Cours « Programmation orientée objet avancée »

Tarak Chaari

Maître assistant à l'Ecole Nationale d'Electronique et de Télécommunications de Sfax

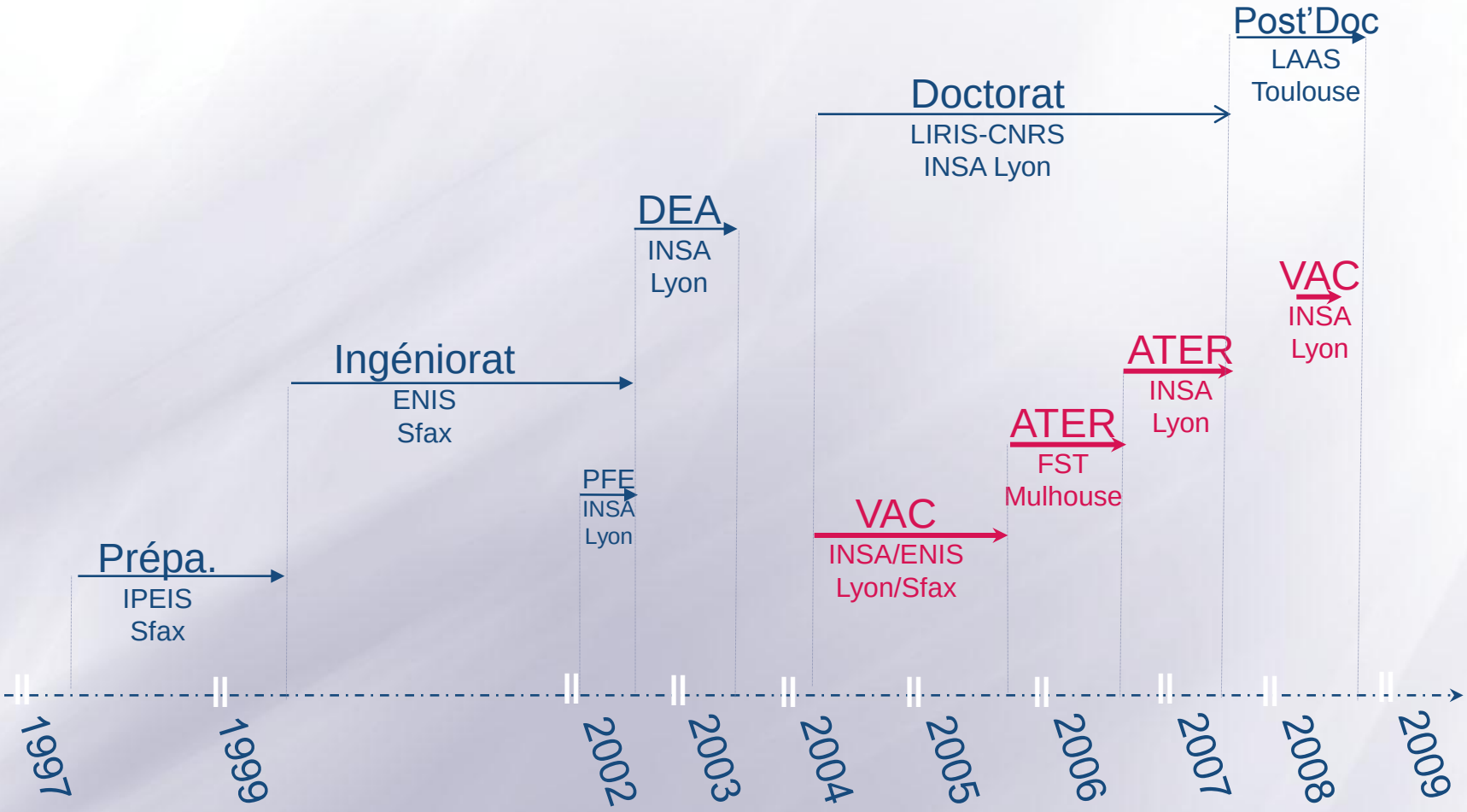
tarak.chaari@isecs.rnu.tn

http://www.redcad.org/members/tarak.chaari/cours/CoursPOO2.pdf

Votre interlocuteur

- Tarak CHAARI
- Maître assistant à l'ENET'com
- Membre de l'unité de recherche RedCad
- Enseignement: Ingénierie des systèmes d'information
- Recherche: l'adaptation dans les environnements dynamiques

Cursus universitaire



Présentation générale du cours

Le nom du cours

- Programmation orientée objet avancée

Volume horaire

- 21 heures
- Cours

Objectifs

- Consolider et compléter les connaissances déjà acquises en langage java
- Manipuler des techniques avancées de programmation Java (structure de données, swing, jdbc…)

Contenu du cours

- **Chapitre 1**
  - rappel sur la résolution de noms de classes
- **Chapitre 2 & 3**
  - Héritage et interfaces
- **Chapitre 4**
  - Exceptions
- **Chapitre 5**
  - Les structures de données en Java
- **Chapitre 6**
  - Interfaces graphiques
- **Chapitre 7**
  - JDBC

Rappel sur la résolution de noms des classes

- Packages
- Classpath
- Importation

Les packages - définition

- Les classes prédéfinies de java sont organisées en packages
  - java.util (Vector, Date,..), java.net (prog réseau), etc…
  - Deux classes ayant le même nom complet ne peuvent pas appartenir au même package
- Un package est un répertoire d'organisation des classes
  - Déclaration dans Watch.java: `package time.clock;`
  - Nom logique de cette classe : `time.clock.Watch`
  - Nom physique d'une classe : `time/clock/Watch.class`

Nom de classe : résolution

Pour résoudre un nom de classe dans une autre classe

1

```
...
time.clock.Watch toto=new time.clock.Watch();
...
```

2

```
import time.clock.Watch;
...
Watch toto=new Watch();
...
```

3

```
import time.clock.*;
...
Watch toto=new Watch();
Clock titi=new Clock();
...
```

4

```
import time.*;
...
Watch toto=new Watch();
Clock titi=new Clock();
...
```

Nom de classe : résolution

- Pour résoudre le nom d'une classe, soit :
  - On donne son nom complet lors de l'utilisation
  - On résout initialement son nom
  - On résout initialement tous les noms d'un package
- Les noms des classes du package java.lang n'ont pas à être résolus (importés par défaut lors de l'exécution)
- On ne peut pas importer 2 classes qui ont le même nom

```
Import java.util.Date
Import java.sql.Date
```

Les packages : Exercice Rapide !

graph/2d/Circle.java

```
package graph.2d;
public class Circle()
{ private double rayon;
  public Circle(double r)
  {   this.rayon=r;
      System.out.println("je suis un nouveau cercle de rayon " +this.rayon);
   }
}
```

graph/3d/Sphere.java

```
package graph.3d;
public class Sphere()
{ private double rayon;
  public Sphere(double r){
    this.rayon=r;
    System.out.println("je suis une nouvelle sphere de rayon " +this.rayon);
}
 }
```

paintShop/MainClass.java

```
package paintShop;
public class MainClass()
{
  public static void main(String[] args) {
    graph.2d.Circle c1 = new graph.2d.Circle(50)
    graph.2d.Circle c2 = new graph.2d.Circle(70);
    graph.3d.Sphere s1 = new graph.3d.Sphere(100);
    Sphere s2 = new Sphere(40); // error: class paintShop.Sphere not found
                                                    }
```

Les packages : utilisation des «alias»

graph/2d/Circle.java

```
package graph.2d;
public class Circle()
{ ... }
```

graph/3d/Sphere.java

```
package graph.3d;
public class Sphere()
{ ... }
```

paintShop/MainClass.java

```
package paintShop;

import graph.2d.Circle;

public class MainClass()
{
  public static void main(String[] args) {
    Circle c1 = new Circle(50)
    Circle c2 = new Circle(70);
    graph.3d.Sphere s1 = new graph.3d.Sphere(100);
    Sphere s2 = new Sphere(40); // error: class paintShop.Sphere not found
}
```

Crée un alias "Circle" pour graph.2d.Circle

Chargement de classe

```
Hello t;        /* La classe n'est pas encore chargée*/
t=new Hello();  /* Le classloader charge la classe
                   Hello en mémoire */
Vector v=new Vector(); /* Le classloader charge la
                          classe Vector en mémoire*/
```

- Le compilateur et l'interpréteur cherchent toutes les classes
- Les classes sont recherchées sur le système de fichiers
- Les classes standards sont automatiquement trouvées
- Pour indiquer l'endroit sur le système de fichiers ***à partir duquel*** il faut chercher les classes, on utilise le **classpath**
- Le classpath a un fonctionnement identique au path d'exécution

Le classpath

- Il indique à partir duquel endroit rechercher une classe

```
Javac -classpath /usr/local /tmp/paintshop/MainClass.java
java -classpath /usr/local:/tmp paintshop.MainClass
```

  - La classe `paintshop.MainClass` se trouve sous `/tmp`
  - La classe `paintshop.MainClass` utilise des classes qui se trouvent dans le répertoire `/usr/local`

- Il faut donc que les classes correspondent à ces fichiers:

```
/tmp/paintshop/MainClass.class
/usr/local/graph/2d/Circle.class
/usr/local/graph/3d/Sphere.class
```

Le classpath : le jar

- Un jar est une archive java
- Regroupement de fichiers dans un fichier zip
- La machine virtuelle java peut trouver les classes dans le système de fichiers (fichiers standards) ou bien dans des jar

Contenu de toto.jar

```
tutu/ours/Grumly.class
```

Contenu de Test.java

```
package test;
import tutu.ours.Grumly;
public class Test {
  Grumly toto=new Grumly();
}
```

```
javac ? ???
```

import / Classpath / Path

- import : alias du nom court sur un nom long
  - import java.util.Vector, permet d'utiliser l'alias Vector
  - import est utilisé par les classes
- Classpath : localisation physique des classes sur le disque
  - Chemin pour trouver la racine des classes externes (repertoire ou .jar)
  - Il est utilisé par la machine virtuelle
- Path : localisation physique des exécutables
  - Chemin pour trouver la racine des executables
  - Il est utilisé par le système d'exploitation
- Il n'y a aucun rapport entre import et classpath

Héritage et classes abstraites en JAVA

L'héritage - Objectifs

- Organiser les classes dans une hiérarchie de fonctionnement
- Les classes présentent dans ces relations d'héritage un rapport parent / fils
- La relation d'héritage représente une relation sémantique non standard entre le père et le fils
- Il n'existe pas de relation d'héritage universelle entre les classes. C'est le rôle de l'architecte d'application de définir la relation qu'il sous-entend

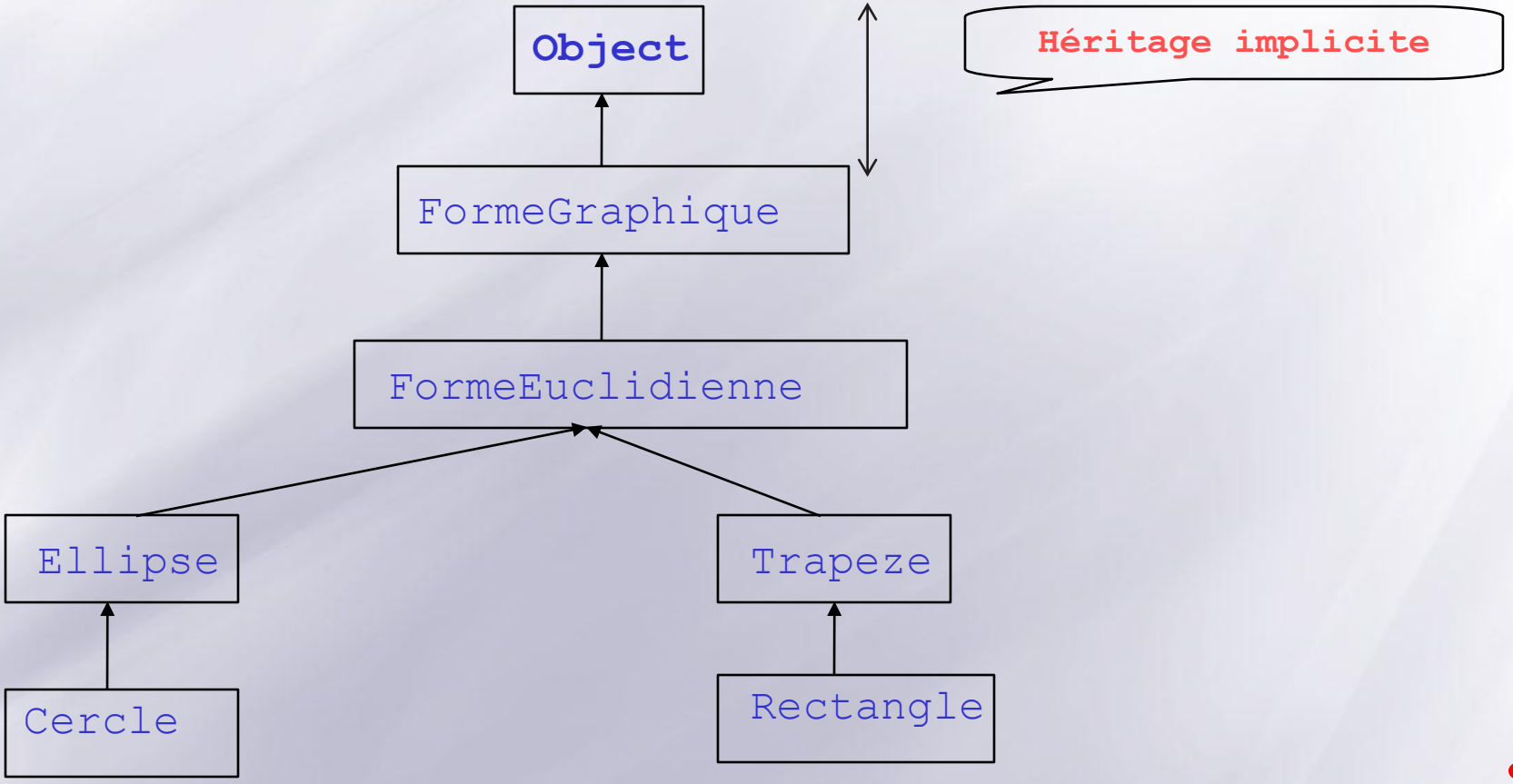
L'héritage - Syntaxe

```
public class Cercle extends FormeGeometrique {
                                    …
                                    }
```

- La classe parente présente généralement soit des attributs et méthodes généraux à toutes les classes filles, soit des attributs et méthodes types qui doivent être (re)définie dans dans les classes filles

L’héritage - Hiérarchie

- La relation d'héritage indique ce que l'objet est.



L’héritage - Sous-type

- Une sous-classe étend les capacités de sa super classe. Elle hérite des capacités de sa parente et y ajoute les siennes
- De plus, une sous-classe est une spécialisation de sa super-classe. Toute instance de la sous-classe est une instance de la super-classe (pas nécessairement l'inverse).

L'héritage – Sous-type

- En Java, une classe ne peut hériter (`extends`) que d'une seule classe
- Les classes dérivent, par défaut, de `java.lang.Object`
- l'héritage est transitif (« transmis ») (si B hérite de A et C hérite de B => C hérite de A via B)
- Une référence sur une classe C peut contenir des instances de C ou des classes dérivées de C.

L'héritage – Sous-type

- L'opérateur `instanceof` permet de déterminer la classe « réelle » d'une instance (renvoie un boolean)
- Les classes `final` (et respectivement les attributs et les méthodes) ne peuvent pas être redéfinies dans des sous-classes.
- Les classes `abstract` (et respectivement les méthodes) doivent être définies dans des sous-classes.
- `this` pour accéder aux membres de classe courante et super pour accéder aux membres de la super classe

Cast/Transtypage

```
1 Object [] tab=new Object[10];
2 tab[0]=new Circle();
3 System.out.println(tab[0]);
4 System.out.println((Circle)tab[0].getArea());
5 Circle c=(Circle)tab[0];
```

- UpCast est implicite (Ligne 2) pas besoin de faire

  tab[0]=(Object)new Circle();

- DownCast doit être explicite (Ligne 4), il faut transtyper tab[0] en Circle

L'héritage - Exemple

```
public abstract class FormeEuclidienne {
   public abstract double area();
}

public class Ellipse extends FormeEuclidienne {
   public double r1, r2;
   public Ellipse(double r1, double r2) { this.r1 =
   r1; this.r2 = r2; }
   public double area(){...}
}

final class Cercle extends Ellipse {
   public Cercle(double r) { super(r, r); }
   public double getRadius() { return r1; }
}
```

Exercices

- Ecrire le code de la classe rectangle
- Ecrire le code de la classe carré qui hérite de rectangle
- Ecrire un programme principal qui instancie des rectangles et des carrés et les stocke dans un même tableau

Les interfaces JAVA

Les interfaces - Objectifs

- une interface décrit ce que sait faire une classe (les méthodes de la classe)
- Spécification des comportements possibles pour une classe, en plus de son comportement de base
- Spécification formelle de classe. Elle indique les services rendus par la classe qui implante l'interface

Les interfaces - rôles

- Conceptuel : factoriser un comportement commun entre plusieurs classes différentes. Ce comportement est représenté par un ensemble de méthodes.
  - Exemple 1 : interface Dessinable qui contient la méthode dessiner (factorisée entre tous les objets dessinables (Cercle, Maison…)
  - Exemple 2 : interface Deplaçable qui contient la méthode déplacer (factorisée entre tous les objets qu'on peut déplacer (Voiture, Personne…)
- Technique : obliger le développeur à implémenter toutes les méthodes définies dans l'interface. Ainsi, il doit respecter la signature de ces méthodes → utile pour la programmation à plusieurs. Ex : affich, affiche(), afficher()…

Les interfaces - Syntaxe

```
interface Dessinable {

   public void draw();

            ...
            }
```

- L'interface définit l'ensemble des méthodes (par leur signature) devant être implémentées dans une classe réalisant cette interface

Les interfaces - Syntaxe

- Une interface correspond à une classe où toutes les méthodes sont abstraites
- Une classe peut implémenter (`implements`) une ou plusieurs interfaces tout en héritant (`extends`) d'une classe
- Une interface peut hériter (`extends`) de plusieurs interfaces

Les interfaces - Exemple

```
abstract class Forme { public abstract double perimeter(); }

interface Dessinable { public void dessiner(); }

interface Comparable{ public boolean egalA(Object o);
public boolean plusGrandQue(Object o);
public boolean plusPetitQue(Object o); }


class Cercle extends Forme implements Dessinable, Comparable{
   public double perimeter() { return 2 * Math.PI * r; }
   public void dessiner() {...}
   public boolean egalA(Object o) {...}
   public boolean plusGrandQue(Object o); {...}
   public boolean plusPetitQue(Object o); {...}

   }

class Rectangle extends Forme implements Dessinable, Comparable {
   public double perimeter() { return 2 * (height + width); }
   public void dessiner() {...}
   public boolean egalA(Object o) {...}
   public boolean plusGrandQue(Object o); {...}
   public boolean plusPetitQue(Object o); {...}

   }
```

Exercice

- Que pensez vous du code suivant :

```
Dessinable d = new Rectangle();
d.dessiner();
```

  Que permet la manière de programmer de type: `Interface i = new Object()` ?

- Définissez une classe AfficheDessinable qui affiche le contenu d'un tableau dont chaque case et de type dessinable

Gestion des erreurs en JAVA : Les 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 :
  - Tenter une correction en capturant l'erreur
  - Passer explicitement le problème au code qui va utiliser le programme concerné

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 prédéfinie d'exception
- provoque la sortie d'un bloc d'instruction
- correspond à un type d'erreur
- contient des informations sur cette erreur

Terminologie

- Une exception est un objet 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 (c.à.d une erreur)

- Capturer l'exception consiste à essayer de la traiter

Nature des exceptions

- En Java, les exceptions sont des objets ayant 3 caractéristiques:
  - Un type d'exception (défini par le nom de la classe de l'objet exception)
  - Une chaîne de caractères (message de l'exception)
  - Un « instantané » de la pile d'exécution au moment de la création

- Les exceptions qu'elles soient prédéfinies ou bien programmées héritent toutes de la classe *Exception*

Quelques exceptions prédéfinies en Java

- 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 des exceptions: try / catch / finally

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

...
finally
  {
    ...
  }
```

Si une erreur se produit ici….

On tente de récupérer là.

ou bien ici

Autant de blocs **catch** que l'on veut.
Bloc **finally** facultatif.

Interception vs propagation

Si une méthode peut émettre une exception (ou appelle une autre méthode qui peut en émettre une) il faut :

- **soit intercepter et traiter l'exception**
- **soit propager l'exception (la méthode doit l'avoir déclarée en utilisant throws)**

Exemple d'interception

```
public int ajouter(int a, String str) {
    try {
        int b = Integer.parseInt(str);
        a = a + b;

    } catch (NumberFormatException e) {
        System.out.println(e.getMessage());
    }
    return a;
}
```

Exemple de propagation

```
public int ajouter(int a, String str) throws NumberFormatException
    int b = Integer.parseInt(str);
    a = a + b;
    return a;
}
```

Les objets `Exception`

- Un objet de type **Exception** admet un champ de type String
- Ce champ est utilisé pour stocker le message décrivant l'exception.
- Il est positionné en passant un argument au constructeur
- Ce message peut être récupéré par la méthode `getMessage()`
- Les développeurs peuvent définir de nouvelles exceptions par héritage de la classe Exception

Création et l'utilisation de nouvelles exceptions

1) Création de la classe de l'exception

Par héritage de la classe Exception

2) Levée de l'exception

Spécifier dans quel cas l'exception que nous avons créée doit être levée

3) Emission (propagation) de l'exception

Déclarer la possibilité d'émission de l'exception dans la méthode qui lève cette exception

4) Interception ou propagation de l'exception dans le code qui invoque la méthode qui l'émet

Intercepter par un *try/catch* ou propager l'exception par un *throws*

1) Exemple de création d'une nouvelle exception

```
public class MonException extends Exception
   {
   public MonException()
      {
         super(« mon message  d'erreur»);
      }
}
```

```
public class CalculImpossibleException extends Exception
{
public CalculImpossibleException (String s)
                {
                super (s);
                }
}
```

2) Levée d'exceptions

- Le programmeur peut lever ses propres exceptions à l'aide du mot réservé **throw**.

- **throw** prend en paramètre un objet instance de **Exception** ou d'une de ses sous-classes

- Les objets exception sont souvent instanciés dans l'instruction même qui assure leur lancement

```
throw new MonException("Mon exception s'est produite !!!");
```

3) Emission d'une exception (1/3)

- L'exception elle-même est levée par l'instruction **throw**.

- Une **méthode** susceptible de lever une exception est identifiée par le mot-clé **throws** suivi du type de l'exception

exemple :

```
public class ExempleException
        {
        public int propageException (int arg1 , int arg2) throws
                                                CalculImpossibleException
        {
        if (arg1!=arg2)       return arg1+arg2;
        else
        throw new CalculImpossibleException("impossible de faire le calcul");
        }
        }
```

Emission d'une exception (2/3)

- Pour "laisser remonter" à la méthode appelante une exception qu'il ne veut pas traiter, le programmeur rajoute le mot réservé **throws** à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

```
public void uneMethode() throws IOException
  {
    // ne traite pas l'exception IOException
    // mais est succeptible de la générer
  }
```

Emission d'une exception (3/3)

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever
- La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée
- Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des ,)
- Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans son code

Conclusion

- Grâce aux exceptions, Java possède un mécanisme sophistiqué de gestion des erreurs permettant d ’écrire du code « robuste »
- Le programme peut déclencher des exceptions au moment opportun
- Le programme peut capturer et traiter les exceptions grâce au bloc d’instruction `try` … `catch` … `finally`
- Le programmeur peut définir ses propres classes d ’exceptions

Exercice

- Définissez une classe DivisionParZeroException comme une sous classe de Exception
- Définissez une classe Division qui contient une méthode diviser(int a, int b)
- Levez une exception (instance de la classe DivisionParZeroException) quand le diviseur est 0
- Ecrivez le code de la classe TestDivision qui appelle la méthode diviser

Les structures de données en Java

Sommaire

Types de base

- Types primitifs
- Classes des types primitifs (wrappers)

Structures de collection

- Structure de base : Tableaux
- Interface qui représente tout type de collection : Collection
- structures les plus utilisées : Vector et Hashtable (implements Collection)

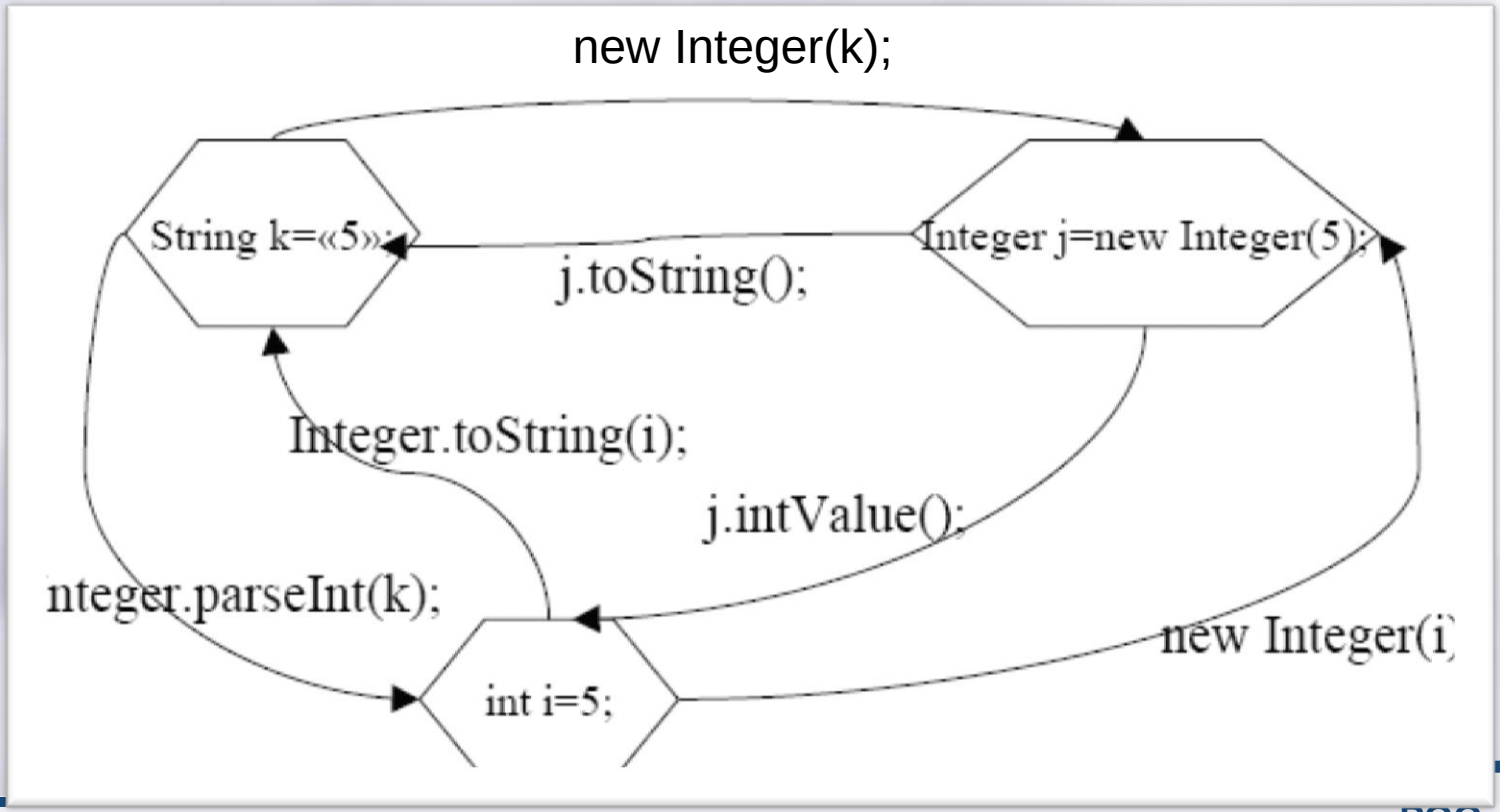
- Remarque: toute classe est une structure de données

Les types primitifs

- Types
  - byte : 1 octet
  - short : 2 octets
  - int : 4 octets
  - long : 8 octets
  - float : 4 octets
  - double : 8 octets
  - boolean : true/false
  - char: 2 octets (en unicode)
- Un type primitif ne prend jamais de majuscule

Classes des types primitifs (Wrappers)

- Classes qui encapsulent des types primitifs
- Exemples: String, Integer, Boolean, Long…
- Possibilité de conversion: Exemple



Structures de collection

- Collection (conteneur) = ensemble générique d'objets
  - main de bridge (collection de cartes), répertoire de fichiers, répertoire téléphonique, …
- Les objets peuvent être soumis à des contraintes
  - Ordre (liste), entrées uniques (ensemble), …

Les tableaux

- Un tableau JAVA est une structure collectant un ensemble (dont on connaît sa cardinalité d'avance) d'objets de même type
- Déclaration d'un tableau

  type[] unTableau; (type peut être n'importe quel objet ou type de base)
- Création du tableau
  - unTableau = new type[10];
- Accès aux éléments
  - unTableau[7] = new type();// en mode écriture
  - System.out.println(unTableau[7]);// en mode lecture

Vector & Hashtable

- **Vector**
  - Une classe de collection qui engendre un tableau d'objets
  - Méthodes: add(Object o), get(int index), remove(int index), remove(Object o), size(), contains(Object o)
  - Gère une taille dynamique du tableau
- **Hashtable**
  - Une classe de collection qui gère un ensemble de couples (clé, valeur)
  - Méthodes: put(Object key, Object value), get(Object key), remove(Object key)
  - Recherche transparente par rapport à l'utilisateur

La Classe Vector (1/5)

- **Les objets de la classe Vector représentent des tableaux à taille variable, c.à.d. il n'y a pas de limite au nombre d'objets qu'il peut contenir :**
  - void addElement ():ajouter un élément dans le vecteur
  - Object elementAt(int index):
    - retourne (sans retirer) l'élément à la position index.
  - void insertElementAt(Object obj,int pos):
    - place l'objet obj à la position pos indiquée (remplacement)
  - boolean contains(Object obj):
    - retourne true si obj est dans le tableau.
  - int indexOf(Object obj):
    - retourne la première occurrence de obj (-1 si obj n'est pas présent)
  - int size() :
    - retourne le nombre d'éléments dans le vecteur

La Classe Vector (2/5)

- **Présentation :**
  - Classe située dans le package java.util
- **Exemple d'utilisation 1:**

```
import java.util.* ;
Vector v = new Vector() ;
v.addElement("1");
v.addElement("2");
v.addElement("3");
System.out.println(v.toString()) ;//affichage du Vector
```

Résultat

[1,2,3]

Exercice:

Ecrire le code de la méthode toString () de la classe Vector

Exemple d'utilisation 2:

- v.insertElementAt("hop",2) //insertion en 3eme position
- System.out.println(v.toString()) ; // affichage du Vector

Résultat

[1,2,hop,3]

v.setElementAt("truc",2) // changement d'un élément

System.out.printl(v.toString()) ; // affichage du Vector

Résultat

[1,2,truc,3]

La Classe Vector (4/5)

- **Parcours d'un Vector :**

```
for(int i = 0 ; i< v.size( ) ;i++)   // affichage du Vector
    System.out.println(v.elementAt(i)) ;
```

- **Autres méthodes utiles**

```
int num ;
num = v.indexOf("elem2") ;   // indice d'un élément
v.removeElement("elem2") ; // retrait d'un élément
v.removeElementAt(2) ;       // retirer et renvoyer le 2 ème élément
System.out.println(v.lastElement( )) ;
```

La Classe Vector (5/5)

Conversion en sortie de Vector :

- Un Vector peut contenir des Object (structure polymorphe)
- un élément retiré d'un Vector doit être converti au type désiré

```
import java.util.* ;
class Personne{
  ...
  Public static void main() {
    Personne b;
    Vector v = new Vector( ) ;
    v.addElement(new Personne(..));          // ajout d'une personne
    v.addElement(new Personne(..));          // ajout d'une personne
    v.addElement(new Personne(..));          // ajout d'une personne
    b = (Personne) v.elementAt(1) ;      // conversion en personne
  }
}
```

Exercice

- Ecrire le code d'une classe Pile1 qui représente une pile LIFO (Last In First Out). Cette classe doit offrir 2 méthodes public Object depiler() et public void empiler(Object o). Cette classe doit se baser sur un attribut de type vecteur qui permet d'encapsuler les éléments de la pile.
- Définir la classe Pile2 avec le même principe que Pile 1 mais avec héritage de la classe Vector

La Classe Hashtable(1/2)

Présentation :

- Tables contenant des doubles <clé,valeur>
- Clé et valeur sont de type Object
- Opérations possibles sur ces tables : ajout, retrait, recherche de la valeur associée à une clé donnée, test de présence de clé ou d'info, etc.

Exemples d'utilisation

```
import java.util.* ;
Hashtable t = new Hashtable() ; // création d'une Hashtable
Integer a, b, c ;                // création de 3 Integer
Integer n ;
a = new Integer(1) ; b = new Integer(2) ; c = new Integer(3) ;
t.put("un",a) ; t.put("deux",b) ; t.put("trois",c) ;
```

La Classe Hashtable(2/2)

- **Méthodes utiles :**

```
n = (integer) t.get("deux") ; // recherche de l'info associée à une clé
if(t.contains(c))..... // test présence d'une info
if(t.containsKey("deux") ).. // test présence d'une clé
t.remove("deux") ; // suppression d'une entrée
```

- **Exercice**

- Ecrire 2 version de la classe GestionEtudiant basées sur un vecteur et sur une hashtable.

- **Dans ces 2 versions il faut définir la méthode:**

public Etudiant rechercherEtudiant(String numInscription)

Exercice

1) Définir une interface Pile (LIFO).

Cette interface contient les méthodes :

```
public void push(Object item) ;
public Object pop ();
```

2) Définir une implémentation de l'interface Pile qui se base sur un vecteur

3) Ecrire une classe cliente de la Pile réalisée

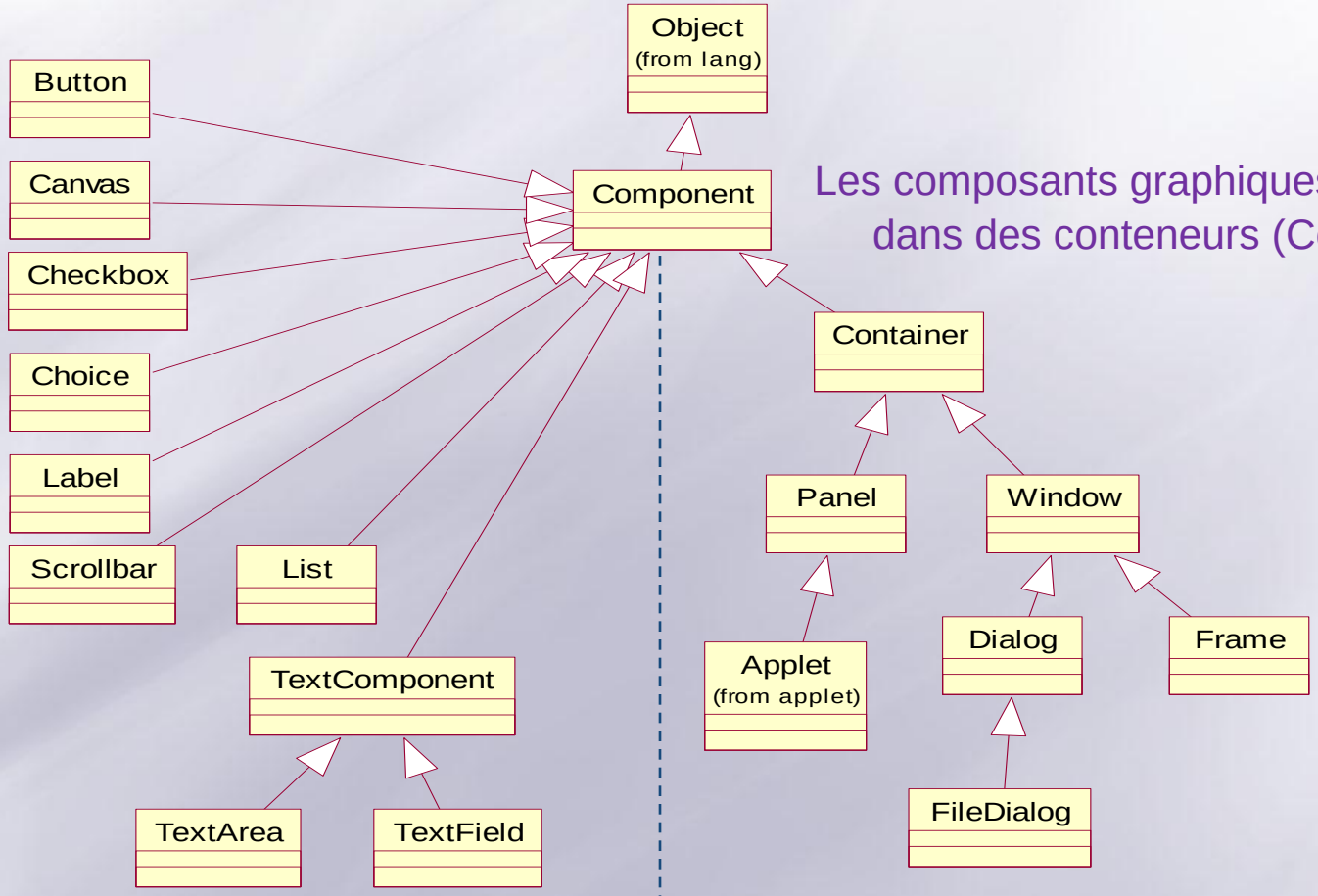
Interfaces graphiques en JAVA

AWT & SWING

- Première bibliothèque graphique JAVA: AWT
  - Package java.awt
  - Utilisation de code qui dépend du système d'exploitation
  - Composants limités (pas de tables)
- Nouvelle bibliothèque: SWING
  - Package javax.swing
  - Plus riche et plus personnalisable
  - Ne remplace pas AWT mais fournit des composants plus performants

Conteneurs et composants

Hiérarchie d'héritage des principaux éléments des interfaces graphiques en Java

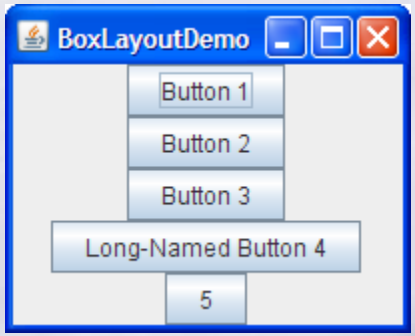


Les composants graphiques sont placés dans des conteneurs (Containers)

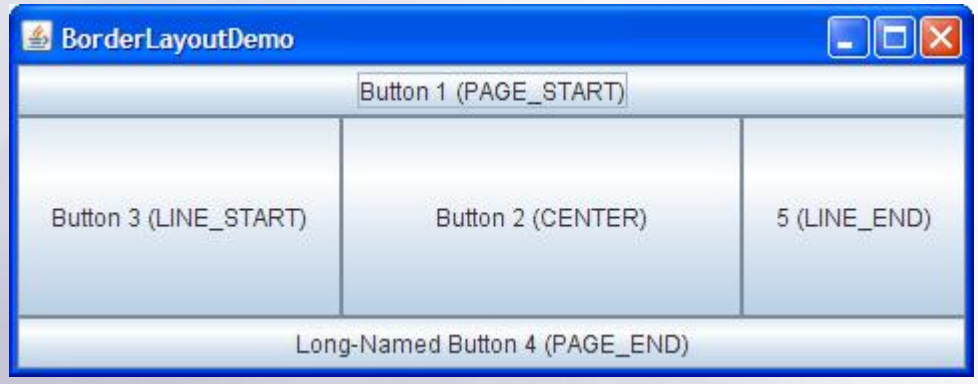
Disposition des composants (1/2)

❖ **Chaque conteneur utilise un gestionnaire de placement (Layout) pour la disposition des composants qu'il contient.**

BoxLayout

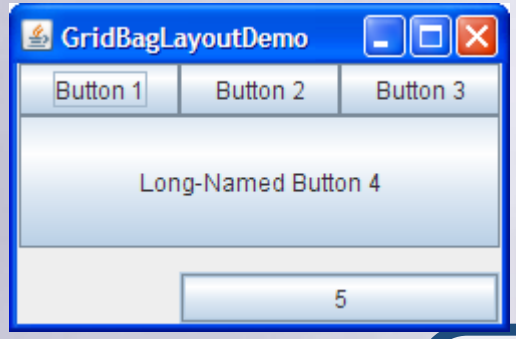


BorderLayout



GridLayout

GridBagLayout



Disposition des composants (2/2)

❖ **Exemples de dispositions**

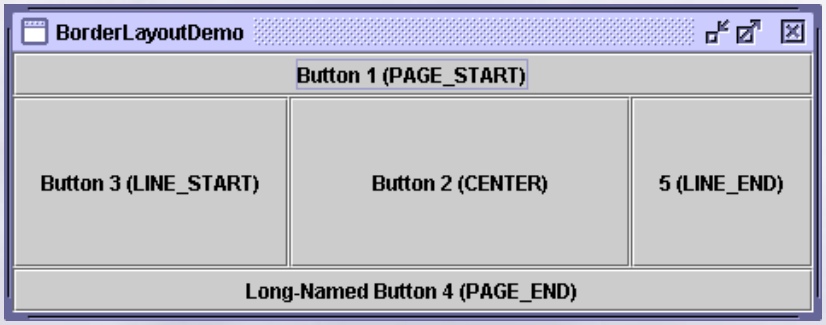
GridLayout


```
JFrame fenetre=new JFrame("GridLayoutDemo");
JPanel tmp = (JPanel)fenetre.getContentPane();//récupérer le panneau principal de la
fenêtre tmp.setLayout(new GridLayout(3,2));//appliquer les layouts
tmp.add(new JButton("Button 1"), 0,0);
tmp.add(new JButton("Button 2"), 0,1);
tmp.add(new JButton("Button 3"), 1,0);
tmp.add(new JButton("Long-Named Button 4 "), 1,1);
tmp.add(new JButton("5"), 2,0);
fenetre.show();
```

Disposition des composants (2/2)

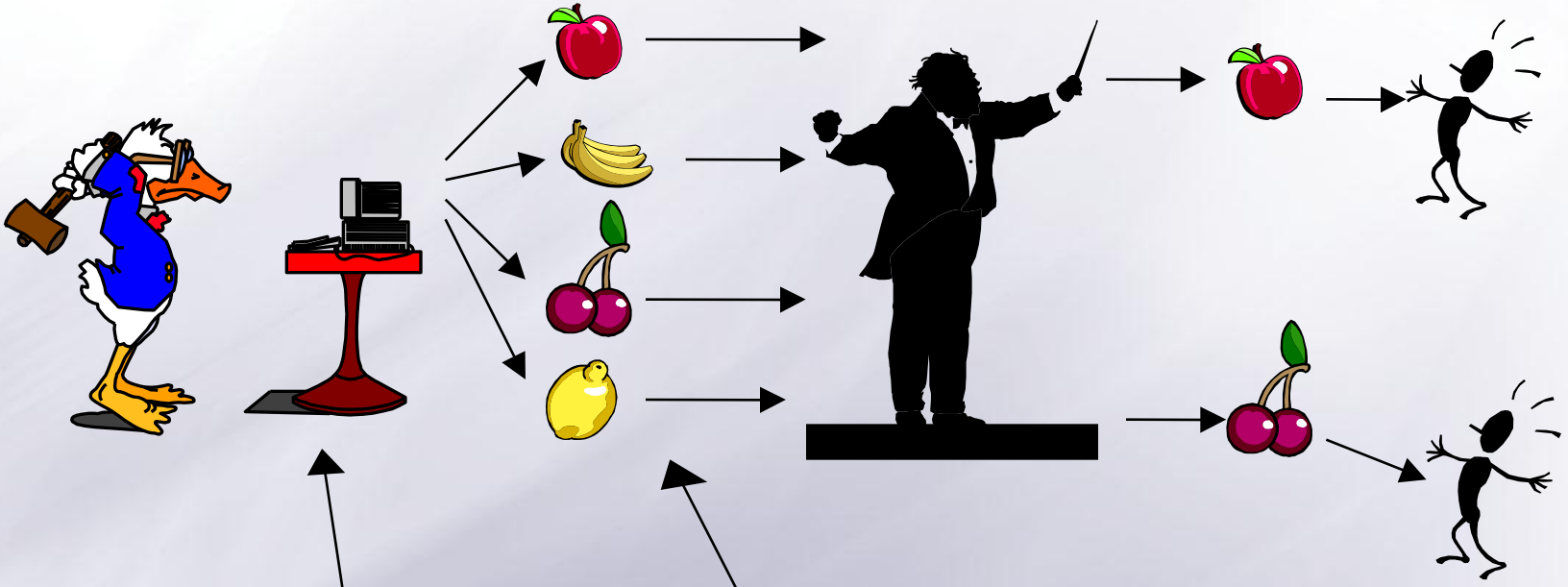
❖ **Exemples de dispositions**

BorderLayout




```
JFrame fenetre=new JFrame("BorderLayoutDemo");
JPanel tmp = (JPanel)fenetre.getContentPane();
tmp.setLayout(new BorderLayout());
tmp.add(new JButton("Button 1(PAGE_START)", BorderLayout.NORTH));
tmp.add(new JButton("Button 3 (LINE_START)", BorderLayout.WEST);
tmp.add(new JButton("Button 2 (CENTER)"), BorderLayout.CENTER);
tmp.add(new JButton("5 (LINE_END)"), BorderLayout.EAST);
tmp.add(new JButton("Long-Named Button 4 (PAGE_END)"),
BorderLayout.SOUTH);
fenetre.show();
```

Propagation des évènements



L ’utilisateur réalise une action

Source de l ’événement

JButton, JTextField…

Des événements sont alors générés

Événements

ActionEvent, MouseEvent…

La Machine Virtuelle reçoit tous les événements

Seuls les événements écoutés sont transmis

Écouteurs

ActionListener, MouseListener…

Les acteurs

- **Le Composant :**
  - Indique les événements qu'il peut générer.
  - Button : ActionEvent, MouseEvent,…
- **L'événement :**
  - Indique l'action que l'utilisateur a générée.
  - Ex : ActionEvent
- **Le listener :**
  - Il indique le traitement à faire sur une catégorie d'événements
  - MouseListener, ActionListener…

Les événements

- Tous les composants génèrent des événements
  - Car il dérivent de la classe Component qui génère des événements
- Tous les composants ne génèrent pas tous les même événements
  - Un bouton ne génère pas d'événements de type text
- Il existe pour les composants élémentaires un événement de sémantique générale appelé ActionEvent, qui représente l'interaction standard avec l'utilisateur
  - Click sur bouton ==> ActionEvent
  - DoubleClick sur une liste ==> ActionEvent
  - Click sur un élément de liste ==> ActionEvent
  - <Return> à la fin d'une saisie dans un TextField ==> ActionEvent

Les Listeners

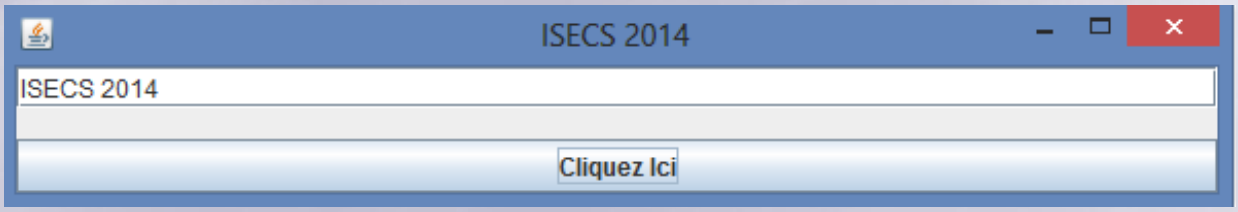
- Les événements qui intéressent le programmeur doivent être capturés dans des écouteurs
- Ces écouteurs sont des objets qui implémentent des interfaces prédéfinies (MouseListener, ActionListener)
- Par exemple, l'interface ActionListener contient la méthode:
  - public void actionPerformed(ActionEvent e)
  - Exemple d'écouteur

```
import java.awt.event.*;
class EcouteurSimple implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("j'ai capturé l'évènement "+e);
  }
}
```

- Ces écouteurs doivent être explicitement affectés aux composants concernés (boutons,..)
  - monBouton.addActionListener(new EcouteurSimple());

Exercice

- Réaliser une classe **ExerciceSwing** qui affiche une fenêtre (JFrame) contenant un Jbutton et une zone texte (JTextField) et un écouteur d'évènements (ActionListener) pour qu'a chaque click sur le bouton, le contenu de la zone texte soit affecté au titre de la fenêtre à l'aide de la méthode setTitle().



Correction de l'exercice Swing (1/2)

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

class ExerciceSwing implements ActionListener
{
        private JButton monBouton;
        private JTextField zoneTexte;
        private JFrame maFenetre;

        public ExerciceSwing()
        {
                maFenetre = new JFrame("Mon premier Exercice Swing");
                zoneTexte = new JTextField("texte par défaut");
                monBouton = new JButton("Cliquez Ici");
        }

        public void actionPerformed(ActionEvent e)
        {
                if (e.getSource()==monBouton)
                        maFenetre.setTitle(zoneTexte.getText());
        }
```

Correction de l'exercice Swing (2/2)

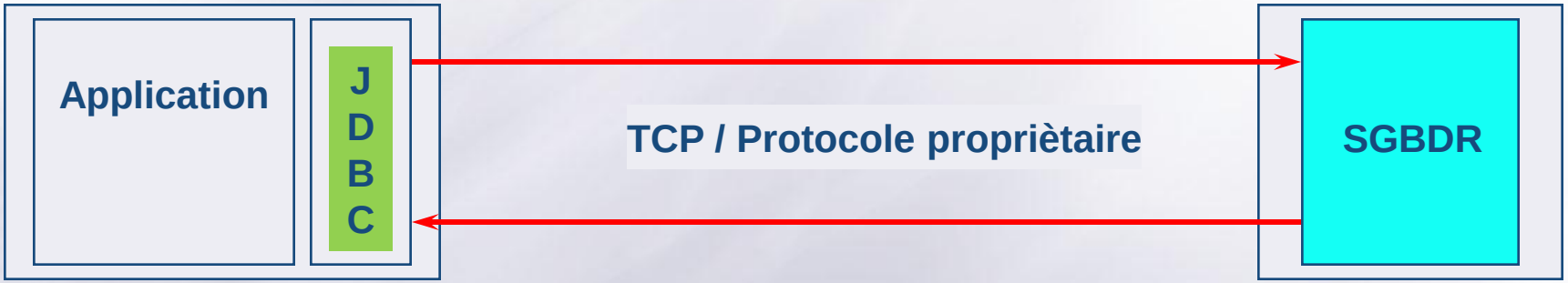
```
	public static void main(String[] arguments)
	{
		ExerciceSwing monAppli = new ExerciceSwing();
		monAppli.afficher();
	}

	public void afficher()
	{
		JPanel conteneurPrincipal = (JPanel) maFenetre.getContentPane();
		conteneurPrincipal.setLayout(new BorderLayout());
		maFenetre.setSize(600,100);
		conteneurPrincipal.add(zoneTexte, BorderLayout.NORTH);
		conteneurPrincipal.add(monBouton, BorderLayout.SOUTH);
		monBouton.addActionListener(this);
		maFenetre.show();
	}
}
```

Accès aux bases de données: JDBC

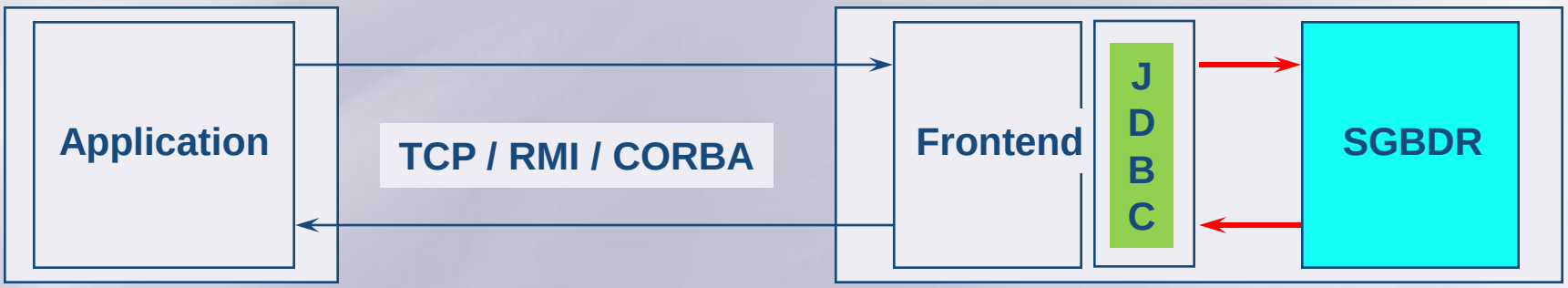
Java DataBase Connectivity : Objectifs

- Fournir un accès *homogène* aux SGBDR
  - Abstraction des SGBDR cibles
- Requêtes SQL
- Simple à mettre en oeuvre
- Inclut dans java depuis la version JDK(1.1)
- JDBC interagit avec le SGBDR par un *driver*
- Il existe des *drivers* pour Oracle, Sybase, Informix, DB2, Mysql...

Architectures 2-tier et 3-tier



Architecture 2-tier



Architecture 3-tier

Accès aux données

1. Charger le *driver*
2. Connexion à la base
3. Création d'un *statement*
4. Exécution de la requête
5. Lecture des résultats

1. Chargement du *driver*

- **Chaque serveur de gestion de bases de données a son propre driver**

- **Utilisation du chargement dynamique de classes**

```
Class.forName("com.mysql.jdbc.Driver").newInstance();

Class.forName("postgres95.pgDriver").newInstance();

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver").
    newInstance();
```

2. Connexion à la base en utilisant le driver

- La connexion se fait à l'aide d'un **url**
- Exemple :
  - jdbc:mysql://localhost/ma_base
  - jdbc:odbc:ma_base
  - jdbc:pg95:ma_base

- Ouverture de la connexion :

```
Connection conn = DriverManager.getConnection(url, user,
    password);
```

Création d'un *Statement*

- Un statement est un objet qui permet d'exécuter des requêtes SQL

- 3 types de statement :
  - `statement` : requêtes simples
  - `prepared statement : requêtes précompilées`
  - `callable statement : procédures stockées`

- Création d'un *statement* :

```
Statement stmt = conn.createStatement();
```

Execution d'une requête (1/2)

3 types d'executions à travers l'objet Statement :

- `executeQuery` : pour les requêtes qui retournent un `ResultSet (SELECT)`
- `executeUpdate` : pour les requêtes `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE` et `DROP TABLE`
- `execute` : pour quelques cas rares (procédures stockées)

Exécution de la requête :

```
String myQuery = "SELECT prenom, nom,
email " + "FROM employe " +
"WHERE (nom='Dupont') AND (email IS NOT
NULL) " + "ORDER BY nom";

ResultSet rs = stmt.executeQuery(myQuery);
```

Lecture des résultats (1/2)

- `executeQuery()` renvoit un `ResultSet`
- Le ResultSet se parcourt itérativement *ligne* par *ligne*
- Les colonnes sont référencées par leurs numéros ou par leur nom
- L'accès aux valeurs des colonnes se fait par les méthodes `getXXX()` où `XXX` représente le type de l'objet

Lecture des résultats (2/2)

```
ResultSet rs = stmt.executeQuery("SELECT a, b FROM Table1");

while (rs.next())
{
        // print the values for the current row.
        int i = rs.getInt("a");
        String s = rs.getString("b");

        System.out.println("Ligne = "+i+" , "+s);
}
```

Accès aux méta-données

- La méthode `getMetaData()` permet d'obtenir les méta- données d'un `ResultSet`.
- Elle renvoit des `ResultSetMetaData`.
- On peut connaitre :
  - Le nombre de colonne : `getColumnCount()`
  - Le nom d'une colonne : `getColumnName(int col)`
  - Le type d'une colonne : `getColumnType(int col)`
  - ...

Exemple complet :

```
import java.sql.*;

public class TestJDBC {
     public static String driver = "org.gjt.mm.mysql.Driver";

     public static void main(String[] args) throws Exception {
         Class.forName(driver).newInstance();
       Connection connection =
     DriverManager.getConnection("jdbc:mysql://127.0.0.1/mysql",
     "root","");
     Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery("select * from user");
       while (rs.next()) {
              String utilisateur = rs.getString("user");
              String poste = rs.getString("host");
              Boolean select = rs.getBoolean("select_priv");
       System.out.println(utilisateur + " , " + poste + " , " +
     select);
       }
     }
```