



Perfectionnement JAVA



Lionel Cavaglia

Développeur JAVA fullstack depuis 2008

@ : cavaglia.lionel@gmail.com

Plan

3

1

Héritage,
polymorphisme, Classes
abstraites

4

Multithreading en JAVA

2

Les collections

5

Les interfaces
graphique avec SWING

3

Les Exceptions

1 Héritage, Polymorphisme, Classes abstraites

Plan

- Rappel classe et objets*
- Héritage*
- Polymorphisme*
- Classe abstraite*
- Interfaces*

Objectifs

- Savoir écrire des classes qui ont des liens d'héritage*
- Redéfinir des comportements grâce à l'héritage*
- Utiliser le polymorphisme pour sélectionner les comportements d'une instance de classe.*

Principes de base de la POO

Classe et Objet :

- *Classe → description des modèles*
classe = nom + propriétés + méthode
propriétés = variables d'instances - attributs -
champs
- *Objet → réalisation concrète de la classe.*
 - instance de la classe.*
 - Structuré par les propriétés de la classe et exécute les méthodes.*
- *Toute classe hérite de la classe java.lang.Object*

Approche objet

Pour épouser l'approche objet:

- *Chaque classe donne lieu à diverses instances (objets)*
- *Il faut se mettre à la place des objets de manière à mettre en évidence les propriétés qui doivent structurer chaque objet et mieux définir le rôle de chaque méthode.*
- *Chaque objet doit être responsable de son propre état (Principe d'encapsulation)*

Structures fondamentales

Rappel : Conventions de codage camelCase

➤ Les classes et interfaces :

en minuscules avec chaque première lettre des mots en majuscules.

*Ex : **MaClasse***

➤ Les attributs :

en minuscules. première lettre d'un mot en majuscules, sauf pour le premier mot

*Ex : **public type** monAttribut;*

➤ Les constantes :

en majuscules avec l'underscore _ pour séparer les mots

*Ex : **public final type** MA_CONSTANTE = valeur;*

➤ Les méthodes :

Même conventions utilisées pour les attributs

```
public class Personne
{
    private String nom;
    private String prenom;

    public Personne(String n, String p){
        nom = n;
        prenom = p;
    }

    public static void main (String args[])
    {
        Personne P1 = new Personne("toto", "titi");
    }
}
```

⇐ *Déclaration de la classe*

- *Constructeur (même nom que la classe)*
- *L'opérateur « new »*

Exemple:

Personne P1 = new Personne();

*A l'appel de l'opérateur **new** :*

- 1- allocation mémoire de l'objet avec ses variables*
- 2- appel du constructeur. Recopie les valeurs.*
- 3- new retourne l'adresse de l'objet (référence) construit*

Création d'un objet

Personne p1 = new Personne("toto", "titi");

- *Création de la variable **p1** en mémoire, initialisée à **null**.*
- *Exécution de « **new** »:*
 - 1- *allocation mémoire de l'objet avec ses variables*
 - 2- *appel du constructeur. Recopie les valeurs.*
 - 3- *new retourne l'adresse de l'objet construit*

Définition de l'héritage

Une classe **B** qui hérite d'une classe **A** hérite des attributs et des méthodes de la classe **A** sans avoir à les redéfinir.

B est une sous-classe de **A** ou **B** est une classe dérivée de la classe **A**, **B** étend la classe **A** ou **B** est une classe fille.

A est la super-classe de **B**, la classe de base ou **A** est une classe parente ou classe mère.

Une classe ne peut avoir qu'une seule super-classe. Il n'y a pas d'héritage multiple en Java. Par contre, elle peut avoir plusieurs sous-classes.

- On utilise l'héritage lorsqu'on définit un objet, par exemple *Etudiant* qui «est-un» autre objet de type par exemple *Personne* avec plus de fonctionnalités qui sont liés au fait que l'objet soit un étudiant
- L'héritage permet de réutiliser dans la classe *Etudiant* le code de la classe *Personne* (lorsque l'on dit que *Etudiant* hérite de *Personne*) sans toucher au code initial : on a seulement besoin du code compilé
- L'héritage minimise les modifications à effectuer : on indique seulement ce qui a changé dans *Etudiant* par rapport au code de *Personne*, on peut par exemple
 - ➔ rajouter de nouvelles variables
 - ➔ rajouter de nouvelles méthodes
 - ➔ modifier certaines méthodes

Ce que peut faire une classe fille :

La classe qui hérite peut

- *ajouter des variables,*
- *ajouter des méthodes et des constructeurs*
- *redéfinir des méthodes*
- *surcharger des méthodes*

** mais elle ne peut pas retirer une variable ou une méthode*

Appels des méthode des classe mère et fille

This

❖ Le mot clé *this* désigne l'objet courant:

this.leNombre = leNombre // la variable *leNombre* précédée de *this* est la variable d'instance de cet objet, alors que *leNombre* sans le *this* représente une variable quelconque qui n'est pas la variable d'instance de l'objet pointé par *this*.

Super

❖ Le mot clé *super* désigne la superclasse:

super.getPrix() // permet d'appeler la méthode *getPrix()* définie dans la superclasse

Cas particulier des constructeurs

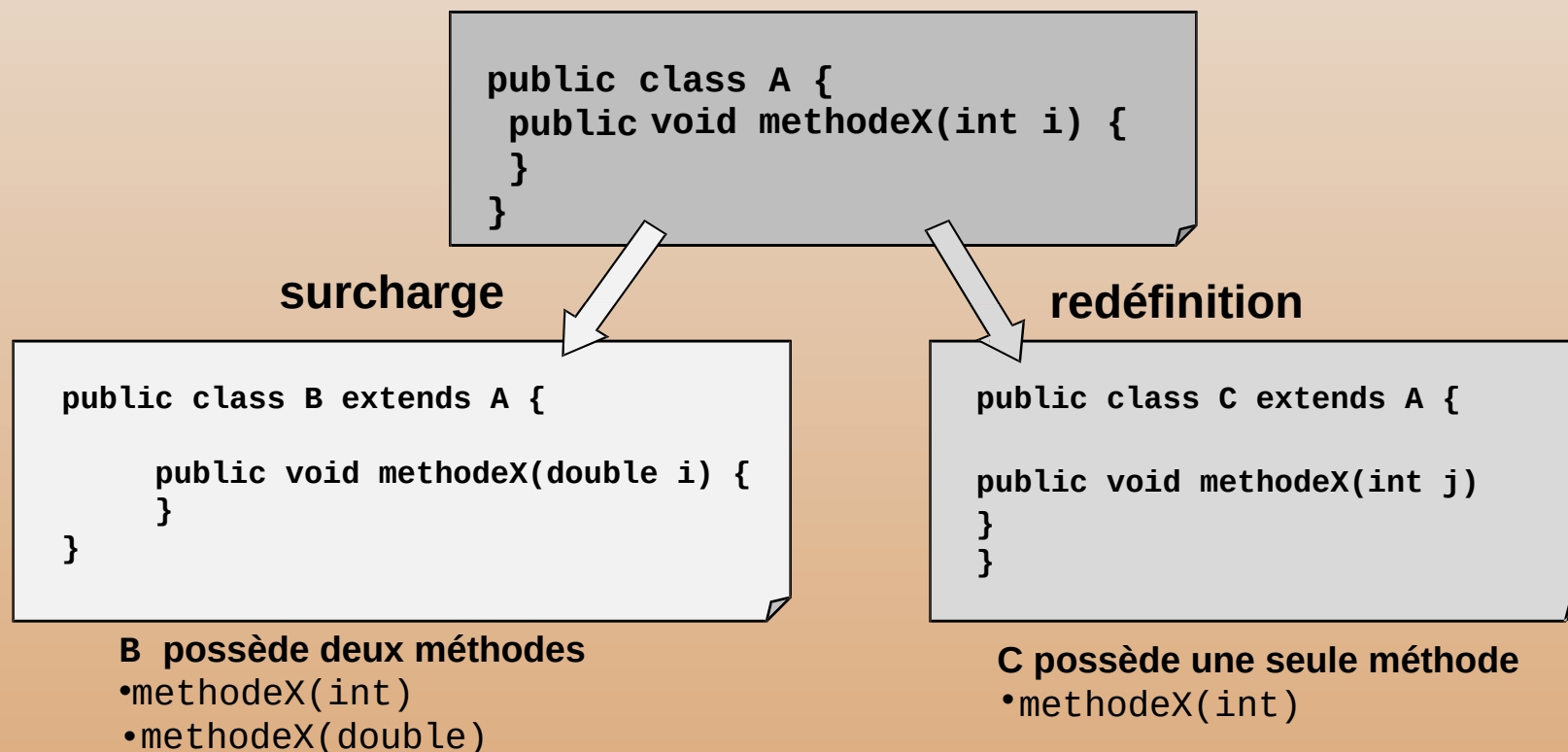
❖ Un constructeur peut appeler un autre constructeur

- *this(arg1,...)* // appel du constructeur d'argument *arg1,...* de l'objet en cours de construction.
- *super(arg1,...)* // appel du constructeur d'argument *arg1,...* de la superclasse.

Redéfinition de méthode

La redéfinition consiste à réécrire une méthode déclarée dans une classe mère et d'y inscrire un nouveau code au sein de la classe fille.

La méthode réécrite doit présenter la même signature que la méthode de la classe mère



ATTENTION : Ne pas confondre **redéfinition** (overriding) avec **surchage** (overloading)

Polymorphisme

Question

- Reprenons l'exemple de la classe *Etudiant* qui hérite de la classe *Personne*:
Soit une méthode *getNom()* de *Personne* qui est redéfinie dans *Etudiant*
- Quelle méthode *getNom()* sera exécutée dans le code suivant, celle de *Personne* ou celle de *Etudiant*?

Personne a = new Etudiant(5); // a est un objet de la classe
a.getNom();

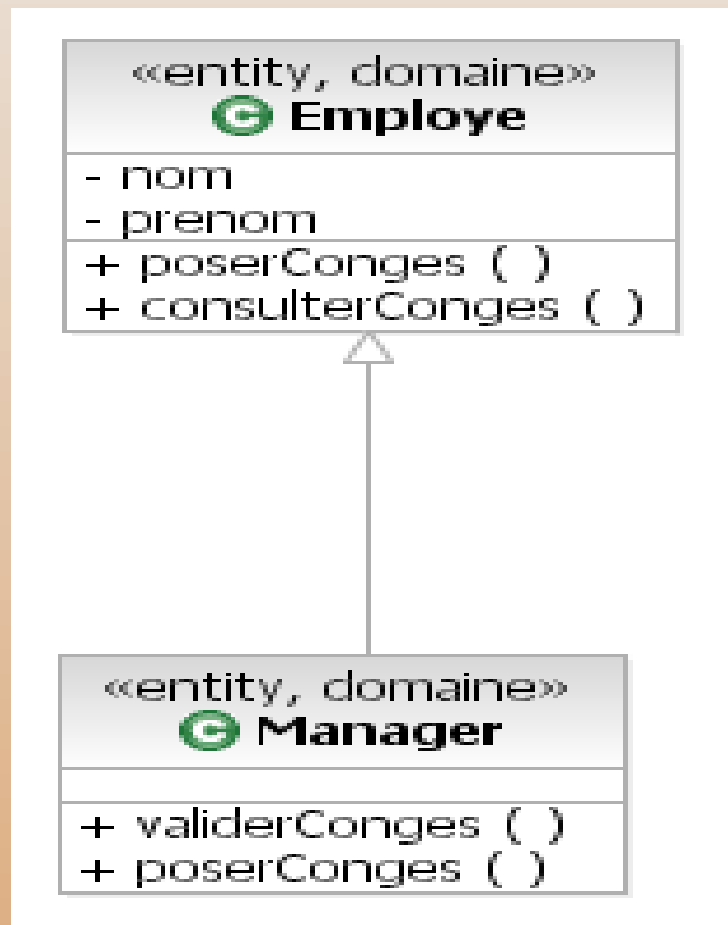
Etudiant mais il est déclaré de la classe *Personne*

La méthode appelée ne dépend que du type réel (*Etudiant*) de l'objet *a* et pas du type déclaré (ici *Personne*)

C'est la méthode de la classe *Etudiant* qui sera exécutée

Exemple de polymorphisme

- La méthode **poserConges()** pour l'employé déclenche une demande avec une attente de validation
- La méthode **poserConges()** pour le manager déclenche une demande qui est automatiquement validée



Abstraction : Interface

- *Jeu de condition pour les classes qui veulent se conformer (contrat)*
- *Sorte de classe abstraite dont toutes les méthodes sont implicitement **public abstract**.*
- *Touts les champs sont des constantes : **public static final***
- *Impossible d'instancier une interface*
- *Mot clés : **interface** / **implements***

```
public interface Motorise { // notre interface
    public void faisLePlein() ;
}

public class Transport {
    // une instance de Transport ne sait pas toujours
    // faire le plein
    public void roule() {}
}

public class Voiture extends Transport
    implements Motorise {
    public void conduit() {}
    public void faisLePlein() {}
}
```

Abstraction : Classe Abstraite

Définition

- La définition d'une *classe abstraite* est entre la définition d'une *classe «normale»* et d'une *interface*
 - ➔ Elle déclare un comportement
 - ➔ Elle ne définit pas d'implémentation
- Pour définir une *classe abstraite*, on utilise le mot clé *abstract*

exemple :

```
public abstract class Shape
```

Classe Abstraite : règles de déclaration

19

Méthode abstraite

Une méthode sans implémentation est obligatoirement *abstraite* et est défini par le mot clé *abstract*

```
public abstract void draw();
```

Classe abstraite

Une classe dont une méthode est *abstraite* est obligatoirement *abstraite* et est donc défini par le mot clé *abstract*

```
public abstract class Shape  
{  
    public abstract void draw();  
}
```

Une classe *abstraite* peut mélanger des méthodes *abstraites* et des méthodes «normales»

•Exemple :

```
public abstract class Shape {  
    ...  
    // recyclage de l'implémentation  
    public Point getPosition() {  
        return posn;  
    }  
    // recyclage de l'interface
```

Méthode « normales » et abstraites

Utilisation

- Pour utiliser les méthodes d'une classe abstraite, on doit passer par l'héritage
- Dans ce cas, on doit fournir le code de toutes les méthodes abstraites de la super-classe *abstraite*

```
public class Rectangle extends Shape {  
...  
  
    public void draw() {  
        ...    // code pour afficher un rectangle  
    }  
}
```

Obligation de fournir le code
Pour les méthodes **abstract**

```
}
```

Première manipulation de Dates

*La classe **LocalDate** de l'api **JAVA** est une des classes qui permet la manipulation de dates.*

- *Initialisation d'un **LocalDate** a l'instant présent :*

```
LocalDate date = LocalDate.now();
```

- *On peut y rajouter une conversion depuis un string :*

```
LocalDate date = LocalDate.parse("2020-01-01");
```

```
LocalDate date =
```

```
LocalDate.parse("01/01/2020", DateTimeFormatter.ofPattern("d/MM/yyyy"));
```

- *On peut récupérer les valeurs des jour / mois / année sous forme de **int** avec les méthodes **getYear()**, **getDayOfYear()** et **getMonth()***

- *On peut les comparer avec les méthodes **isBefore()** et **isAfter()***

2

Les Collections

Plan

- *Types de collections*
- *Listes*
- *Collection clé / valeur*

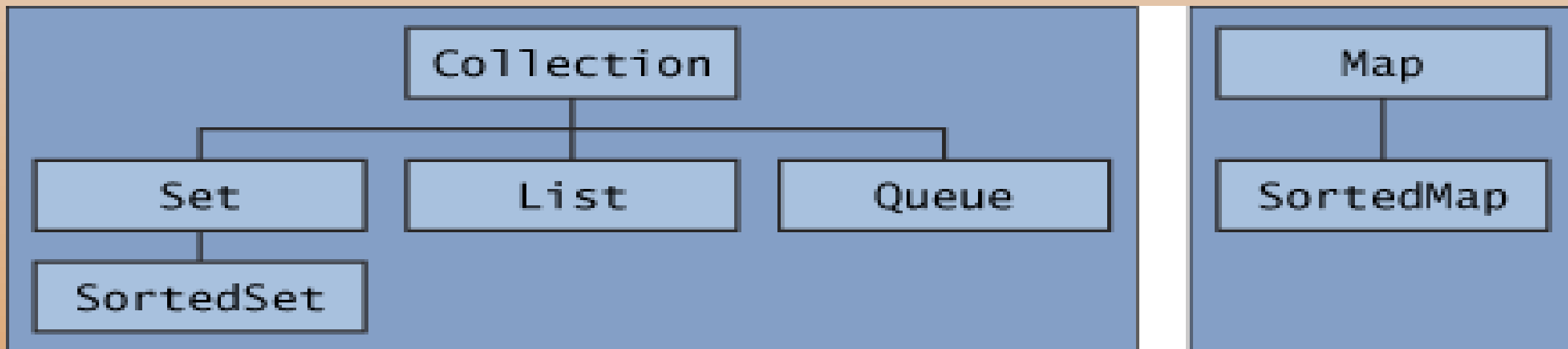
Objectifs

- *Connaitre les différents types de collections*
- *Savoir déclarer et réaliser des opérations sur les collections*
- *Manipuler les collections de manière approprié selon le besoin de l'algorithme*

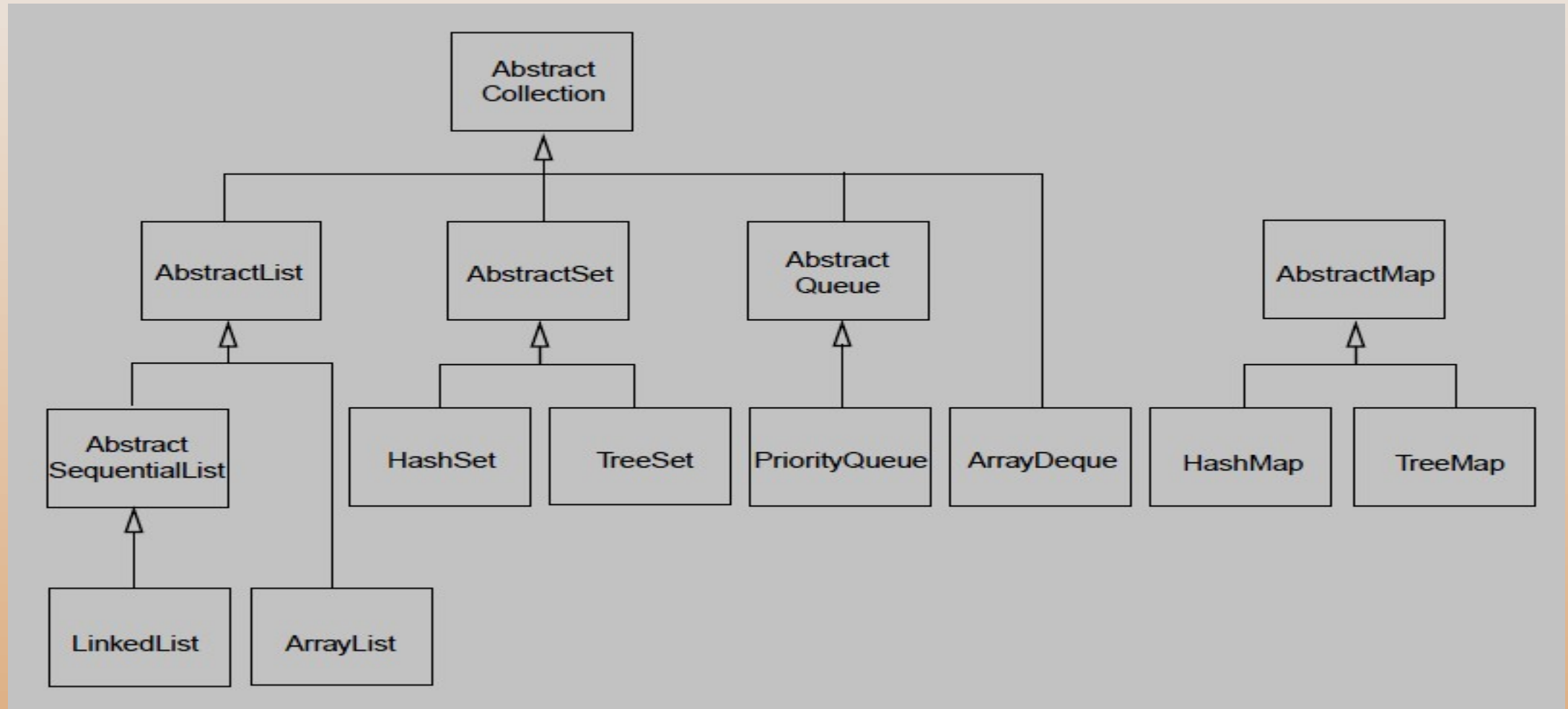
Types de collection

Il existe 2 grands ensemble de collection:

- Les listes pures avec des indexs numériques equivalentes à des tableaux. Ces collections vont descendre de la super interface **Collection** et ses dérivées
- Les listes de type «clé /valeurs » qui vont descendre de **Map**. Les clés et les valeurs peuvent être des objets



Classes abstraites



Classes Concrètes

Type de collection	Description
ArrayList	Une séquence indexée qui grandit et se réduit de manière dynamique
LinkedList	Une séquence ordonnée qui permet des insertions et des retraits effectifs à n'importe quel endroit
ArrayDeque	Une queue à deux extrémités implémentée sous forme de tableau circulaire
HashSet	Une collection non ordonnée qui refuse les répliquions
TreeSet	Un ensemble trié
EnumSet	Un ensemble de valeurs de type énuméré
LinkedHashSet	Un ensemble qui se souvient de l'ordre d'insertion des éléments
PriorityQueue	Une collection qui permet un retrait effectif de l'élément le plus petit
HashMap	Une structure de données qui stocke les associations clé/valeur
TreeMap	Une concordance dans laquelle les clés sont triées
EnumMap	Une concordance dans laquelle les clés appartiennent à un type énuméré
LinkedHashMap	Une concordance qui se souvient de l'ordre d'ajout des entrées
WeakHashMap	Une concordance avec des valeurs pouvant être réclamées par le ramasse-miettes si elles ne sont pas utilisées ailleurs
IdentityHashMap	Une concordance avec des clés comparées par ==, et non par equals

Utilisation des interfaces

- *Collection<E>*: *add, remove size toArray...*
- *Set<E>*: *éléments sans duplication*
 - *SortedSet<E>*: *ensembles ordonnés*
- *List<E>*: *des listes éléments non ordonnés et avec duplication*
- *Queue<E>*: *files avec tête: peek, poll (défiler), offer (enfiler)*
- *Map<K,V>*: *association clés valeurs*

En plus:

- *Iterator<E>*: *interface qui retourne successivement les éléments
next(), hasNext(), remove()*
- *ListIterator<E>*: *itérateur pour des List, set(E) previous, add(E)*

Interface Collection (dans l'API Java)

```
public interface Collection<E> extends Iterable<E> {  
    // operations de base  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator<E> iterator();  
  
    // operations des collections  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
  
    // Conversion en Tableau  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

Parcours d'une Collection

- *Implements Iterable<T>*

- *Contient la méthode `Iterator<T> iterator()`*

- *On peut parcourir les éléments par « for »:*

- ```
for (Object o : collection)
 System.out.println(o);
```

- *Ou avec un Iterator:*

- ```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Interface Set

- *Interface pour contenir des objets différents*
 - *Opérations ensemblistes*
 - *N'admet pas de doublons de valeur*
 - *SortedSet* pour des ensembles ordonnés
- *Implémentations:*
 - *HashSet* par hachage
 - *TreeSet* éléments – ordre ascendant
 - *LinkedHashSet* ordonnés par ordre d'insertion

Code de l'interface Set

```
public interface Set<E> extends Collection<E> {  
  
    // opérations de base  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator<E> iterator();  
  
    // autres  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
  
    // Array  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

Exemple avec Set

```
public static void chercheDoublons(String ... st){  
    Set<String> s = new HashSet<String>();  
    for (String a : st)  
        if (!s.add(a))  
            System.out.println("Doublon: " + a);  
  
    System.out.println("il y a "+s.size() + " mots différents: " + s);  
}
```

La méthode **add** retournera false si elle ne peut pas insérer une valeur. Cela signifie qu'une duplication de valeur est en cours

List

- *En plus de Collection:*
 - *Peut Contenir des éléments égaux*
 - *position déterminée pour chaque élément*

- *Implémentations:*
 - *ArrayList*
 - *LinkedList*

Code de *List*

```
public interface List<E> extends Collection<E> {
    // accès par position
    E get(int index);
    E set(int index, E element);
    boolean add(E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index,
        Collection<? extends E> c);

    // recherche
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // sous-liste
    List<E> subList(int from, int to);
}
```

Itérateur pour les objects List

```
public interface ListIterator<E> extends Iterator<E> {  
  
    boolean hasNext();  
    E next();  
    boolean hasPrevious();  
    E previous();  
    int nextIndex();  
    int previousIndex();  
    void remove();  
    void set(E e);  
    void add(E e);  
}
```

Map

- **Map** associe des clés à des valeurs
 - *Association injective: à une clé correspond exactement une valeur.*
 - *Trois implémentations, comme pour set*
 - *HashMap,*
 - *TreeMap,*
 - *LinkedHashMap*

Code de Map

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

3

Gestion des exceptions

Plan

- *Définition*
- *Types d'exception*
- *Gestion des exceptions*

Objectifs

- *Gérer les erreurs lors de l'exécution d'un programme*
- *Reconnaitre les types d'exceptions qui surviennent dans un programme*
- *Personnaliser des types d'exceptions*

Prévoir les erreurs d'utilisation

- Certains cas d'erreurs peuvent être prévus à l'avance par le programmeur.
exemples:
 - ✓ *erreurs d'entrée-sortie (I/O fichiers)*
 - ✓ *erreurs de saisie de données par l'utilisateur*

Le programmeur peut :

- *«Laisser planter» le programme à l'endroit où l'erreur est détectée*
- *Manifester explicitement le problème à la couche supérieure*
- *Tenter une correction*

Notion d'exception

- En Java, les erreurs se produisent lors d'une exécution sous la forme d'exceptions

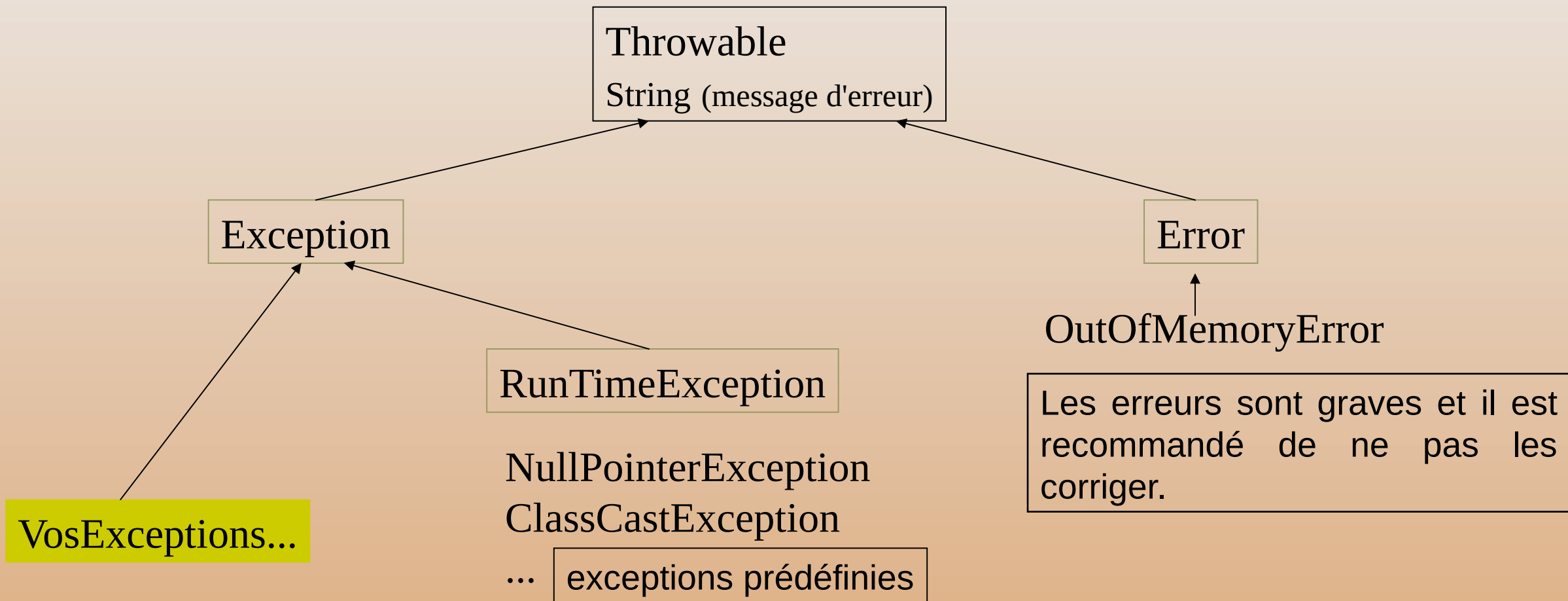
Une exception :

- *est un objet, instance d'une classe d'exception*
- *provoque la sortie d'une méthode*
- *correspond à un type d'erreur*
- *contient des informations sur cette erreur*

- Une exception est un signal qui indique que quelque chose d'exceptionnel est survenu en cours d'exécution.
- Deux solutions alors :
 - *laisser le programme se terminer avec une erreur*
 - *essayer, malgré l'exception, de continuer l'exécution normale.*
- Lever une exception consiste à signaler quelque chose d'exceptionnel.
- Capturer l'exception consiste à essayer de la traiter.

- ❑ Les exceptions construites par l'utilisateur étendent la classe *Exception*
- ❑ *RunTimeException*, *Error* sont des exceptions et des erreurs prédéfinies et/ou gérées par Java

Arbre des Exceptions



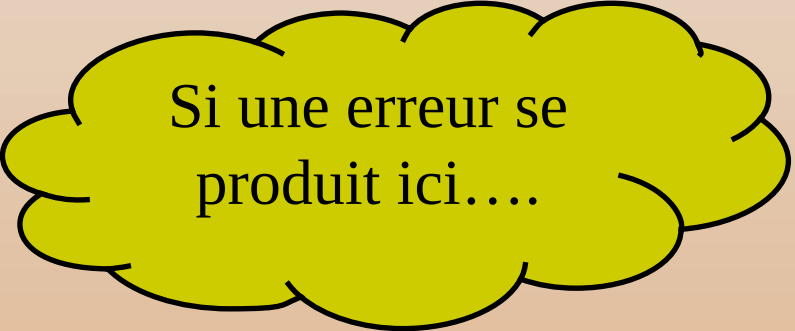
Quelques Exceptions prédéfinies

- ❑ Division par zéro pour les entiers : `ArithmeticException`
- ❑ Référence nulle : `NullPointerException`
- ❑ Tentative de forçage de type illégale : `ClassCastException`
- ❑ Tentative de création d'un tableau de taille négative : `NegativeArraySizeException`
- ❑ Dépassement de limite d'un tableau : `ArrayIndexOutOfBoundsException`

Capture d'une exception

- Les sections **try** et **catch** servent à capturer une exception dans une méthode
- exemple :

```
public void XXX(.....) {  
    try{ ..... }  
    catch {  
        .....  
        .....  
    }  
}
```



Si une erreur se produit ici....



On tente de récupérer là.

Bloc try...catch...finally

```
try
{
    ...
}
catch (<une-exception>)
{
    ...
}
catch (<une_autre_exception>)
{
    ...
}
...
finally
{
    ...
}
```

→ Autant de blocs **catch** que l'on veut.

→ Bloc **finally** facultatif.

Il sera exécuté peu importe ce qu'il se passe même si un throw est appelé

Clotûre des traitements avec finally

- Dès qu'une exception intervient, les traitements suivants dans le bloc try sont éludés.
 - *Il peut néanmoins être nécessaire d'effectuer d'autres traitements*
 - ➔ *libérer les ressources, comme la fermeture de la connexion vers un fichier*

```
try{  
    //Ecrire dans un fichier  
}catch(Exception e){  
    e.printStackTrace();  
}finally{  
    // Libération de la ressource (fermeture d'une connexion avec close())  
}
```

- Si l'exception n'est pas relancée mais traitée dans le bloc catch, l'exécution de la méthode se poursuit après le dernier catch

Rédéfinir un type d'exception

- *Pourquoi redéfinir un type ?*
 - *Dans un programme, ceci permet de différencier les erreurs Exceptions « systèmes » d'une exception « métier »*
 - *Permet de « tagguer » une erreur propre à l'exécution du programme que l'on écrit selon un cahier des charges défini*
- *La redéfinition se fait en héritant de la classe Exception.*
- *Une fois définie, on peut faire des throw de ce type*
- *Les bloc catch sera en capacité de pouvoir capturer notre nouveau type d'exception*

Levée d'une Exception

- Le mot clé **throw** permet de lever des Exception dans une méthode
- Le mot clé **throws** indique qu'un type d'exception est levée par une méthode, et donc que les méthodes appelante doivent traiter cette exception en cas d'apparition.

```
public void methodCall() {  
    try {  
        methodThatThrowsException();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

```
public void methodThatThrowsException() throws Exception {  
    throw new Exception("Je suis une Exception");  
}
```

4 *Multi-threading en Java*

Plan

- *Définition du multi-tâche*
- *Déclarer des Thread*
- *Synchronisation*

Objectifs

- *Connaitre les principes du multitâche en JAVA*
- *Maîtriser l'écriture de programme multitâche*
- *Comprendre les principes de synchronisation de tâche*
- *Opérer dans un environnement multitâche*

Système multitâche

- Du point de vue du processeur
 - *1 coeur = 1 tâche à la fois*
 - *Le temps d'exécution est partagé entre les programmes / threads*
- Du point de vue du programmeur, 2 unités d'exécution
 - *Le programme : application elle même auti gérée (gestion mémoire, threads, ...)*
 - *Le Thread :*
 - *unité de travail légère*
 - *obligatoirement rattachée à un processus*
 - *Partagent les même ressources (mémoire, accès aux fichiers,...)*
 - *Différentes branches d'exécution*

*Il existe toujours un **premier thread** qui commence à exécuter la fonction **main()***

Dans les interfaces graphiques

- *Il existe un autre thread qui attend les actions de l'utilisateur et déclenche les écouteurs ("listeners")*
- *Il existe un autre thread qui redessine les composants graphiques qui ont besoin de l'être*

On peut créer ses propres threads

La classe Thread

- Pour créer un thread en JAVA il nous faut instancier un objet de Type *java.lang.Thread*
- La classe **Thread** permet de manipuler un thread (démarrage, arrêt, suspension, reprise, ...)
- Cette classe a besoin d'un objet de type *Runnable* qui est une interface avec une méthode *run()*, qui va contenir le code d'exécution de la tâche à effectuer :

```
public interface Runnable {  
    public void run();  
}
```

Note : Pour tout thread, le but est l'exécution d'une méthode Run

Création et démarrage d'un Thread

1

Déclaration d'une classe implémentant Runnable



2

Instanciation de la classe Thread

```
Thread t = new Thread(new Compteur());
```

3

Démarrage du thread

```
t.start();
```

```
public class Compteur implements Runnable{

    @Override
    public void run() {
        int cpt = 0;
        while(cpt < 10) {
            try {
                Thread.sleep(1000);
                System.out.println("CPT="+cpt);
                cpt++;
            } catch (InterruptedException e) {
                System.err.println("Interrupted!!!");
                e.printStackTrace();
            }
        }
    }
}
```

Une fois créé, un thread ne fait rien tant qu'il n'a pas commencé avec start().

Note : la classe thread implémente l'interface Runnable donc il est possible de déclarer un thread en héritant de Thread et en redéfinissant la méthode run()

Race Condition

- *Dans un environnement multi-threadé, on peut avoir au moins 2 thread qui exécute la même portion de code sans qu'aucune mesure de synchronisation ne soit effectuée.*
 - Cette portion de code est appelée **Race condition**
- *C'est une situation risquée où l'état des variables peut être compromis car la lecture et l'écriture aléatoire des threads est imprévisible*
- *Solution : la **Synchronisation***

```
public class SimpleThread extends Thread {  
    private static volatile boolean flag = true;  
  
    public void run(){  
        flag = true;  
        if(flag != true) {  
            System.out.println("ARRFFF! flag false");  
        }  
        flag = false;  
    }  
  
    public static void main(String[] args) {  
        for(int i = 0; i < 100; i++)  
            new SimpleThread().start();  
        System.out.println("All Threads Started");  
    }  
}
```

Note : le mot clé **volatile** permet une meilleure gestion des lecture / écriture des variables par la JVM et favorise la mise a disposition des valeurs dans un environnement multithread

Cf : https://www.jmdoudoux.fr/java/dej/chap-acces_concurrents.htm

La synchronisation de thread

*La synchronisation consiste à **poser des verrous** autour des portions de code qui sont jugés « critiques » pour éviter leur exécution simultanée par plusieurs thread.*

Ces verrous vont garantir qu'un seul ou un nombre déterminé de thread pourront exécuter le code verrouillé. Ceci permettra de pouvoir prévoir et suivre l'évolution de l'état du programme sur le temps.

*La portion de code verrouillée s'appelle une **Section Critique** (ou Critical Section)*

Java met à disposition 2 méthodes pour poser des verrous :

- *Les « moniteurs »*
- *Les locks*

Les moniteurs

- Le mot clé **synchronized** permet de poser un verrou exclusif sur une portion de code : ceci permet de garantir que les accès à une ressource partagée ne se feront pas en concurrence.
- Pour la Synchronisation, on a besoin d'un thread et d'un moniteur.
 - En Java, le **Thread** est toujours le thread courant.
 - Le moniteur est précisé en utilisant l'instance d'un objet concerné. Un moniteur permet la mise en oeuvre de verrous implicites.
- La JVM garantit qu'un **bloc de code déclaré synchronized ne sera exécuté que par un seul thread** à un instant T sous réserve que le moniteur utilisé pour le verrou soit le même pour tous les threads.

```
Object monitor = new Object();

public void myMonitoredMethod {
    synchronized(monitor) {
        monitoredBoolean = false;
    }
}
```

Les méthodes synchronisées

- *Il est possible d'ajouter le mot clé `synchronized` à une méthode même si celle ci est statique.*
- *Comme pour un bloc `synchronized`, la JVM va garantir qu'un seul thread utilisateur de la méthode*
- *Déclaration d'une méthode synchronisée :*

```
public synchronized void monitoredMethod() {  
    monitoredBoolean = false;  
}
```
- *Cette écriture utilise l'objet courant comme moniteur et est équivalente à :*

```
public synchronized void monitoredMethod() {  
    synchronized (this) {  
        monitoredBoolean = false;  
    }  
}
```

L'interface Lock

- *Un Lock est un mécanisme de verrou qui permet un accès exclusif à une portion de code par un seul thread.*
- *Il permet de mettre en place des mécanismes de synchronisation similaires à ceux proposés par le mot clé synchronized mais avec la possibilité d'utiliser des fonctionnalités avancées.*
- *Le grand avantage d'utiliser un Lock est sa flexibilité pour obtenir ou non un verrou sans que cela soit obligatoirement bloquant comme dans le cas de l'utilisation du mot clé synchronized.*
- *L'utilisation d'un Lock est donc plus souple que l'utilisation du mot clé synchronized :*
 - *attente bloquante, non bloquante avec prise en compte possible de l'interruption du thread ou avec timeout, ...*
 - *distinguer les accès concurrents en lecture et mise à jour*
 - *support de conditions*
 - *les verrous peuvent être acquis et libérés dans n'importe quel ordre*

Les méthodes de l'interface Lock

void lock()	Obtenir le verrou : attente indéfinie si celui-ci est déjà pris
void lockInterruptibly()	Obtenir le verrou : attente jusqu'à son obtention ou si le thread courant est interrompu
Condition newCondition()	Obtenir une instance de type Condition associée à l'instance
boolean tryLock()	Obtenir le verrou immédiatement : pas d'attente. Elle renvoie un booléen qui indique si le verrou est obtenu
boolean tryLock(long time, TimeUnit unit)	Obtenir le verrou : attente maximale pour la durée précisée en paramètre ou si le thread courant est interrompu
void unlock()	Libérer le verrou

Exemple d'utilisation de Lock

```
private final Lock verrou = new ReentrantLock();
private boolean criticalBoolean = true;

public void methodeA() throws InterruptedException {
    verrou.lock();
    try {
        Thread.sleep(2000);
        criticalBoolean = false ;
        Thread.sleep(5000);
    } finally {
        verrou.unlock();
    }
}
```

Dans cet exemple, l'écriture du booléen est dans une section critique verrouillée par l'objet lock

On peut noter qu'il **faut absolument libérer** le verrou car sinon plus aucun thread ne pourra entrer dans la section critique donc le unlock est placé dans un finally qui sera exécuté même en cas d'exception levé par le code du try ou InterruptedException.

Interblocage

*Aussi appelé **Deadlock** ou **blocage mortel**, l'interblocage se produit lorsque des processus (ou Threads) concurrents s'attendent mutuellement.*

Exemple :

P1 et P2 sont 2 thread qui vont être lancés en parallèle. Ils ont tout 2 besoin de 2 ressources R1 et R2 partagées pour fonctionner

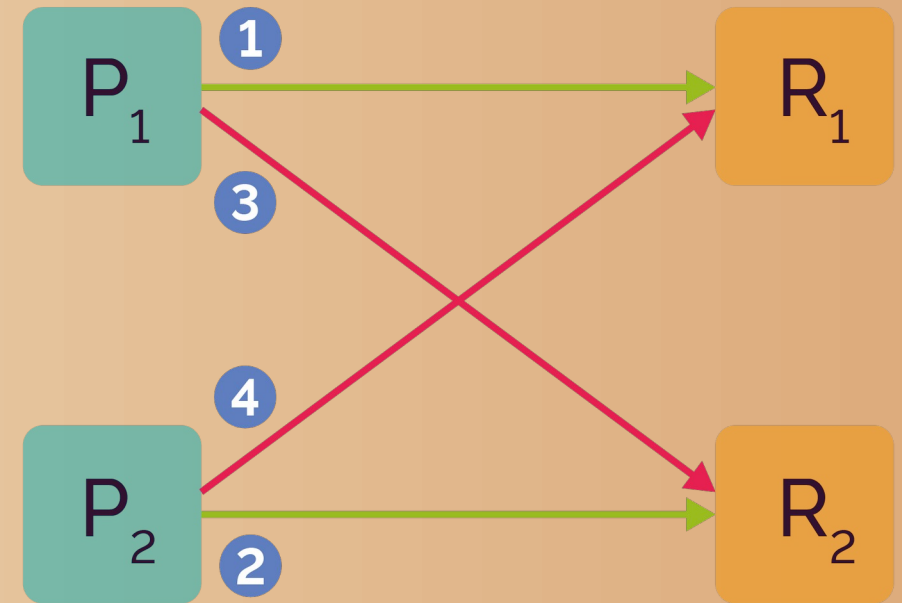
→ Au démarrage de l'application P1 prend la main et verrouille R1.

→ Puis c'est au tour de P2 qui verrouille R2

→ P1 cherche à acquérir R2 mais cette ressource est verrouillée, il se met donc à attendre

→ Idem pour P2 avec R1.

<=> On a une attente indéfinie !!



—→ ressource obtenue
—→ ressource non obtenue

5 Interface graphique en JAVA

Plan

- Définition de l'IHM*
- Éléments de construction d'une interface*
- Gestion des évènements*

Objectifs

- Apprendre à construire des programmes graphiques en JAVA*
- Savoir gérer les différentes interaction de l'utilisateur avec le programme*

Qu'est une IHM ?

- *Interface Homme – Machine : (Def ici)*

- *Dans le monde JAVA :*
 - *Package : javax.swing*

 - *Toutes les classes héritent de la classe 'Container' de AWT (Abstarct Windowing Toolkit)*

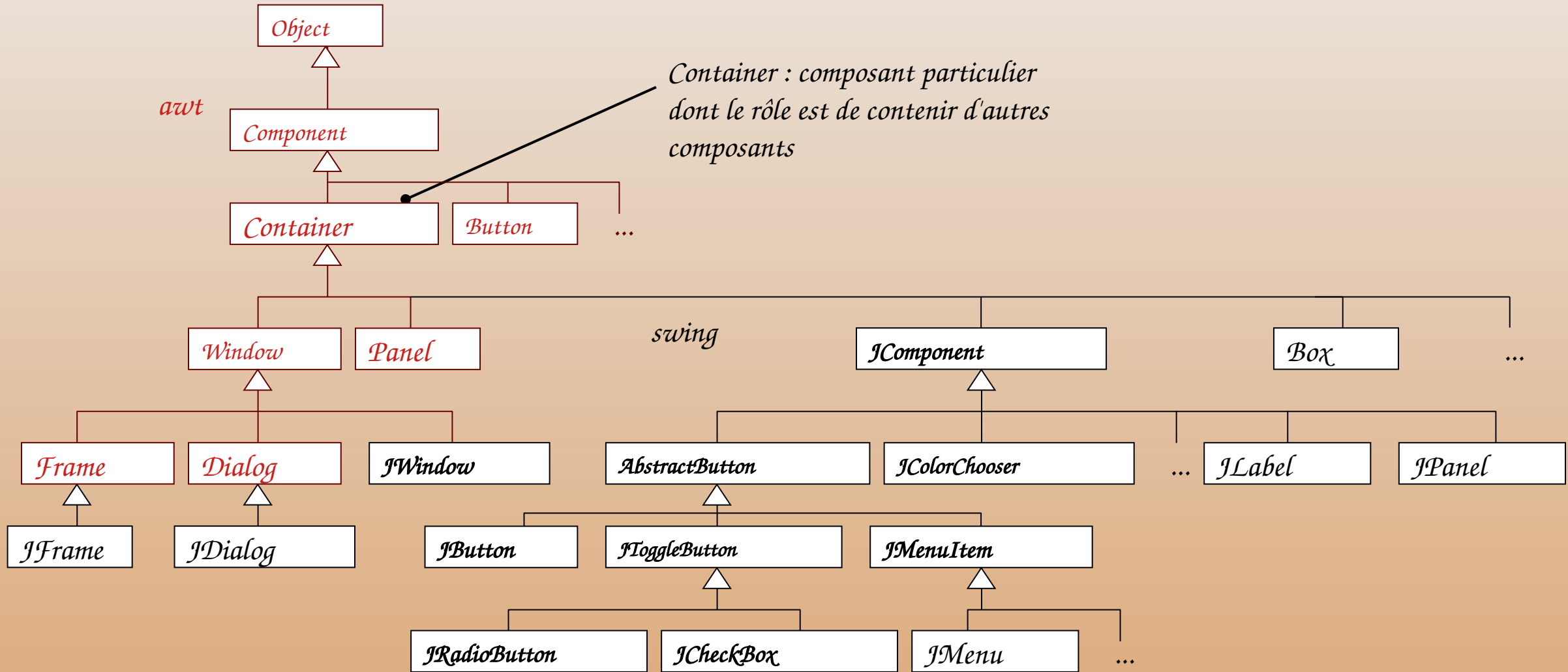
 - *2 types de classes :*
 - *les composants : Components*
 - *les conteneurs : Containers*

 - *Toutes les classes du package commencent par 'J' . Ex: JPanel*

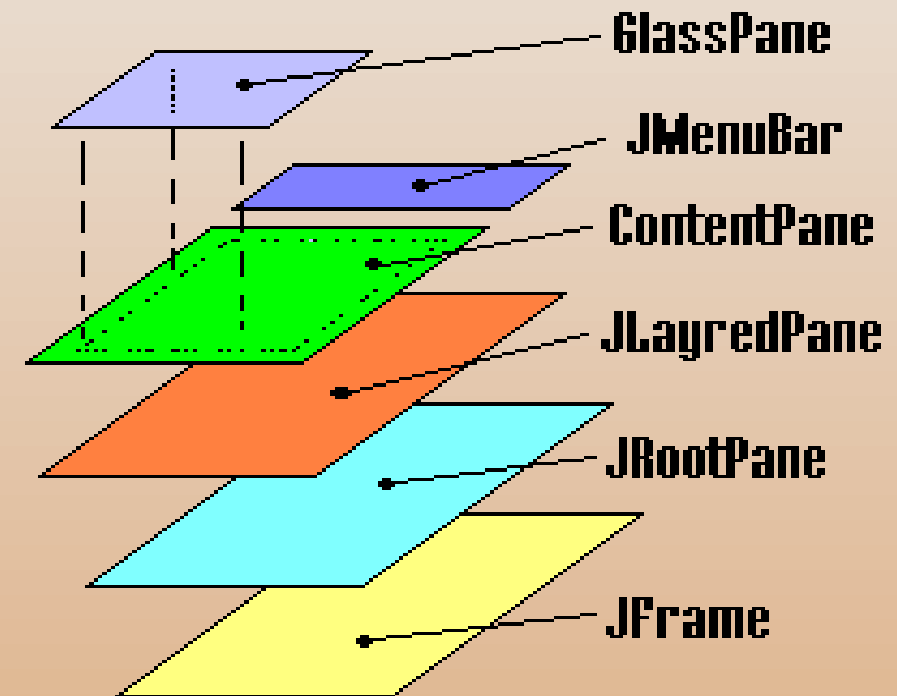
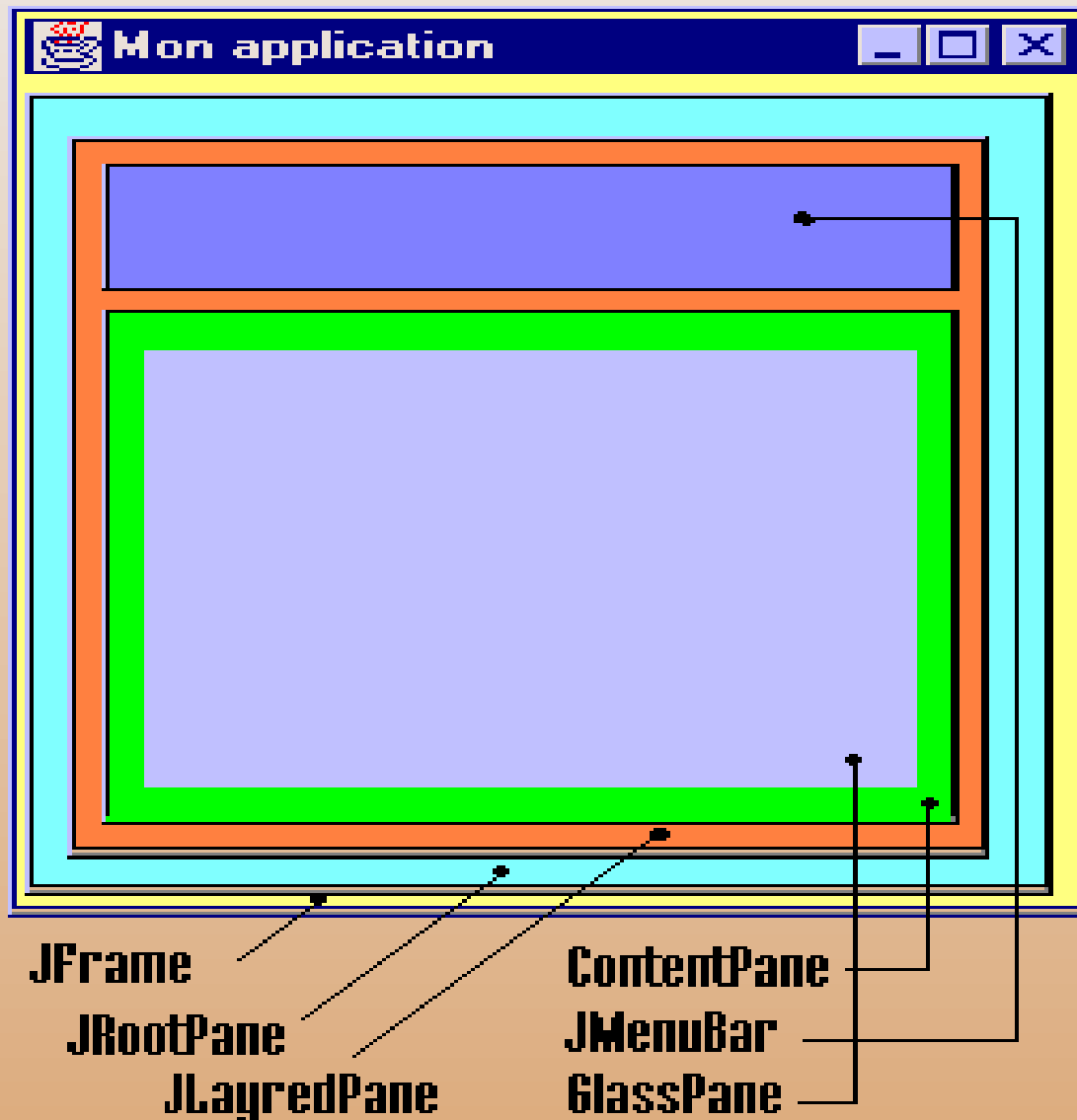
Structure de l'interface

- *La création de l'interface graphique passe **forcément** par une instance de la classe **JFrame***
- *Du point de vue du système d'exploitation cette fenêtre représente l'application.*
- *La fenêtre joue le rôle de « **conteneur** » dans lequel vont être disposés les différents éléments constitutifs (**composants**) de l'interface graphique de l'application (boutons, listes déroulantes, zone de saisie...)*
 - *ces éléments sont désignés sous les termes de*
 - *contrôles (IHM)*
 - *composants (components en JAVA)*

Arbre d'héritage des classes de *SWING*

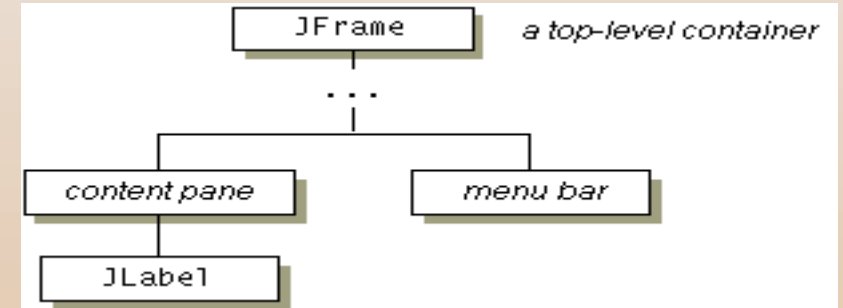
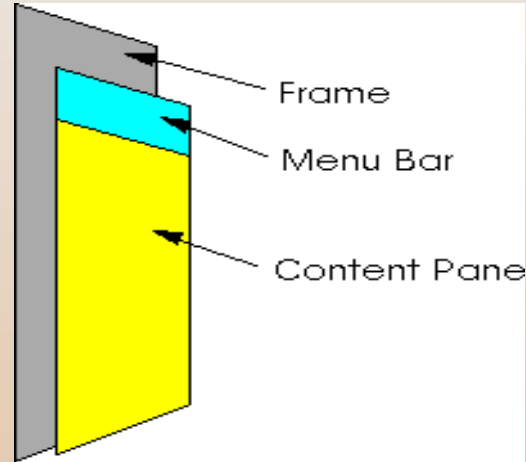


Décomposition d'une interface en couches



Les Conteneurs SWING

68



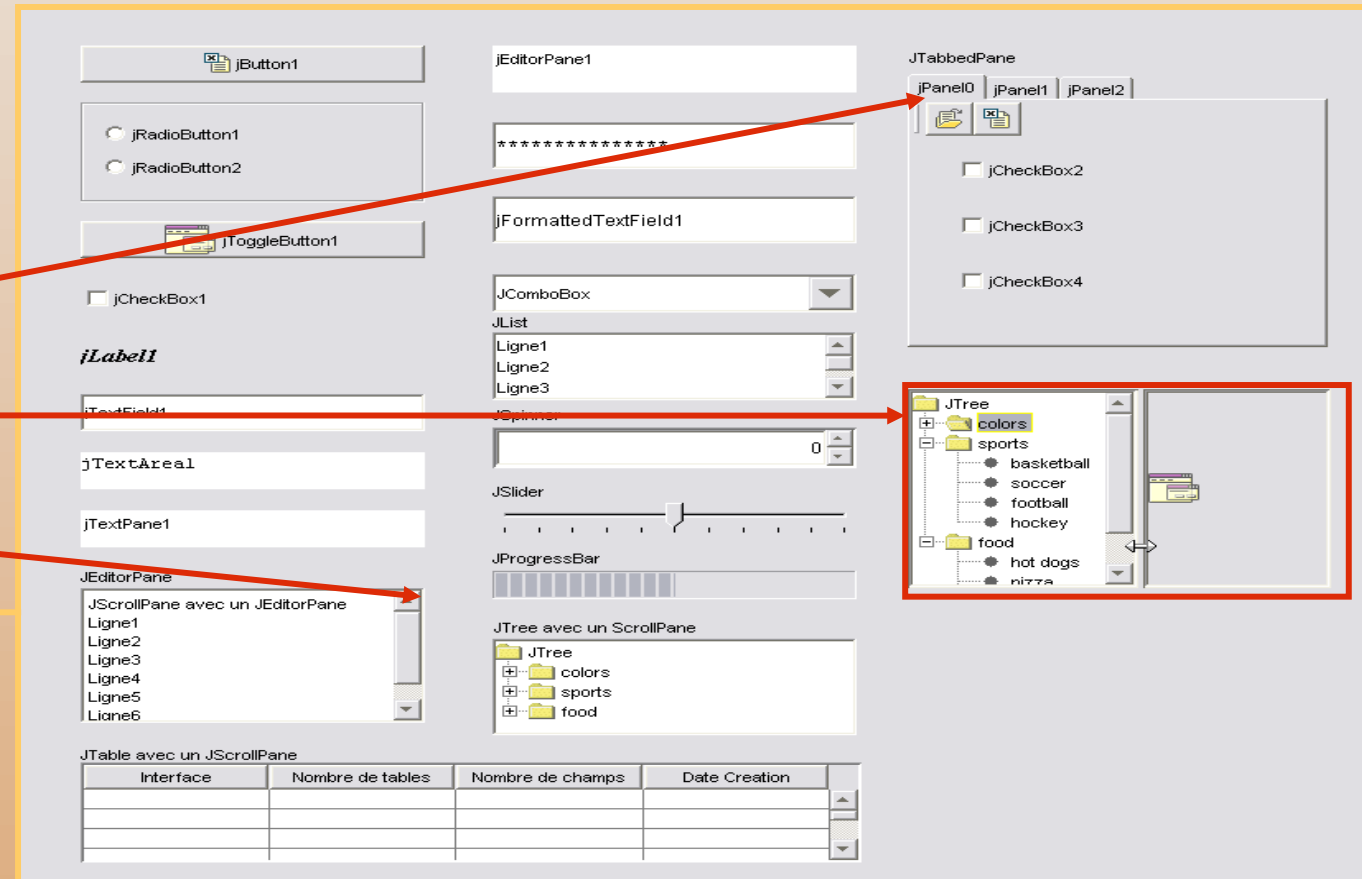
- *Les composants qui seront visibles dans la fenêtre seront placés dans un conteneur particulier associé à celle-ci : Content Pane*
 - *pour récupérer ce conteneur :*
`getContentPane() → Container`
- *La fenêtre peut contenir de manière optionnelle une barre de menus (qui n'est pas dans le content pane)*

Types de contener *SWING*

69

Containers

- *JOptionPane*
- *JDialog*
- *JTabbedPane*
- *JSplitPane*
- *JScrollPane*
- *JFrame*
- *JInternalFrame*
- *JDesktopPane*
- *JWindow*



De plus pres...

70

JPanel:

Aspect réduit au minimum :
rectangle invisible dont on peut
fixer la couleur de fond
Utilisé pour regrouper des
composants dans une fenêtre

JSplitPane:

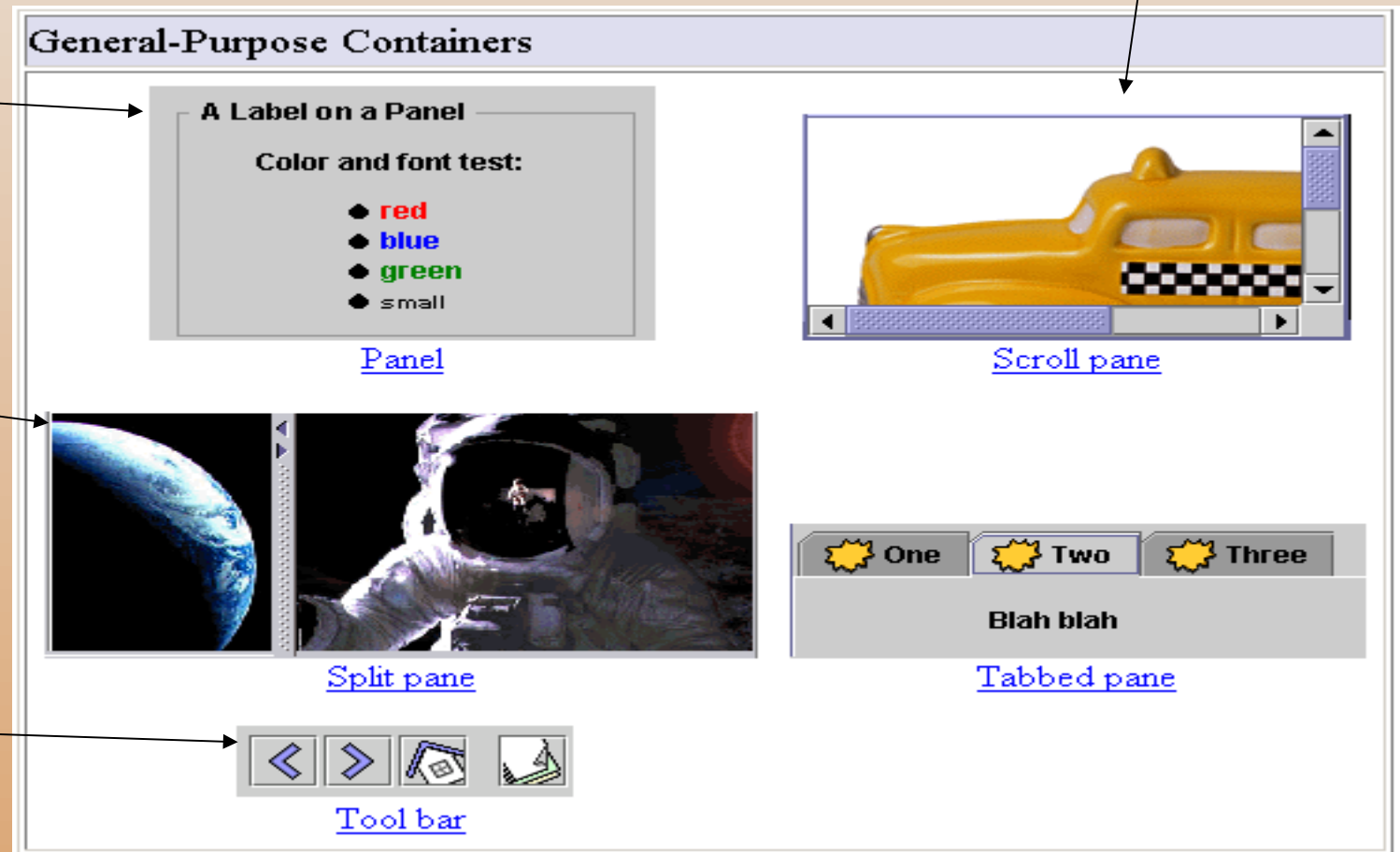
permet de séparer son
contenu en deux zones
distinctes dont les surfaces
respectives peuvent varier
dynamiquement

JToolBar:

barre d'outils (regroupe des boutons)

JScrollPane:

lorsque le composant
qu'il contient n'est pas
affichable dans sa totalité



Encore des conetner...

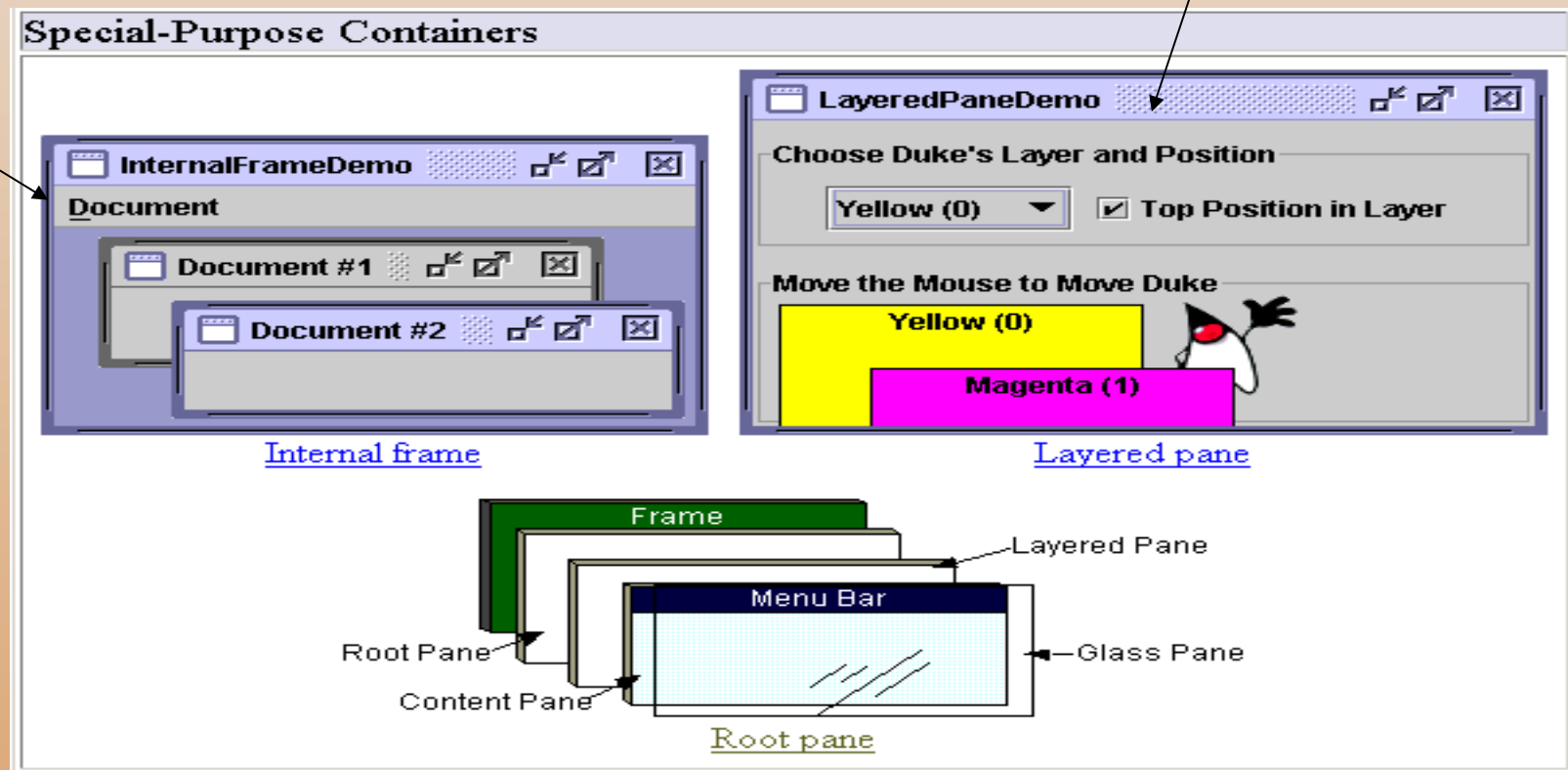
71

JDesktopPane:

permet de définir des
fenêtres internes
dans une fenêtre

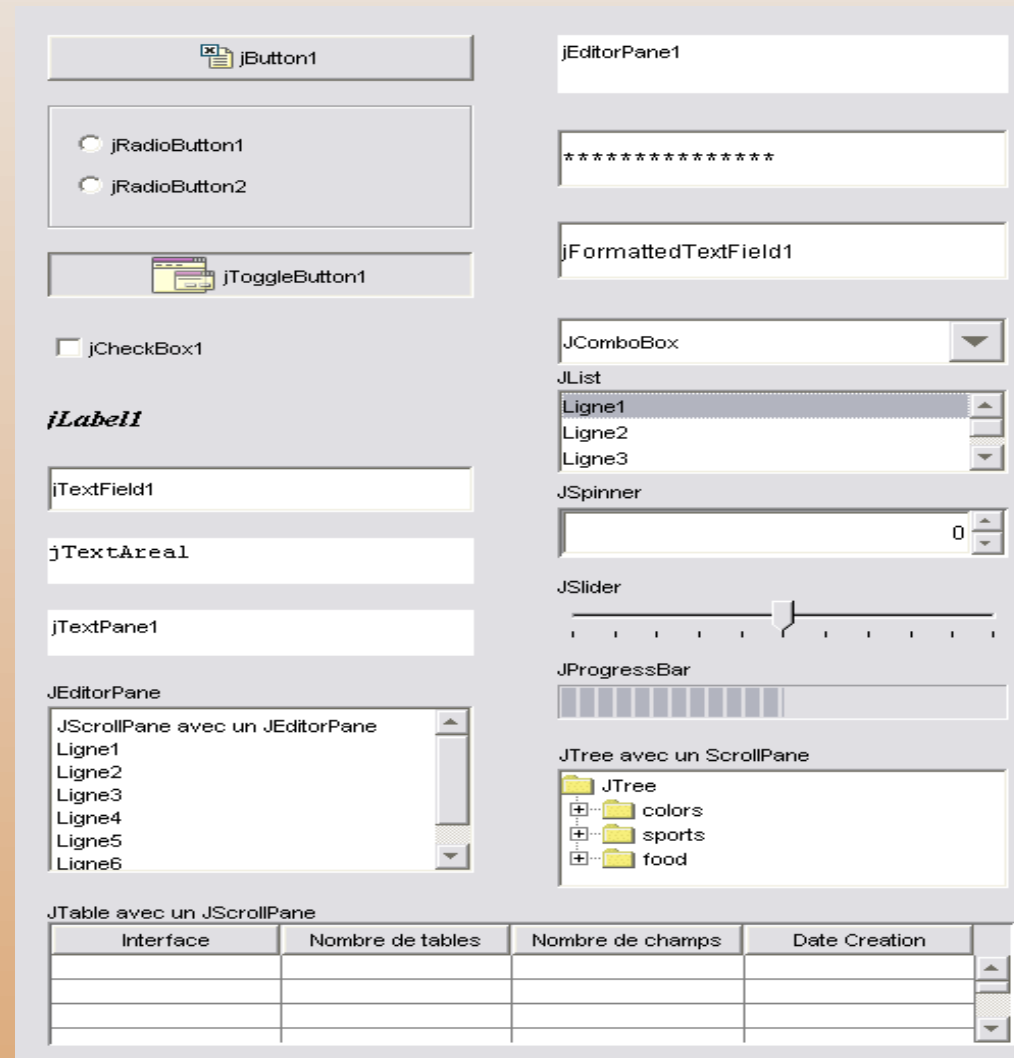
JLayeredPane:

fournit une troisième dimension
(z : profondeur) pour positionner
les composants qu'il contient



Les composants graphiques basiques

- *JApplet*
- *JButton*
- *JCheckBox*
- *JRadioButton*
- *JToggleButton*
- *JComboBox*
- *JList*
- *JSlider*
- *JTable*
- *JTree*
- *JProgressBar*
- *JSpinner*



Les composants de menu

□ Menus, Bar d'outils et ToolTips

■ *JMenuBar*

■ *JMenu*

■ *JMenuItem*

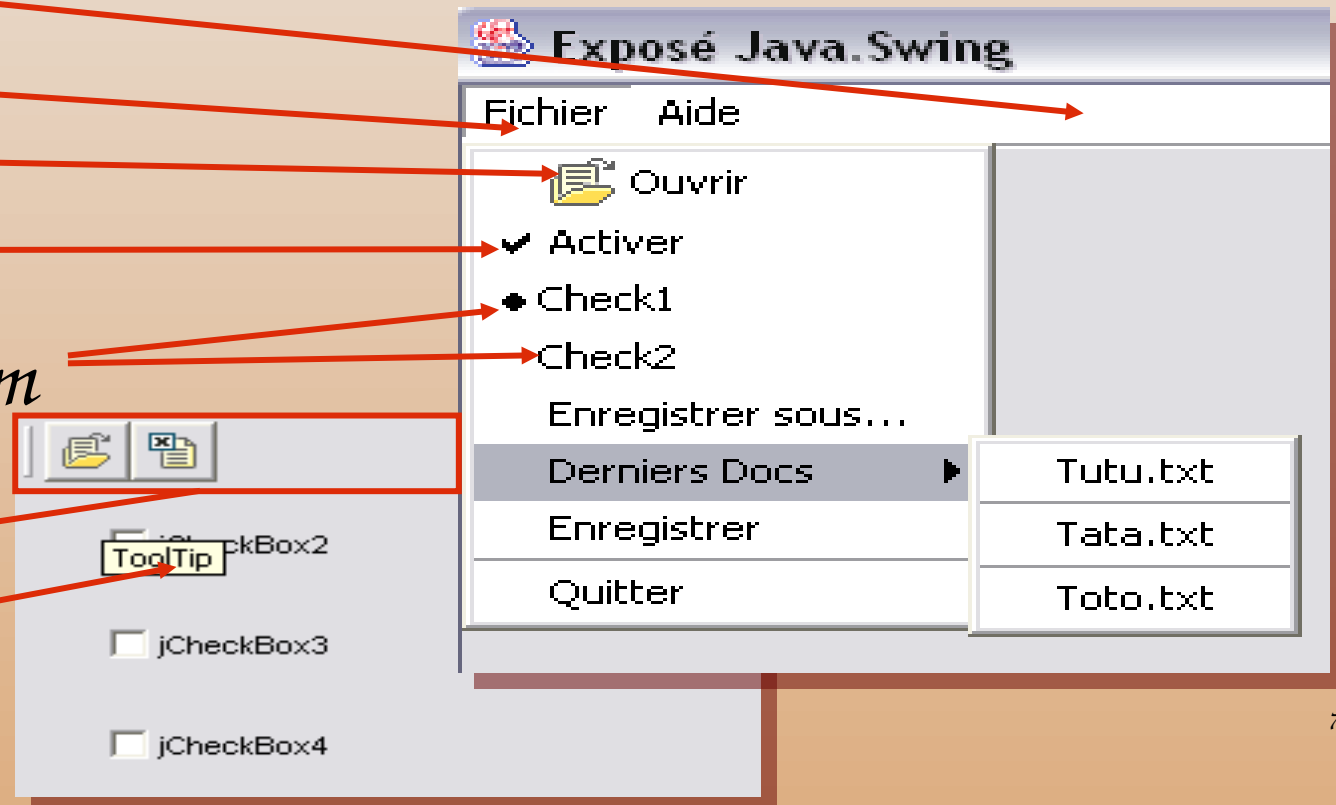
■ *JCheckBoxMenuItem*

■ *JRadioButtonMenuItem*

■ *JPopupMenu*

■ *JToolBar*

■ *JToolTip*



Les composants éditeurs de texte

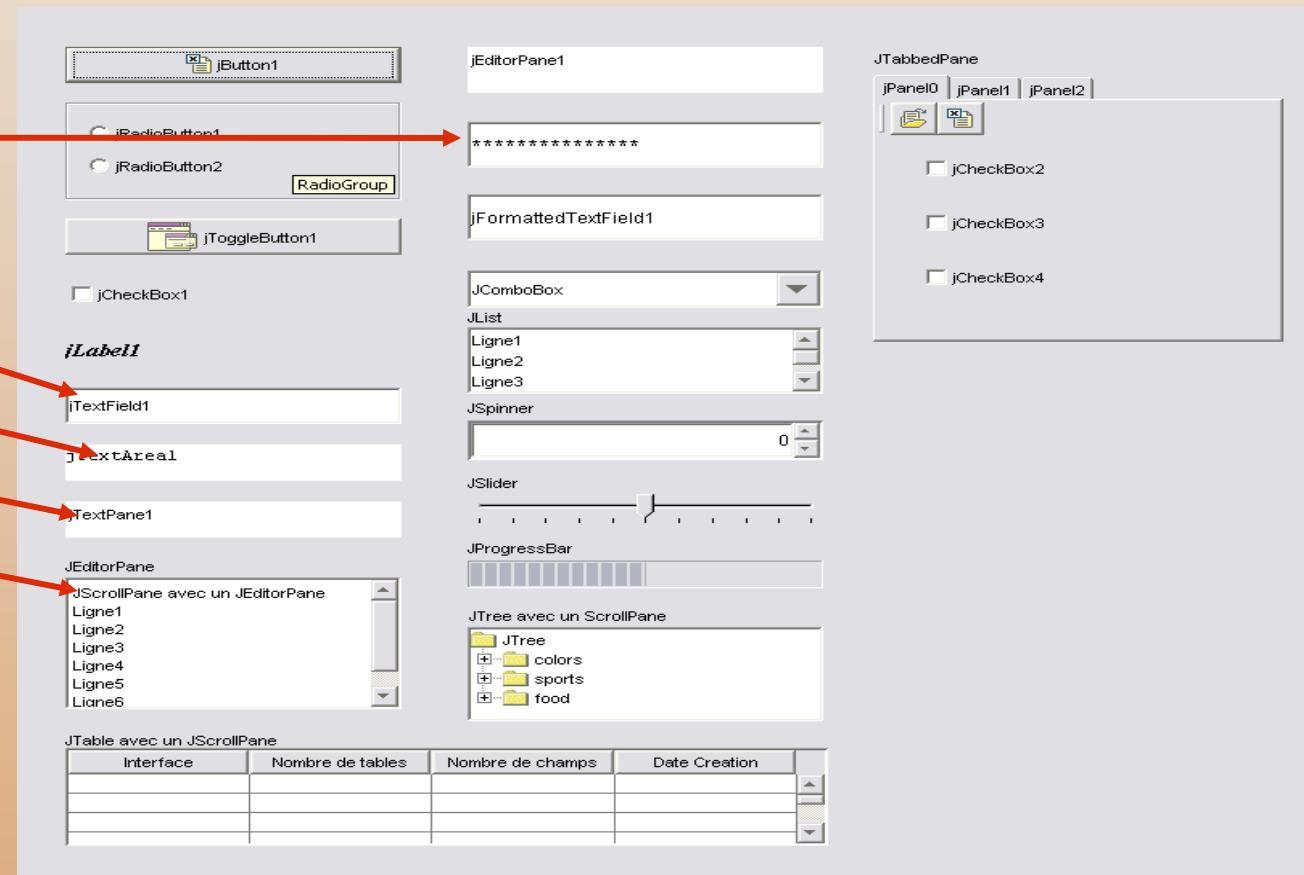
■ *JPasswordField*

■ *TextField*

■ *TextArea*

■ *TextPane*

■ *EditorPane*



Types d'éditeurs de Texte

75

JTextComponent



Légende

Texte Simple Ligne

Texte Multi Ligne

Texte Ordinaire

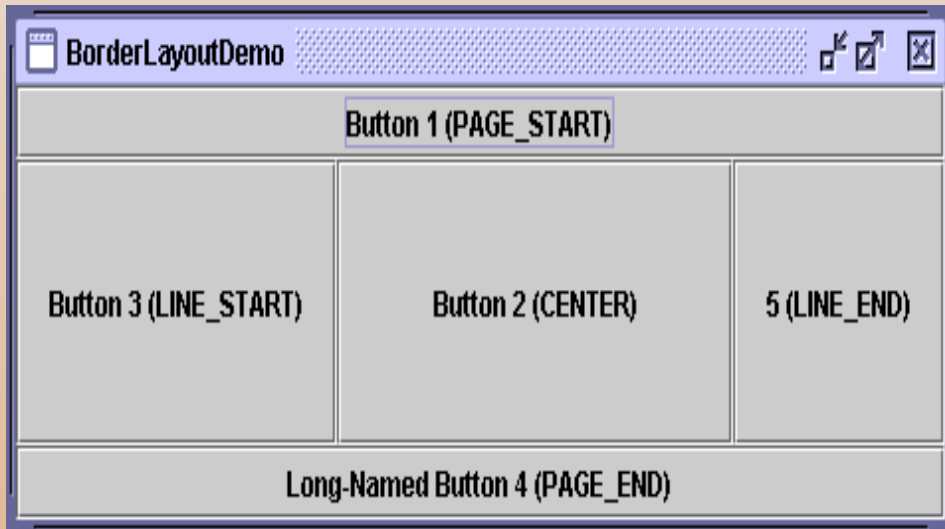
Texte Stylisé

Layout et les Containers

*Pour positionner un composant, nous avons plusieurs positions prédéfinis.
Ces positions qui sont proposés par Java sont:*

- ☐ *BorderLayout*
- ☐ *BoxLayout*
- ☐ *CardLayout*
- ☐ *FlowLayout*
- ☐ *GridBagLayout*
- ☐ *GridLayout*

BorderLayout



```
...//Container pane = uneFrame.getContentPane()...  
JButton button = new JButton("Button 1 (PAGE_START)");  
pane.add(button, BorderLayout.PAGE_START);  
  
button = new JButton("Button 2 (CENTER)");  
pane.add(button, BorderLayout.CENTER);  
  
button = new JButton("Button 3 (LINE_START)");  
pane.add(button, BorderLayout.LINE_START);  
  
button = new JButton("Long-Named Button 4 (PAGE_END)");  
pane.add(button, BorderLayout.PAGE_END);  
  
button = new JButton("5 (LINE_END)");  
pane.add(button, BorderLayout.LINE_END)
```

FlowLayout



//FlowLayout est la disposition par défaut d'un conteneur
contentPane.setLayout(new FlowLayout());

```
contentPane.add(new JButton("Button 1"));  
contentPane.add(new JButton("Button 2"));  
contentPane.add(new JButton("Button 3"));  
contentPane.add(new JButton("Long-Named Button 4"));  
contentPane.add(new JButton("5"));
```

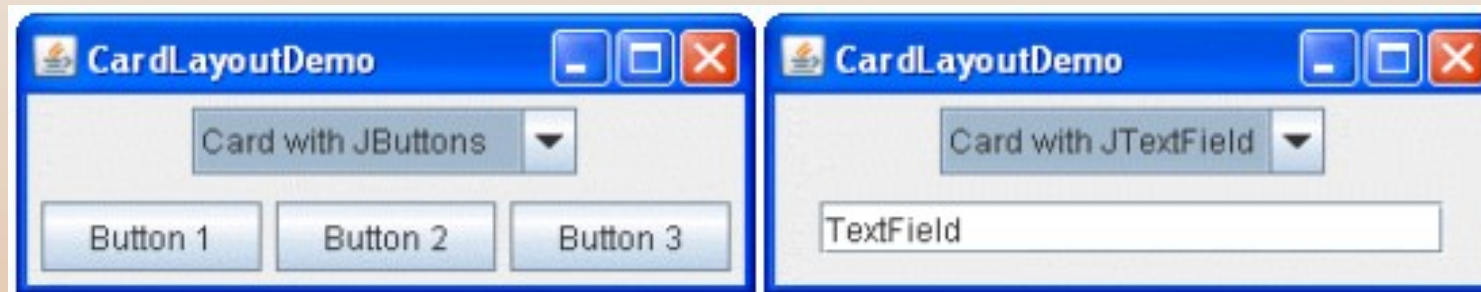
GridLayout



```
pane.setLayout(new GridLayout(0,2));  
  
pane.add(new JButton("Button 1"));  
pane.add(new JButton("Button 2"));  
pane.add(new JButton("Button 3"));  
pane.add(new JButton("Long-Named Button 4"));  
pane.add(new JButton("5"));
```

CardLayout

Gestionnaire de contenu graphique de type « tab » qui fait apparaître et cache les panel comme des slides



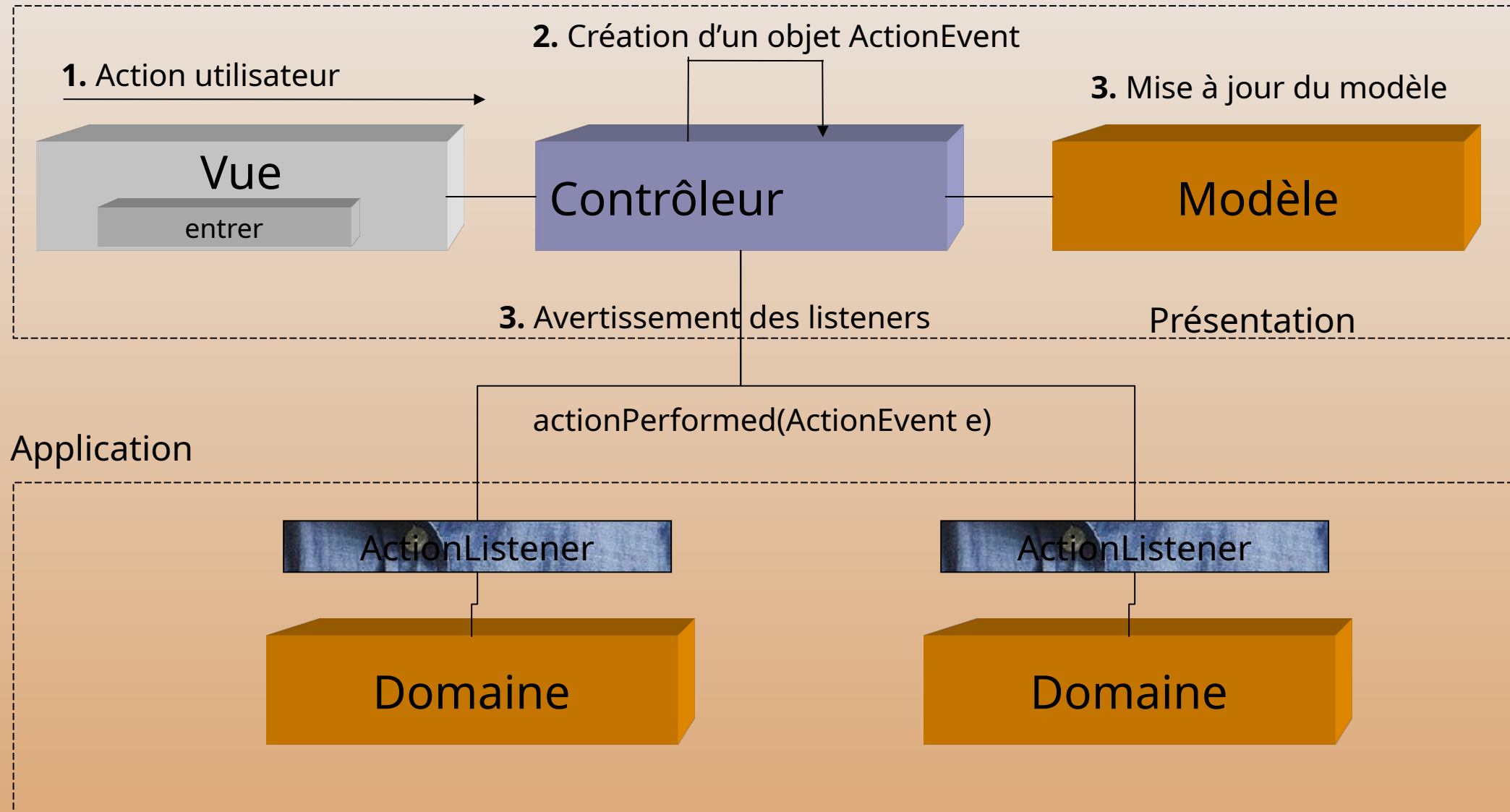
```
//Create the "cards".
JPanel card1 = new JPanel();
...
JPanel card2 = new JPanel();
...
//Create the panel that contains the "cards".
cards = new JPanel(new CardLayout());
cards.add(card1, BUTTONPANEL);
cards.add(card2, TEXTPANEL);
...

cards.show(cards, BUTTONPANEL) ;
```


- Quand ?
 - *Chaque fois qu'un utilisateur presse une touche de clavier, clique sur un bouton ou bouge sa souris, un événement est émis.*
- Où ?
 - *Le contrôleur interne du composant graphique récupère l'événement et crée un objet **ActionEvent**. Il transmet cet objet aux écouteurs associés à l'objet qui a émis l'événement.*
- Comment ?
 - *Un écouteur est une classe qui implémente une interface particulière et reçoit un événement. À partir du type d'événement, elle effectue une action particulière, puis répercute éventuellement des modifications sur le composant graphique.*

Gestion des évènements

Composant graphique (ex: JButton)



Exemples d'évènements

□ Définis dans java.awt.event

- *FocusEvent* : activation ou désactivation du focus du clavier
- *MouseEvent* : mouvement et clics de souris, et entrer/sortir d'un composant
- *KeyEvent* : événements clavier
- *WindowEvent* : dés/activation, ouverture, fermeture, dés/iconification de fenêtres
- *ComponentEvent* : changement de taille, position ou visibilité d'un composant

Les écouteurs

- *WindowListener* : pour les événements de la fenêtre
- *MouseListener* : pour les clics et entrées/sorties fenêtre
- *MouseMotionListener* : pour les mouvements de la souris
- *KeyListener* : pour les touches clavier
- *FocusListener* : pour le focus clavier
- *ComponentListener* : pour la configuration du composant

□ **WINDOWSLISTENER** Concerne tout ce qui est en rapport avec la fenêtre :

■ *Ouvrir, fermer, réduire, agrandir,*

windowActivated(WindowEvent e) - clic ***windowDeactivated(WindowEvent e)***

windowClosed(WindowEvent e) ***windowOpened(WindowEvent e)*** 1ere fois

windowClosing(WindowEvent e) menu fermé du système

windowDeiconified(WindowEvent e) ***windowIconified(WindowEvent e)***

□ *MouseListener : Gestion des états de la souris :*

- *Clic, Pressé, Relâché*
- *Et également l'entrée/sortie sur un composant*

mouseClicked(MouseEvent e)

mouseEntered(MouseEvent e)

mouseExited(MouseEvent e)

mousePressed(MouseEvent e)

mouseReleased(MouseEvent e)

Gestion de mouvements de la Souris

MouseMotionListener :

Mouvements de la souris sur un composant avec bouton appuyé ou relâché

- ***mouseDragged(MouseEvent e)***
- ***mouseMoved(MouseEvent e)***

□ KEYLISTENER

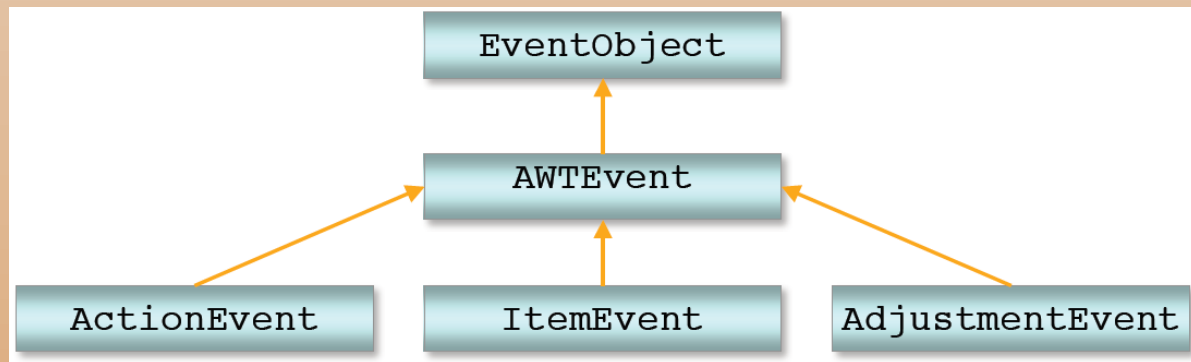
- *concerne tout ce qui est en rapport avec le clavier : tapé, pressé, relâché...*
 - *keyPressed(KeyEvent e)*
 - *keyReleased(KeyEvent e)*
 - *keyTyped(KeyEvent e)*

□ FOCUSLISTENER

- *gère le focus - savoir si un composant a obtenu le focus ou s'il la perdu*
 - *focusGained(FocusEvent e)*
 - *focusLost(FocusEvent e)*

Les événements sémantique

- 2eme catégorie d'évènement de JAVA à opposer au évènements dits « physiques »
- Sont issus des évènements physiques (clic sur un bouton, presse d'une touche de clavier, ...)
- Permettent de rassembler les comportements derrière des « series » d'évènement physique
 - Exemple : on mettra le même gestionnaire d'évènement sur l'entrée dans un `JtextField`, la sortie et la saisie d'un caractère au sein du `TextField`
- 3 Types concrets :
 - `ActionEvent`
 - `ItemEvent`
 - `AdjustmentEvent`



Evenements ActionEvent

- une action sur un composant réactif :
 - *Clic sur un item de menu,*
 - *Clic sur un bouton,*
 - *...*

- émis par les objets de type :
 - *Boutons : JButton, JToggleButton, JCheckBox*
 - *Menus : JMenu, JMenuItem, JCheckBoxMenuItem,*
JRadioButtonMenuItem ...
 - *Texte : JTextField*

Evènement ItemEvent

- composant sélectionné ou désélectionné
- émis par les objets de type :
 - *Boutons : JButton, JToggleButton, JCheckBox*
 - *Menus : JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem*
 - *Mais aussi JComboBox, JList*

Evènement AjustementEvent

- *Se produisent quand un élément ajustable comme une JScrollBar sont ajustés*
- *Sont émis par ScrollBar, JScrollBar*

COMPONENTLISTENER

permet de gérer: L'apparition/disparition d'un composant

- ❑ *componentHidden(ComponentEvent e)*
- ❑ *componentShown(ComponentEvent e)*

■ *Le déplacement d'un composant*

- ❑ *componentMoved(ComponentEvent e)*

■ *Le redimensionnement d'un composant*

- ❑ *componentResized(ComponentEvent e)*

Comment ajouter un écouteur d'événement ?

94

- Pour ajouter un écouteur, on utilise la méthode `addXXXListener (XXXListener l)` sur le **composant désiré**
- Il suffit alors de remplacer les **XXX** parce que l'on souhaite avoir
 - Exemple : pour un écouteur de souris on va avoir **addMouseListener**

Solutions pour installer un écouteur

Implémenter une interface

Inconvénient : doit implémenter toutes les méthodes

```
public class MaClass implement MouseListener {  
    ...  
    unObject.addMouseListener(this);  
    ...  
    void mouseClicked(MouseEvent e) {}  
    void mouseEntered(MouseEvent e) {}  
    void mouseExited(MouseEvent e) {}  
    void mousePressed(MouseEvent e) {}  
    void mouseReleased(MouseEvent e) {}  
}
```

Étendre une classe abstraite

```
public class MaClass extends MouseAdapter {  
    ...  
    unObject.addMouseListener(this);  
    ...  
    public void mouseClicked(MouseEvent e) {  
        ...  
        // l'implémentation de la méthode  
        // associée à l'événement vient ici ...  
    }  
}
```

Définition d'une classe anonyme

C'est une classe qui est déclarée
« à la volée » c'est à dire au
moment de son instantiation

```
button = new JButton("test");  
button.addMouseListener(new MouseAdapter() {  
    @Override  
    public void mouseClicked (MouseEvent e) {  
        // code que l'on souhaite effectuer  
    }  
});
```