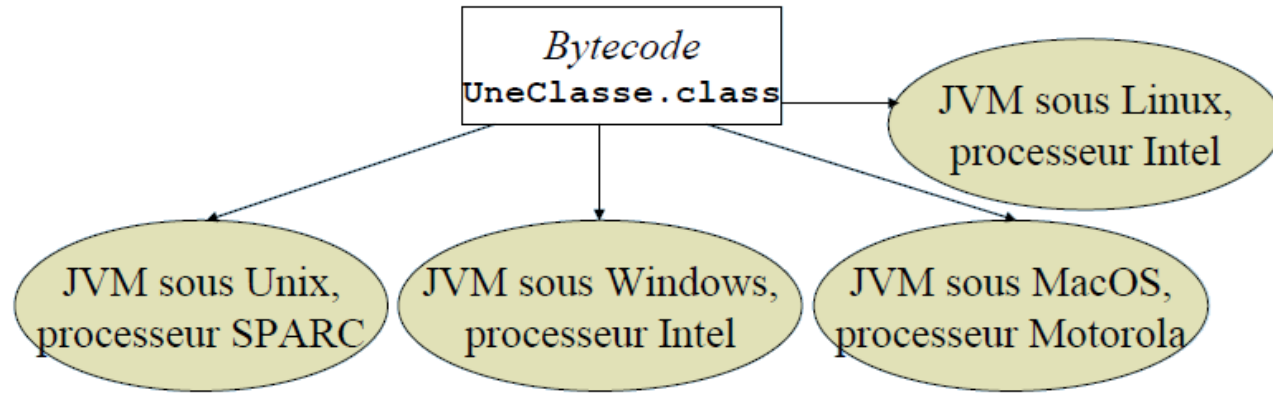
Programme source
`UneClasse.java`

Compilateur

Programme en *bytecode*,
indépendant de l'ordinateur

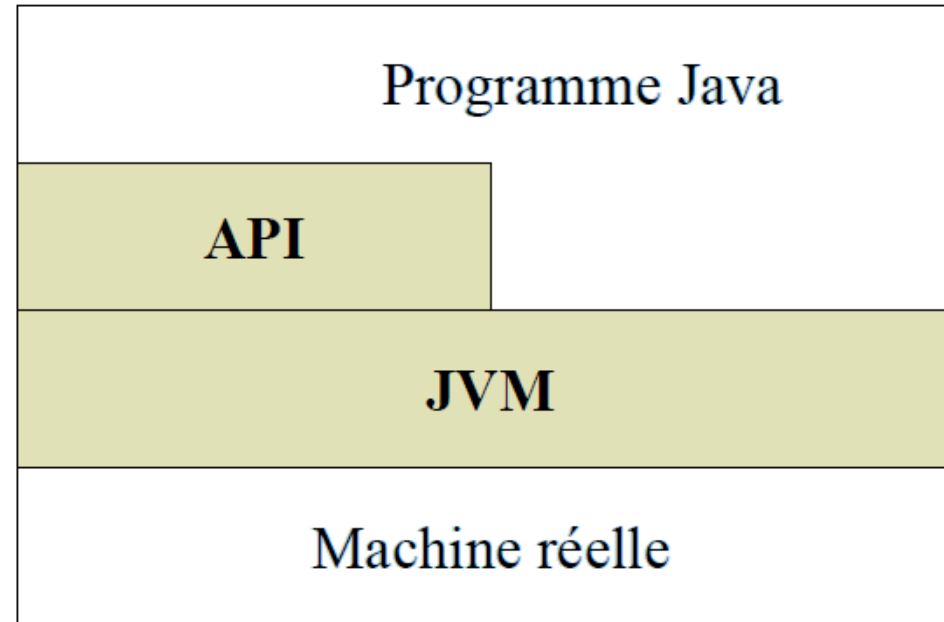
Bytecode
`UneClasse.class`

Le *bytecode*
peut être
exécuté par
n'importe
quelle JVM



Si un système possède une JVM, il peut exécuter tous les fichiers `.class` compilés sur n'importe quel autre système

Plate-forme Java



- API (*Application Programming Interface*) : bibliothèques de classes standard

éditions de Java

- Java SE (J2SE) : Java Standard Edition ; JDK = *Java SE Development Kit*
- Java EE (J2EE) : Enterprise Edition qui ajoute les API pour écrire des applications installées sur les serveurs dans des applications distribuées : servlet, JSP, JSF, EJB,...
- Java ME (J2ME) : Micro Edition, version pour écrire des programmes embarqués (carte à puce/*Java card*, téléphone portable,...)

Notion d'objet en Java

- Un objet a
 - une adresse en mémoire (identifie l'objet)
 - un comportement (ou interface)
 - un état interne
- L'état interne est donné par des valeurs de variables
- Le comportement est donné par des fonctions ou procédures, appelées méthodes

Exemple : classe livre

```
public class Livre {
```

```
    private String titre, auteur;  
    private int nbPages;
```

Variables d'instance

```
    // Constructeur
```

```
    public Livre(String unTitre, String unAuteur) {  
        titre = unTitre;  
        auteur = unAuteur;  
    }
```

Constructeurs

```
    public String getAuteur() {           // accesseur  
        return auteur;  
    }
```

Méthodes

```
    public void setNbPages(int nb) { // modificateur  
        nbPages = nb;  
    }
```

```
}
```

Scanner

- L'objet Scanner se charge de la lecture des entrées clavier
- Il se trouve dans le package java.util
- Afin de récupérer les données dans la console, il faudra initialiser l'objet Scanner avec l'entrée standard, System.in.
- Il y a une méthode de récupération de données pour chaque type (sauf les char) : nextInt() pour les int

```
import java.util.Scanner;  
Scanner s1 = new Scanner(System.in);  
nombre=s1.nextDouble();
```

Tostring()

- La méthode toString, définie dans la classe Object, admet pour prototype : `public String toString();`
- Quand on redéfinit la méthode toString, on fait en sorte qu'elle renvoie une chaîne de caractères servant à décrire l'objet concerné

Représentation
graphique
d'une
classe en
notation UML
(*Unified
Modeling
Language*)

Cercle
private Point centre private int rayon
public Cercle(Point, int) public void setRayon(int) public int getRayon() public double surface()

Cercle
- Point centre - int rayon
+ Cercle(Point, int) + void setRayon(int) + int getRayon() + double surface()

(- : private, # : protected, + : public, \$ (ou souligné) : static)

Quizz

- Q1 : JAVA est un langage

a) compilé interprété b) Interprété c) Compilé et interprété d) Ni compilé ni interprété

- Q2 : Quel est le résultat du code suivant :

```
public class Incrementation
```

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

```
{ int i, j, n ;
```

```
i = 0 ; n = i++ ;
```

```
System.out.println ("i = " + i + " n = " + n ) ;}}
```

i=1 ,n=0

b) i=1 ,n=1

Q3 : Est-ce que on peut avoir plusieurs constructeurs pour la même classe

Oui

b) non

Quizz

• **Q1 :** JAVA est un langage

- a) compilé interprété b) Interprété c) Compilé et interprété d) Ni compilé ni interprété

• **Q2 :** Quel est le résultat du code suivant :

```
public class Incrementation
{ public static void main(String[] args)
{ int i, j, n ;
i = 0 ; n = i++ ;
System.out.println ("i = " + i + " n = " + n ) ;}}
```

a) i=1 ,n=0

b) i=1 ,n=1

Q3 : Est-ce que on peut avoir plusieurs constructeurs pour la même classe

a) Oui

b) non

Exercices d'application:

Exercice1:

- Écrivez un programme qui calcule la racine carrée de nombres fournis en donnée (les nombres doivent être saisis au clavier).
- Le programme n'accepte que les nombres positifs.
- Si on introduit un nombre négatif, on reçoit un message d'information
- Tapez 0 pour sortir du programme

Exercice2:

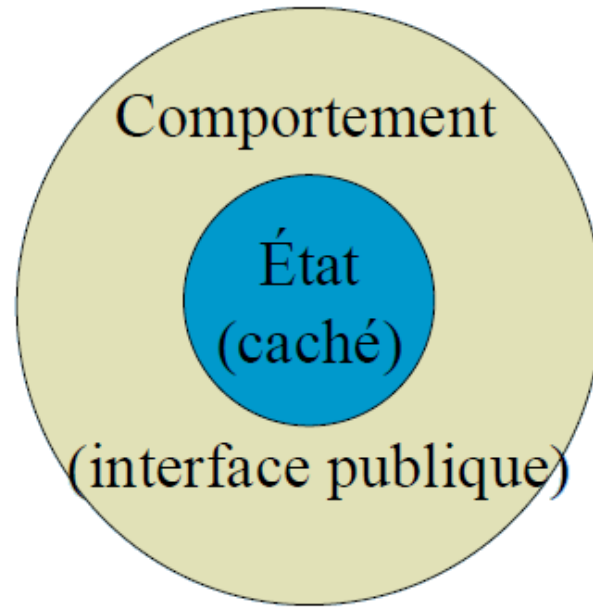
Un compte bancaire est de somme X (à saisir), avec un taux d'intérêt mensuelle de 0.01%

- Quelles sont les intérêts cumulés après un nombre Y de mois (à saisir)
- Quelle est le montant finale

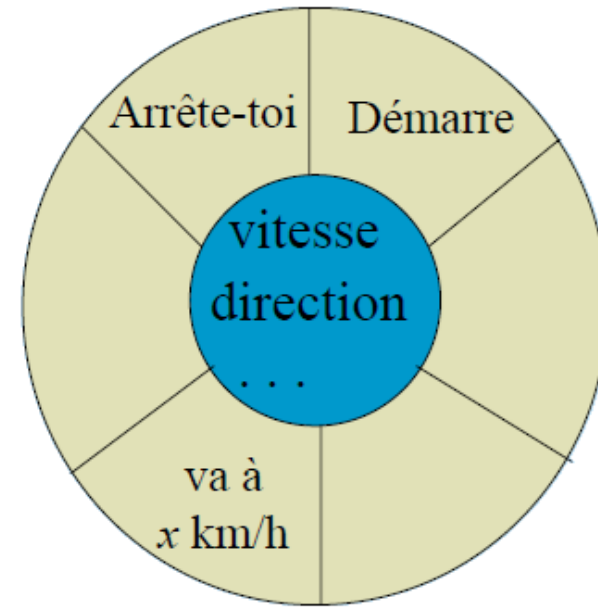
• Exercice3:

Ecrivez un programme JAVA qui modélise un point de coordonnées x, y et z. Créez trois instantiations du point : point a, point b et point c.

Un objet



Un objet



Une voiture

Interactions entre objets

Interactions entre objets

- Les objets interagissent en s'envoyant des **messages synchrones**
- Les méthodes d'un objet correspondent aux messages qu'on peut lui envoyer : quand un objet reçoit un message, il exécute la méthode correspondante

- Exemples :

```
objet1.decrisToi() ;  
employe.setSalaire(20000) ;  
voiture.demarre() ;  
voiture.vaAVitesse(50) ;
```

Objet qui reçoit
le message

Message envoyé

Paramètre
du message

Regrouper les objets

- Les constructeurs (il peut y en avoir plusieurs) servent à créer des objets appelés instances de la classe
- Quand une instance est créée, son état est conservé dans les variables d'instance
- Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message
- Les variables et les méthodes s'appellent les membres de la classe

Rôles d'une classe

- Une classe est
 - un type qui décrit une structure (variables d'état) et un comportement (méthodes)
 - un module pour décomposer une application en entités plus petites
 - un générateur d'objets (par ses constructeurs)
- Une classe permet d'encapsuler les objets :
les membres public sont vus de l'extérieur mais les membres private sont cachés

Classes et instances

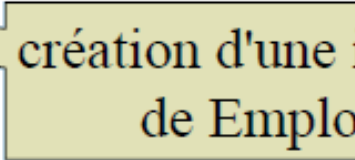
- Une instance d'une classe est créée par un des constructeurs de la classe
- Une fois qu'elle est créée, l'instance
 - a son propre état interne (les valeurs des variables)
 - partage le code qui détermine son comportement (les méthodes) avec les autres instances de la classe

Constructeurs d'une classe

- Chaque classe a un ou plusieurs constructeurs qui servent à
 - créer les instances
 - initialiser l'état de ces instances
- Un constructeur
 - a le même nom que la classe
 - n'a pas de type retour

Création d'une instance

```
public class Employe {  
    private String nom, prenom; } variables  
    private double salaire;    } d'instance  
  
    // Constructeur  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
    }  
    . . .  
    public static void main(String[] args) {  
        Employe e1;  
        e1 = new Employe("Dupond", "Pierre");  
        e1.setSalaire(1200);  
        . . .  
    }  
}
```



création d'une instance
de Employe

Plusieurs constructeurs (surcharge)

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
  
    // 2 Constructeurs  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
    }  
    public Employe(String n, String p, double s) {  
        nom = n;  
        prenom = p;  
        salaire = s;  
    }  
  
    . . .  
    e1 = new Employe("Dupond", "Pierre");  
    e2 = new Employe("Durand", "Jacques", 1500);  
}
```

Désigner un constructeur par `this()`

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
  
    // Ce constructeur appelle l'autre constructeur  
    public Employe(String n, String p) {  
        this(n, p, 0);  
    }  
    public Employe(String n, String p, double s) {  
        nom = n;  
        prenom = p;  
        salaire = s;  
    }  
    . . .  
    e1 = new Employe("Dupond", "Pierre");  
    e2 = new Employe("Durand", "Jacques", 1500);  
}
```

Constructeur par défaut

- Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java
- Pour une classe **Classe**, ce constructeur par défaut sera :

```
[public] Classe() { }
```

Même accessibilité que
la classe (**public** ou non)

Les méthodes

Accesseurs:

- Deux types de méthodes servent à donner accès aux variables depuis l'extérieur de la classe :
 - les accesseurs en lecture pour lire les valeurs des variables ; « accesseur en lecture » est souvent abrégé en « accesseur » ; *getter* en anglais
 - les accesseurs en écriture, ou modificateurs, ou mutateurs, pour modifier leur valeur ; *setter* en anglais

Surcharge

- En Java, on peut surcharger une méthode,
- c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :
- `calculerSalaire(int)`
- `calculerSalaire(int, double)`

Exemple

```
public class Employe {  
    . . .  
    public void setSalaire(double unSalaire) {  
        if (unSalaire >= 0.0)  
            salaire = unSalaire;  
    }  
    public double getSalaire() {  
        return salaire;  
    }  
    public boolean accomplir(Tache s) {  
        . . .  
    }  
}
```

Modificateur

Accesseur

Les variables

Types de variables

- Les variables d'instances
 - sont déclarées en dehors de toute méthode
 - conservent l'état d'un objet, instance de la classe
 - sont accessibles et partagées par toutes les méthodes de la classe
- Les variables locales
 - sont déclarées à l'intérieur d'une méthode
 - conservent une valeur utilisée pendant l'exécution de la méthode
 - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées

Types d'autorisation d'accès

- **private** : seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur)
- **public** : toutes les classes sans exception y ont accès
- **protected** : un élément protected (protégé) est accessible uniquement aux classes d'un package et à ses classes filles
- **Par défaut**: seules les classes du même package que la classe dans lequel il est déclaré y ont accès (un package est un regroupement de classes ; cette notion sera étudiée plus loin dans le cours)

Encapsulation

En générale le principe de l'encapsulation offre:

- Rapprochement des données (attributs) et traitements (méthodes)
- Protection de l'information (private, public, ...)

L'héritage

- L'héritage existe dans tous les langages objet à classes
- L'héritage permet d'écrire une classe **B**
 - qui se comporte dans les grandes lignes comme la classe **A**
 - mais avec quelques différences sans toucher au code source de **A**
- La classe **A** s'appelle une classe mère, classe parente ou super-classe
- La classe **B** qui hérite de la classe **A** s'appelle une classe fille ou sous-classe

L'héritage

- En Java, chaque classe a une et une seule classe mère (pas d'héritage multiple) dont elle hérite les variables et les méthodes
- Le mot clef **extends** indique la classe mère :
class RectangleColore extends Rectangle
- Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object**.

Classe mère

- `public class Rectangle {`
- `private int x, y;`
- `private int largeur, hauteur;`
- `// La classe contient des constructeurs,`
- `// des méthodes getX(), setX(int)`
- `// getHauteur(), getLargeur(),`
- `// setHauteur(int), setLargeur(int),`
- `// contient(Point), intersecte(Rectangle)`
- `// translateToi(Vecteur), toString(),...`
- `//...`
- `public void dessineToi(Graphics g) {`
- `g.drawRect(x, y, largeur, hauteur);`
- `}`
- `}`

Classe fils

- **public class RectangleCouleur extends Rectangle {**
- **private Color couleur; // nouvelle variable**

// Constructeurs

...

// Nouvelles Méthodes

- **public getCouleur() { return this.couleur; }**
- **public setCouleur(Color c) { this.couleur = c; }**

// Méthodes modifiées

- **public void dessineToi(Graphics g) {**
g.setColor(couleur);
g.fillRect(getX(), getY(),
getLargeur();
getHauteur());
• **}**
• **}**

Héritage

- Un constructeur de la classe mère : **super(...)**
- Un constructeur de la classe : **this(...)**

```
public class Rectangle {  
    private int x, y, largeur, hauteur;  
  
    public Rectangle(int x, int y,  
                    int largeur, int hauteur) {  
        this.x = x;  
        this.y = y;  
        this.largeur = largeur;  
        this.longueur = longueur;  
    }  
    . . .  
}
```

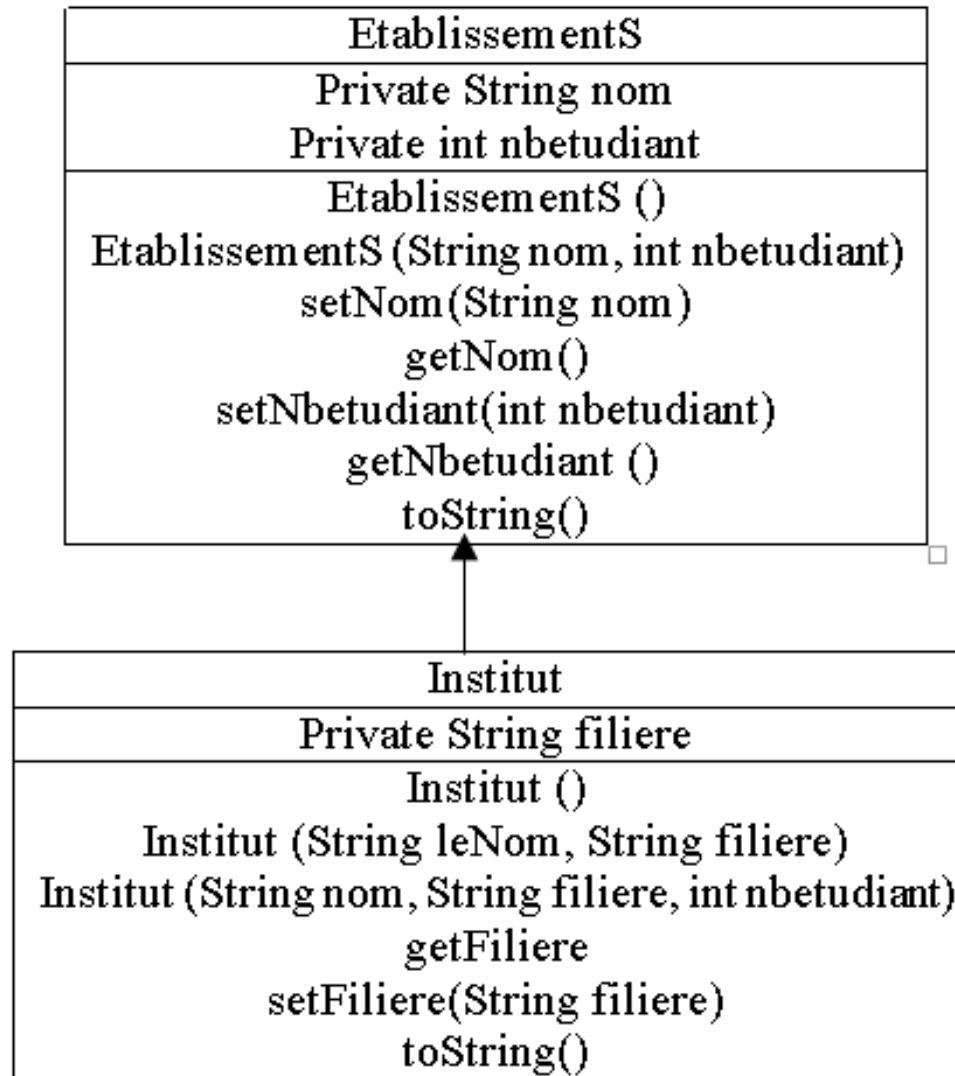

Héritage

- Un constructeur de la classe mère : **super(...)**
- Un constructeur de la classe : **this(...)**

```
public class RectangleColore extends Rectangle {  
    private Color couleur;  
    public RectangleColore(int x, int y,  
                           int largeur, int hauteur  
                           Color couleur) {  
        super(x, y, largeur, hauteur);  
        this.couleur = couleur;  
    }  
    public RectangleColore(int x, int y,  
                           int largeur, int hauteur) {  
        this(x, y, largeur, hauteur, Color.black);  
    }  
    . . .  
}
```

Exercice d'application

Ecrivez un programme qui modélise le diagramme suivant :



Tableaux

Les tableaux sont des objets

- En Java les tableaux sont considérés comme des objets (dont la classe hérite de Object) :
 - les variables de type tableau contiennent des références aux tableaux
 - les tableaux sont créés par l'opérateur new
 - ils ont une variable d'instance (final) : `final int length`
 - ils héritent des méthodes d'instance de Object

Tableaux

```
typeElements nomTableau [ ] = new typeElements [nombre de cases] ;
```

Déclaration et création des tableaux

- Création : on doit donner la taille

```
int tabEntiers = new int[5];
```

Chaque élément du tableau reçoit la valeur par défaut du type de base du tableau

- La taille ne pourra plus être modifiée par la suite

Tableaux

- Faute fréquente : utiliser les objets du tableau avant de les avoir créés

```
Employe[] personnel = new Employe[100];  
personnel[0].setNom("Dupond");
```

```
Employe[] personnel = new Employe[100];  
personnel[0] = new Employe();  
personnel[0].setNom("Dupond");
```

Création
1er employé

Tableaux

- On peut affecter « en bloc » tous les éléments d'un tableau avec un **tableau anonyme** :

```
int[] t;  
.  
.  
.  
t = new int[] {1, 2, 3};
```

- **Affectation** des éléments ; l'indice commence **0** et se termine à `tabEntiers.length - 1`

```
tabEntiers[0] = 12;
```

Polymorphisme

- Le nom de **polymorphisme** signifie "**qui peut prendre plusieurs formes**". Cette caractéristique est un des concepts essentiels de la programmation orientée objet. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est relatif aux méthodes des objets.

Surcharge

- En programmation objet et en Java en particulier, un même nom de fonction peut être utilisé pour plusieurs fonctions. C'est notamment le cas lorsque les fonctions réalisent des actions conceptuellement voisines. On dit qu'il y a surcharge des fonctions. Il doit cependant y avoir possibilité de distinguer quelle est la fonction à mettre en œuvre.

Surcharge

- La surcharge de méthode consiste à garder le nom d'une méthode et à changer la liste ou le type de ses paramètres
- Par exemple l'utilisation de plusieurs constructeurs:

```
public Employe ()  
{  
  
public Employe (String name, double salaire)  
{  
    this.name=name;  
    this.salaire=salaire;  
}  
public Employe (String name)  
{  
    this.name=name;  
}
```

Redéfinition

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques)

@Override

```
public String toString() {  
    return "Upemploye [name=" + name + ", salaire=" +  
    + salaire + "];"  
}
```

Redéfinition de la méthode *toString()* héritée de la super classe *Object*.

Mot-clé final

Le mot-clé final peut-être utilisé dans différentes situations :

1. avec une *variable* : il spécifie que la variable ne pourra plus être modifiée et doit dès lors être initialisée dès sa déclaration ;
2. avec une *méthode* : il empêche toute redéfinition de la méthode déclarée comme telle ;
3. avec une *classe* : il empêche d'hériter de la classe déclarée comme telle ; celle-ci ne pourra donc jamais être une *superclasse*.

Dans la hiérarchie des classes, on retrouvera donc les *classes abstraites* "en haut" et les *classes finales* "en bas".

Mot-clé final

- `private final float pi = 3.14f;`
- `getClass()` de la classe `Object` est un exemple de ce type de méthode
- `public final int MethodeX(){`
`//to do`
`}`
- `final class ClassX{`
`//to do`
`}`

Mot-clé static

- Les variables statiques n'existent qu'en un seul exemplaire pour toutes les instances de la classe.
- Le mot clé static permet alors à une méthode de s'exécuter sans avoir à instancier la classe qui la contient
 - => elle exécute une action indépendante d'une instance particulière de la classe

Mot-clé static

```
public class Exemple{  
    ...  
    public static int methode( int a, int b) {  
        // to do  
    }  
}  
  
Public Test{  
    public static void main ( String [] args ){  
        int x = Exemple. methode (1,1);  
    }  
}
```

Mot-clé static

```
package ex_static;

public class CitoyenA {
    public int cin;
    public String nom;
    static String pays="Maroc";
    public CitoyenA(int cin, String nom) {
        this.cin = cin;
        this.nom = nom;
    }
    public void afficher() {
        System.out.println(cin + " | " + nom + " | " + pays + " !");
    }
}
```

```
public class Test {

    public static void main(String[] args) {
        CitoyenA ct=new CitoyenA(1000026,"ali");
        ct.afficher();
    }
}
```

```
1000026 | ali | Maroc !
```

Mot-clé static

- La méthode **main()** est nécessairement **static**.

Pourquoi ?

- La méthode **main()** est exécutée au début du programme. Aucune instance n'est donc déjà créée lorsque la méthode **main()** commence son exécution.
- Ça ne peut donc pas être une méthode d'instance.

Classe abstraite

- Lors de la conception d'une hiérarchie de classes, il peut être judicieux de créer une classe de laquelle hériteront toutes les autres. Cette superclasse ne sera jamais instanciée et permettra aux autres classes d'hériter de son interface et de son implémentation. On parle dès lors de *superclasse abstraite*.
- À l'opposé, une *classe concrète* permet d'instancier des objets

Classe abstraite

Une classe doit être déclarée abstraite (**abstract class**)

- si elle contient une méthode abstraite
- Il est interdit de créer une instance d'une classe

Abstraite

Une méthode est abstraite (modificateur **abstract**) lorsqu'on la déclare, sans donner son implémentation

- La méthode sera implémentée par les classes Filles
- Si on veut empêcher la création d'instances d'une classe on peut la déclarer abstraite même si aucune de ses méthodes n'est abstraite

Interfaces

Une interface est une classe particulière qui présente uniquement :

- des *méthodes* publiques et abstraites ;
- des *données* publiques, statiques et finales.

Une telle interface est définie à l'aide du mot clé `interface` en lieu et place du mot clé `class` :

- `public interface NomInterface`
- Une classe voulant implémenter l'interface le fera à l'aide du mot clé `implements` :

`public class NomClasse implements NomInterface`

Interfaces

- On adoptera la convention de nommage des interfaces en utilisant le suffixe « able » (e.g. Comparable, Drawable, ou Ixxx etc.).
- Cette classe peut implémenter plusieurs interfaces simultanément en spécifiant, après le mot clé implements, la liste des interfaces séparées par une virgule. C'est ce procédé qui permet en Java de simuler l'héritage multiple.
- Java n'autorise que l'héritage simple. Une classe ne peut avoir qu'une seule superclasse. Une certaine forme d'héritage multiple est obtenue grâce à la notion d'interface

Interfaces

- La nouvelle classe NomClasse ainsi créée devra définir toutes les méthodes (avec les mêmes arguments et type de retour) présentées dans l'interface (dans le cas contraire, cette classe devra être *abstraite*). Ceci revient à garantir que toutes les classes implémentant une même interface mettront à disposition un ensemble commun de méthodes qui permettront d'interagir avec elles. Dès lors, d'autres classes "clientes" pourront dialoguer avec les classes réalisant l'interface en étant sûre qu'elles définissent bien les méthodes nécessaires à ce dialogue.

Interfaces

- Soit une interface **I1** et une classe **C** qui l'implémente :

```
public class C implements I1 { ... }
```
 - **C** peut ne pas implémenter **toutes** les méthodes de **I1**
 - Mais dans ce cas **C** doit être déclarée **abstract** (il lui manque des implémentations)
 - Les méthodes manquantes seront implémentées par les classes filles de **C**
-
- Une classe peut implémenter une ou **plusieurs** interfaces (et hériter d'une classe...) :

```
public class CercleColore extends Cercle  
    implements Figure, Coloriable {
```

Interfaces

```
public interface Figure {  
    public abstract void dessineToi();  
    public abstract void deplaceToi(int x,  
                                     int y);  
    public abstract Position getPosition();  
}
```

```
public interface Comparable {  
    /** renvoie vrai si this est plus grand que o */  
    boolean plusGrand(Object o);  
}
```

public abstract
peut être implicite

Polymorphisme et Interface

```
public interface Figure {  
    void dessineToi();  
}
```

```
public class Rectangle implements Figure {  
    public void dessineToi() {  
        . . .  
    }  
}
```

```
public class Cercle implements Figure {  
    public void dessineToi() {  
        . . .  
    }  
}
```

```
public class Dessin {  
    private Figure[] figures;  
    . . .  
    public void afficheToi() {  
        for (int i=0; i < nbFigures; i++)  
            figures[i].dessineToi();  
    }  
    . . .  
}
```


Les exceptions/ les erreurs

- On utilise les erreurs/exceptions pour traiter un fonctionnement anormal d'une partie d'un code (provoqué par une erreur ou un cas exceptionnel)
 - dépassement des limites d'un tableau;
 - division par zéro;
 - paramètres de méthode non valides;
 - épuisement de la mémoire.
- En Java, une erreur ne provoque pas l'arrêt brutal du programme mais la création d'un objet, instance d'une classe spécifiquement créée pour être associée à des erreurs/exceptions

Les exceptions

- Le traitement des erreurs s'effectue dans une zone du programme spéciale (bloc « **catch** »)

```
try {  
    // lire la valeur de n au clavier;  
    int n = lireEntierAuClavier();  
    etagere.add(livre, n);  
}  
catch (NumberFormatException e) {  
    System.err.println("Mauvais numéro de livre");  
}  
catch {EtagerePleineException e} {  
    System.err.println("Etagere pleine");  
}
```

Traitement
normal

Les exceptions

- Si une des instructions du bloc **try** provoque une exception, les instructions suivantes du bloc **try** ne sont pas exécutées
- si au moins une des clauses **catch** correspond au type de l'exception,
 1. la première clause **catch** appropriée est exécutée
 2. l'exécution se poursuit juste après le bloc **try/catch**

Les exceptions

- Dans les cas où l'exécution des instructions de la clause try ne provoque pas d'erreur/exception,
 - le déroulement du bloc de la clause try se déroule comme s'il n'y avait pas de bloc try-catch
 - le programme se poursuit après le bloc try-catch (sauf si exécution de return, break,...)

finally

La clause finally contient un traitement qui sera exécuté de toute façon, que la clause try ait levé ou non une exception, que cette exception ait été saisie ou non

```
try {  
    . . .  
}  
catch (...) {  
    . . .  
}  
finally {  
    . . .  
}
```

On peut, par exemple,
fermer un fichier

Structure générale

```
try
{
    //Code à surveillé

}
catch (ClasseException ex)
{
    //Traitement de l'exception
}

finally // Facultatif
{

    /*Code à exécuter dans tous les cas
    (plantage ou non)
    exemple d'utilisation: fermeture des
    fichiers, des connexions, ... etc.*/
}
```

Traitement général des exceptions

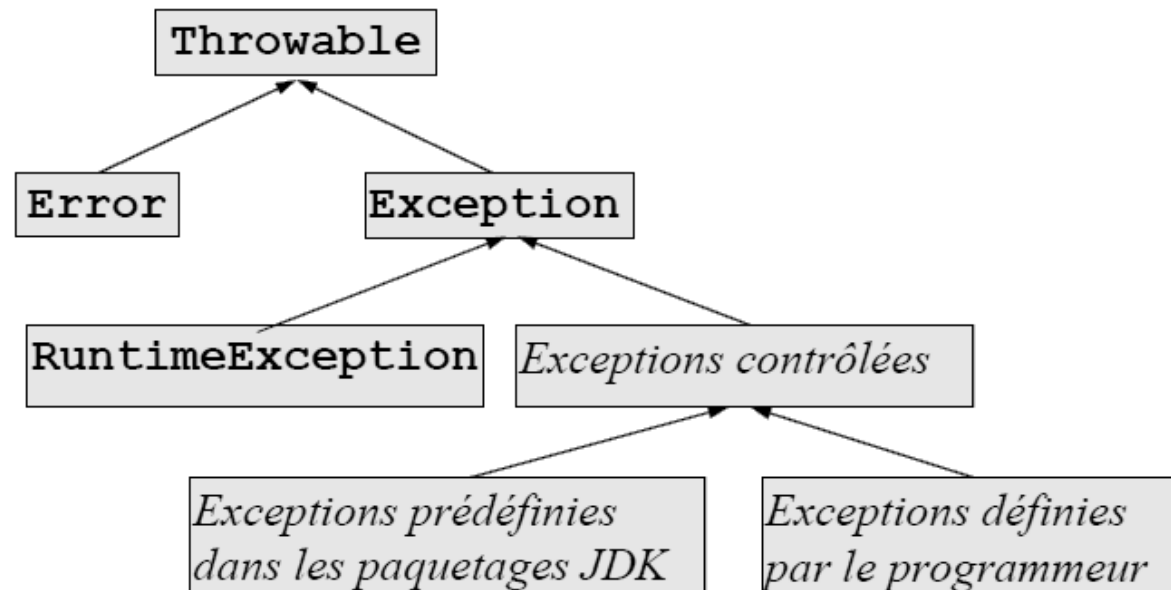
```
try {  
    . . .  
}  
catch (ClasseException1 e) {  
    . . .  
}  
catch (ClasseException2 e) {  
    . . .  
}  
. . .  
finally {  
    . . .  
}
```

Il est possible d'avoir un ensemble
try - finally sans clause **catch**

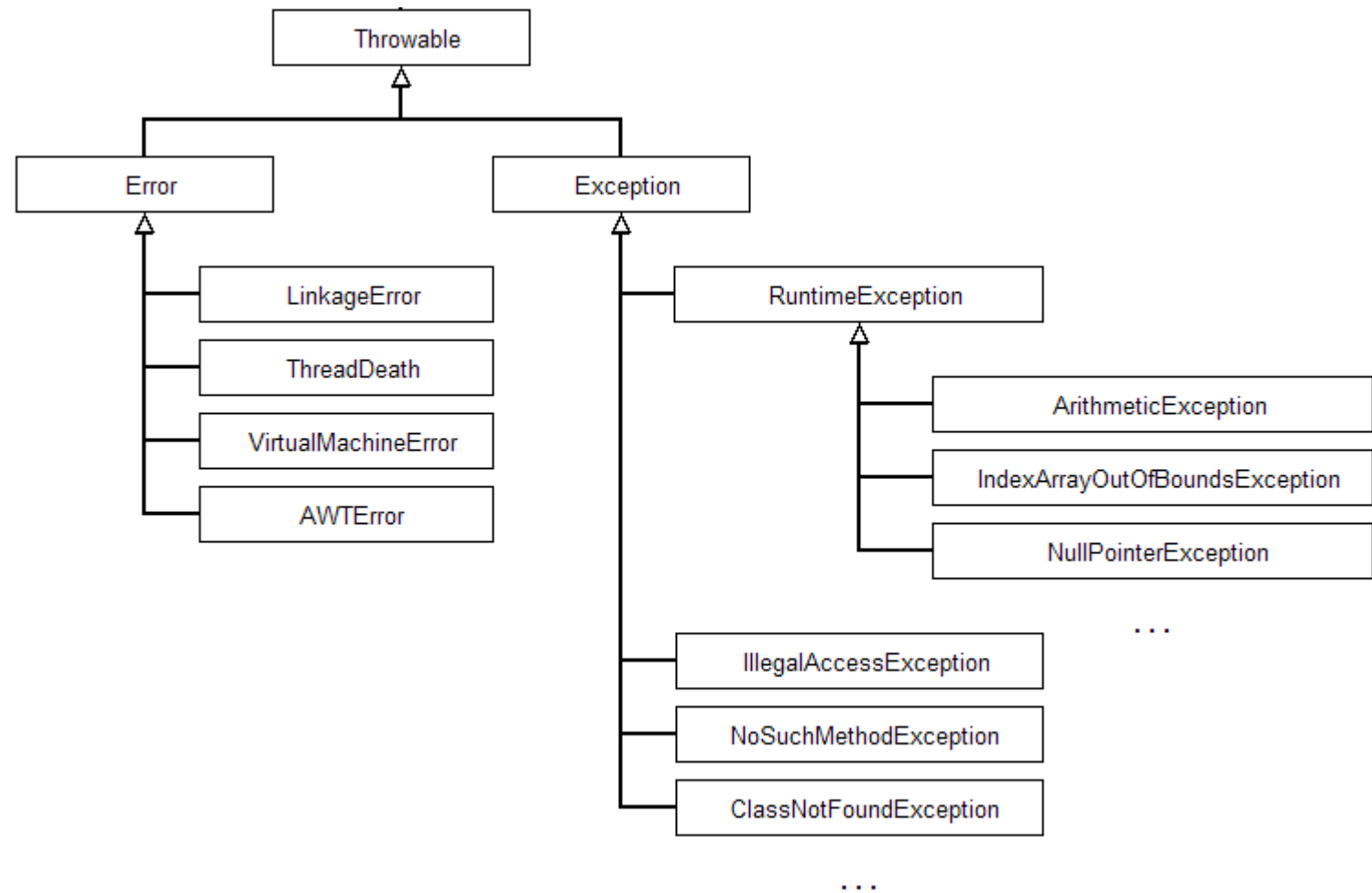
Si il y a plusieurs catch, il faudra les classer du plus spécifique au plus général, sinon une erreur sera signalé par le compilateur

Les exceptions

- Les classes liées aux erreurs/exceptions sont des classes placées dans l'arborescence d'héritage de la classe **java.lang.Throwable** (classe fille de **Object**)
- Le JDK fournit un certain nombre de telles classes
- Le programmeur peut en ajouter de nouvelles



Les exceptions



Error

- Les **Error** sont réservées aux erreurs qui surviennent dans le fonctionnement de la JVM
- Par exemple, un problème de mémoire **OutOfMemoryError**, une méthode qui n'est pas trouvée **NoSuchMethodError**
- Elles ne doivent donc être choisies par le programmeur que très rarement

Les exceptions

Quelques sous-classes de **RuntimeException**

- **NullPointerException**
- **IndexOutOfBoundsException**
- **ArrayIndexOutOfBoundsException**
- **ArithmeticException**
- **IllegalArgumentException**
- **NumberFormatException**
- **ClassCastException**
- **NoSuchElementException**

Les exceptions

Exceptions « contrôlées »

- Exceptions qui héritent de la classe **Exception** mais qui ne sont pas des **RuntimeException**
- Le compilateur vérifie que les méthodes utilisent correctement ces exceptions
- Toute méthode qui peut lancer une exception contrôlée, **doit** le déclarer dans sa déclaration:

```
int m() throws TrucException {
```

Throws VS throw

- **throws** : permet de signaler à la JVM qu'il faut utiliser un bloc `try {...} catch {...}`. Il est suivi du nom de la classe qui va gérer l'exception.

Utilisé dans certains cas où il est plus adéquat de traiter l'exception non pas dans la méthode où elle se produit, mais dans la méthode appelante.

NB : throws est nécessaire que pour les Exceptions (pas les erreurs ou les runtimes)

- **throw** : permet de lever une exception manuellement en utilisant un objet de type Exception (ou un objet hérité).

throw

L'instruction throw permet de générer une exception. Pour l'utiliser, il faut lui fournir une référence vers un objet de type Exception

Throw interrompt le déroulement régulier du programme, soit:

- Saute au bloc catch du bloc try « si il existe »
- Sinon, quite le programmme si elle n'est pas à l'intérieur d'un bloc try

Les exceptions

```
package ma.tets;
import java.io.IOException;

public class Maclasse {
    public static void mafonction() throws IOException {
        // corps de la fonction
        throw new IOException("message d'erreur");
    }

    public static void main(String[] args) {
        try {
            mafonction();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Les exceptions

- Constructeurs des exceptions

Par convention, toutes les exceptions doivent avoir au moins 2 constructeurs :

- un sans paramètre

ex: `public Throwable(){}`

- un autre dont le paramètre est une chaîne de caractères utilisée pour décrire le problème :

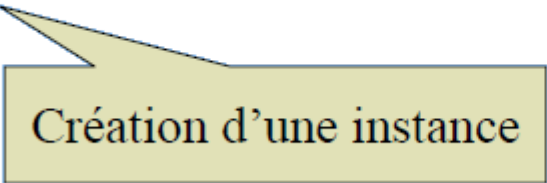
ex: `public Throwable(String message){ }`

Méthodes de la classe **Throwable**

- **getMessage()** retourne le message d'erreur associé à l'instance de **Throwable**
- **(getLocalizedMessage())** : idem en langue locale)
- **printStackTrace()** affiche sur la sortie standard des erreurs (**System.err**), le message d'erreur et la pile des appels de méthodes qui ont conduit au problème
- **getStackTrace()** récupère les éléments de la pile des appels

Les exceptions

```
public class Employe {  
    . . .  
    public void setSalaire(double salaire) {  
        if (salaire >= 0)  
            this.salaire = salaire;  
        else  
            throw  
                new IllegalArgumentException("Salaire négatif !");  
    }  
    . . .  
}
```



Création d'une instance

Créer une classe d'exception

- Le programmeur peut avoir à créer ses propres classes d'exception pour s'adapter mieux aux classes qu'il a créées
- Par convention, les noms de classes d'exception se terminent par « Exception »

Les exceptions

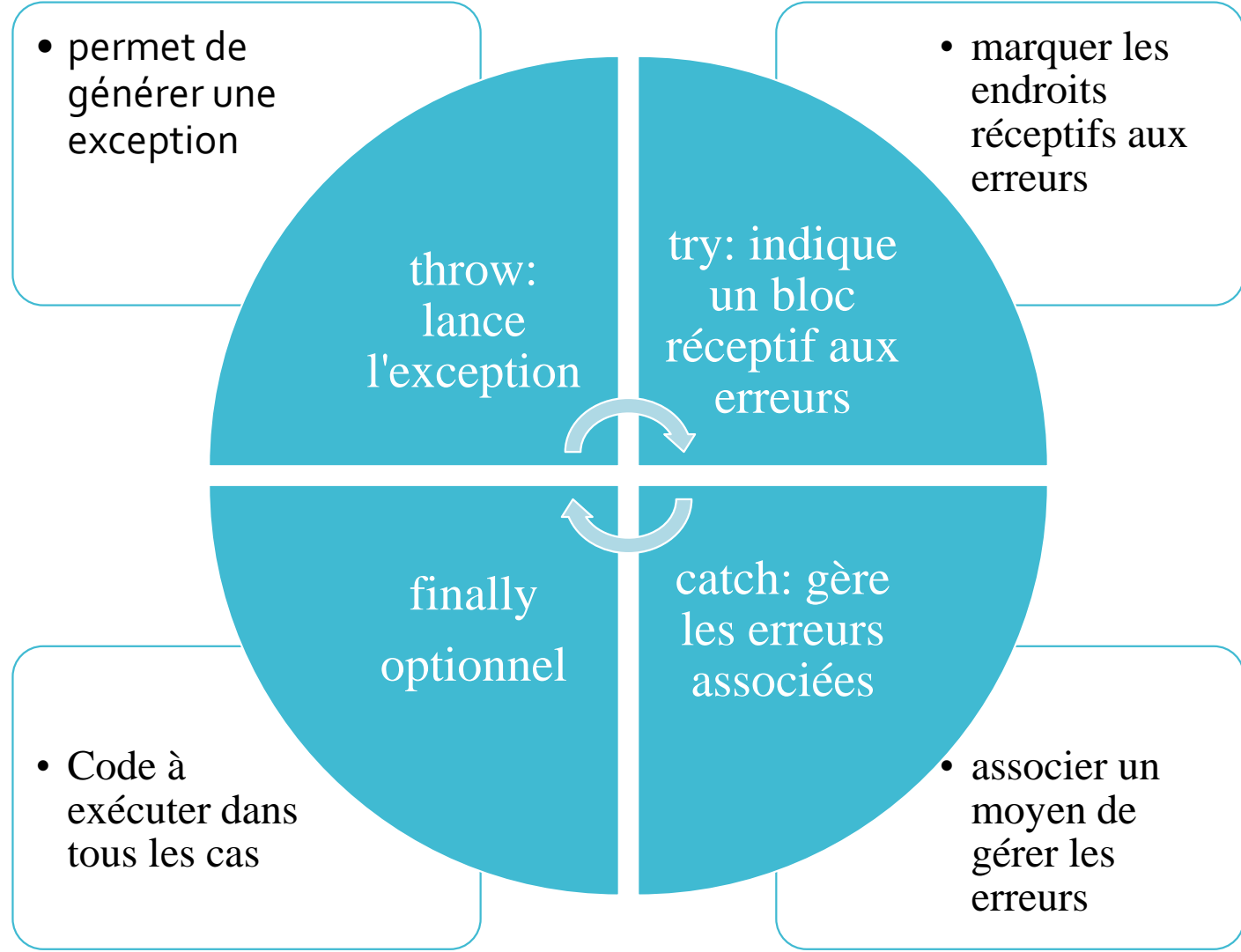
```
public class EtagerePleineException extends Exception {  
    private Etagere etagere;  
    public EtagerePleineException(Etagere etagere) {  
        super("Étagère pleine");  
        this.etagere = etagere;  
    }  
    . . . // autres constructeurs  
    public Etagere getEtagere() {  
        return etagere;  
    }  
}
```

Hériter de **Exception** et
pas de **Error** ou de
RuntimeException

N'oubliez pas d'ajouter au moins
le constructeur sans paramètre et
celui qui prend un message en paramètre

```
public class Etagere {  
    . . .  
    public void ajouteLivre(Livre livre)  
        throws EtagerePleineException {  
        if (nbLivres < livres.length)  
            livres[nbLivres++] = livre;  
        else  
            throw new EtagerePleineException(this);  
    }  
}
```

Les exceptions



Quizz

1. La classe père de Error est:

- Object
- Collections
- Throwable
- Exception

Quizz

1. La classe père de Error est:

- Object
- Collections
- Throwable
- Exception

Quizz

```
class Main {  
    public static void main(String args[]) {  
        int x = 0;  
        int y = 10;  
        int z = y/x;  
    }  
}
```

Il s'agit de:

- Erreur de compilation
- Pas d'erreur de compilation mais throws ArithmeticException exception

Quizz

```
class Main {  
    public static void main(String args[]) {  
        int x = 0;  
        int y = 10;  
        int z = y/x;  
    }  
}
```

Il s'agit de:

- Erreur de compilation
- Pas d'erreur de compilation mais throws ArithmeticException exception

Quizz

```
class Test{
    public static void main (String[] args) {
try {
        int a = 0;
        System.out.println ("a = " + a );
        int b = 10 / a;
        System.out.println ("b = " + b);
    }
catch(ArithmeticException e)
    { System.out.println ("diviser par 0!"); }
finally
    { System.out.println ("je suis à l'intérieur de finally");} } }
```

Quizz

- Erreur de compilation
- diviser par 0!
- $a = 0$
diviser par 0!
je suis à l'intérieur de finally
- $a = 0$

Quizz

- Erreur de compilation
- diviser par 0!
- $a = 0$
diviser par 0!
je suis à l'intérieur de finally
- $a = 0$

Exercice d'application

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.NoSuchElementException
                    java.util.InputMismatchException
```

- en utilisant la classe `IllegalArgumentException` et `InputMismatchException`
- On veut écrire la fonction **saisirage** qui permet de saisir correctement l'âge d'un adulte de plus de 18 ans.
- Si l'utilisateur saisit une donnée dont le format n'est pas celui d'un entier, le programme lève une exception.
- Le nombre de tentative est limité à 3

Interface graphique

- La quasi-totalité des programmes informatiques nécessitent
 - l'affichage de questions posées à l'utilisateur
 - l'entrée de données par l'utilisateur au moment de l'exécution
 - l'affichage d'une partie des résultats obtenus par le traitement informatique
- Cet échange d'informations peut s'effectuer avec une interface utilisateur (UI en anglais) en mode texte (ou console) ou en mode graphique

Interface graphique

- Une interface graphique est formée d'une ou plusieurs fenêtres qui contiennent divers composants graphiques (*widgets*) tels que
 - boutons
 - listes déroulantes
 - menus
 - champ texte
 - etc.
- Les interfaces graphiques sont souvent appelés GUI d'après l'anglais *Graphical User Interface*

Interface graphique

- L'utilisation d'interfaces graphiques impose une façon particulière de programmer
- La programmation « conduite par les événements » est du type suivant :
 - les actions de l'utilisateur engendrent des événements qui sont mis dans une file d'attente
 - le programme récupère un à un ces événements et les traite

les écouteurs

- Le JDK utilise une architecture de type « observateur - observé » :
 - les composants graphiques (comme les boutons) sont les observés
 - chacun des composants graphiques a ses observateurs (ou écouteurs, listeners), objets qui s'enregistrent (ou se désenregistrent) auprès de lui comme écouteur d'un certain type d'événement (par exemple, clic de souris)

Rôle d'un écouteur

- Il est prévenu par le composant graphique dès qu'un événement qui le concerne survient sur ce composant
 - Il exécute alors l'action à effectuer en réaction à l'événement
 - Par exemple, l'écouteur du bouton « **Exit** » demandera une confirmation à l'utilisateur et terminera l'application

Les API utilisées pour les interfaces graphiques en Java

2 bibliothèques :

- AWT (*Abstract Window Toolkit*, JDK 1.1)
- Swing (JDK/SDK 1.2)
- Swing et AWT font partie de JFC (*Java Foundation Classes*) qui offre des facilités pour construire des interfaces graphiques
- Swing et de AWT possèdent la même gestion des événements
- les classes de *Swing* héritent des classes de AWT

Paquetages principaux

- AWT : **java.awt** et **java.awt.event**
- Swing : **javax.swing**, **javax.swing.event**, et tout un ensemble de sous-paquetages de **javax.swing** dont les principaux sont
 - liés à des composants ; **table**, **tree**, **text** (et ses sous-paquetages), **filechooser**, **colorchooser**
 - liés au *look and feel* général de l'interface (plaf = *pluggable look and feel*) ; **plaf**, **plaf.basic**, **plaf.metal**, **plaf.windows**, **plaf.motif**

GUI

```
import javax.swing.JFrame;
```

```
public class Fenetre extends JFrame {
```

```
    public Fenetre() {
```

```
        super("Une fenêtre");
```

```
        setSize(300, 200);
```

```
        pack();
```

```
        setVisible(true);
```

```
    }
```

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

```
        JFrame fenetre = new Fenetre();
```

```
    }
```

```
}
```

ou
setTitle("...")

ou setBounds(...)

compacte le contenu de la fenêtre
(annule setSize)

affiche la fenêtre

GUI

- Pour afficher des fenêtres (instances de **JFrame**), Java s'appuie sur les fenêtres fournies par le système d'exploitation hôte dans lequel tourne la JVM
 - Les composants Java qui, comme les **JFrame**, s'appuient sur des composants du système hôte sont dit « lourds »
 - L'utilisation de composants lourds améliore la rapidité d'exécution mais nuit à la portabilité et impose les fonctionnalités des composants

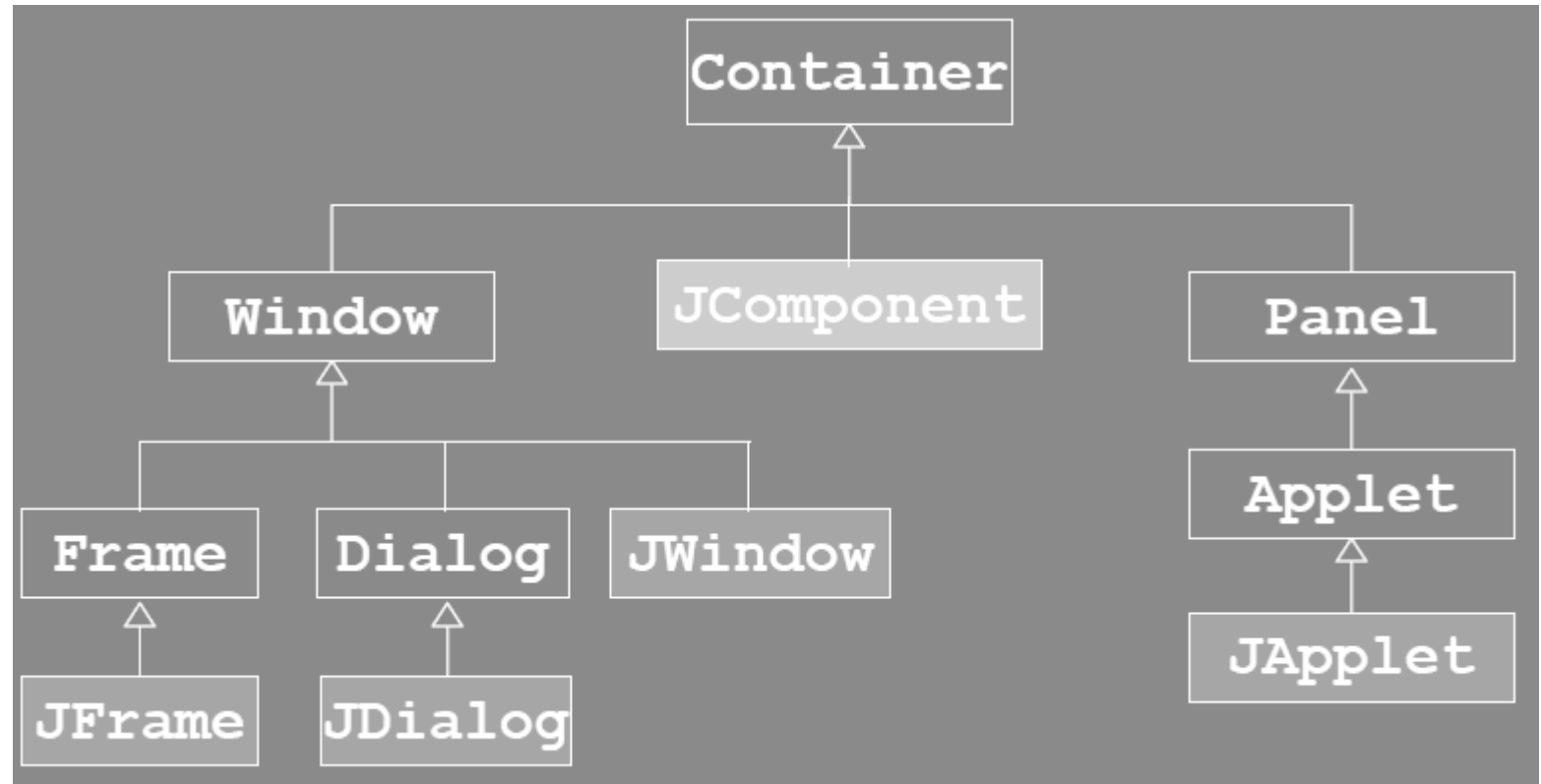
GUI

Il y a 3 sortes de containers lourds (un autre, **JWindow**, est plus rarement utilisé) :

- **JFrame** fenêtre pour les applications
- **JApplet** pour les *applets*
- **JDialog** pour les fenêtres de dialogue

Pour construire une interface graphique avec Swing, il faut créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers qui forment l'interface graphique

GUI



GUI

L'utilisateur utilise le clavier et la souris pour intervenir sur le déroulement du programme

- Le système d'exploitation engendre des événements à partir des actions de l'utilisateur
- Le programme doit lier des traitements à ces événements

GUI

- Les événements sont représentés par des instances de sous-classes de **java.util.EventObject**
- Événements liés directement aux actions de l'utilisateur : **KeyEvent**, **MouseEvent**
- Événements de haut niveau :
 - **FocusEvent**
 - **WindowEvent** « fenêtre ouverte, fermée, icônifiée ou désicônifiée »
 - **ActionEvent** « Déclenchent, une action »
 - **ItemEvent** « choix dans une liste, dans une boîte à cocher »
 - **ComponentEvent** « Composant, déplacé, retaillé, caché ou montré »

GUI

- Chacun des composants graphiques a ses observateurs (ou écouteurs, *listeners*)
- Un écouteur doit s'enregistrer auprès des composants qu'il souhaite écouter, en lui indiquant le type d'événement qui l'intéresse (par exemple, clic de souris)
- Il peut ensuite se désenregistrer

ActionEvent

Cette classe décrit des événements de haut niveau qui vont le plus souvent déclencher un traitement (une *action*) :

- clic sur un bouton
- *return* dans une zone de saisie de texte
- choix dans un menu

Ces événements sont très fréquemment utilisés et ils sont très simples à traiter

=> Un objet *ecouteur* intéressé par les événements de type « action » (classe **ActionEvent**) doit appartenir à une classe qui implémente l'interface : **java.awt.event.ActionListener**

Méthode **getSource**

- L'écouteur peut interroger l'événement pour lui demander le composant qui l'a généré
- La méthode « **public Object getSource()** » de **EventObject** renvoie le composant d'où est parti l'événement, par exemple un bouton ou un champ de saisie de texte
- Souvent indispensable si l'écouteur écoute plusieurs composants

Gestion des événements

Ce mécanisme de gestion comporte trois éléments :

- la *source de l'événement* (e.g. un bouton)
- l'*objet de l'événement*
- l'*écouteur d'événement*.

Pour spécifier qu'un événement vient de se produire, la source envoie un objet événementiel à l'écouteur qui pourra l'utiliser pour répondre à l'événement.

Pour gérer un événement, le programmeur doit inscrire un écouteur d'événement (souvent à l'aide de la méthode `addActionListener`) et définir un gestionnaire d'événement (dont le nom est passé comme paramètre à `addActionListener`)

GUI

- Dans cet exemple, la classe ButtonHandler (qui constitue le gestionnaire d'événement) doit être définie par le programmeur et doit implémenter l'interface ActionListener. La méthode actionPerformed de cette interface doit dès lors être définie :

```
class ButtonHandler implements ActionListener
{
    public void actionPerformed(ActionEvent evenement)
    { ... }
}
```

GUI: exemple

un clic sur un bouton sera g  r   de la mani  re suivante :

- `Jbutton bouton = new Jbutton("Bouton");`
- `ButtonHandler gestionnaire = new ButtonHandler();`
- `bouton.addActionListener(gestionnaire);`
- Parmi les types d'  v  nements, on retrouve `ActionEvent`, `MouseEvent`, `KeyEvent`, `ItemEvent`, etc.    chacun de ces   v  nements est associ   un   couteur correspondant et de nom semblable (e.g. `ActionListener` <> `ActionEvent`)

Exercice d'application

Créer deux classes:

- une classe driver contenant la fonction main()
- Et une classe « Container » qui contient un JButton, un JLabel et un JTextField

Telle que:

- Le JTextField permet de saisir un texte
- En cliquant sur le JButton , le texte saisi dans le JTextField sera affiché au niveau du JLabel