

# FSAB1402: Informatique 2

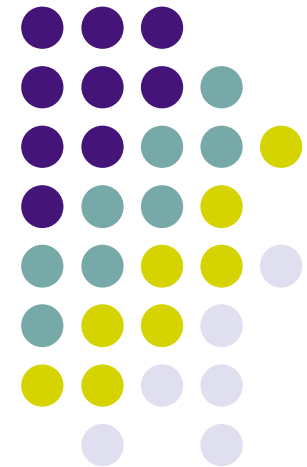
## Le Langage Java et les Exceptions



**Peter Van Roy**

Département d'Ingénierie Informatique, UCL

[pvr@info.ucl.ac.be](mailto:pvr@info.ucl.ac.be)





# Ce qu'on va voir aujourd'hui

- Nous allons voir quelques concepts de Java, un langage populaire basé sur la programmation orientée objet:
  - Le passage de paramètres
  - L'héritage simple
  - Les classes abstraites et finales, la classe Object
  - Les interfaces
- Nous allons aussi voir un autre concept de programmation, les exceptions
  - Les contextes d'exécution
  - Les exceptions en Java

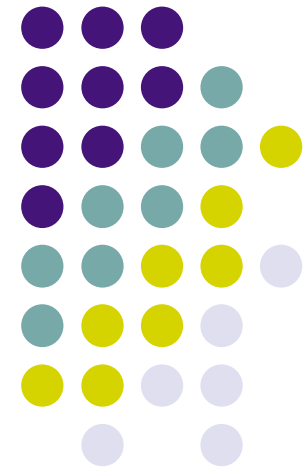


# Lecture pour cette séance

- Chapitre 5 (section 5.4.3):
  - Le passage de paramètres
- Chapitre 6 (section 6.3.4)
  - Les règles de visibilité (contrôle de l'encapsulation)
- Chapitre 6 (section 6.5)
  - Le langage Java
- Chapitre 2 (section 2.7)
  - Les exceptions

# Résumé du dernier cours

---





# Les objets et les classes

- Un objet est une collection de procédures (les “méthodes”) qui ont accès à un état commun (les “attributs”)
  - L’état est accessible uniquement par les méthodes
- L’envoi procédural: il y a un seul point d’entrée à l’objet, qui se comporte comme une procédure avec un argument (le “message”)
- La création de plusieurs objets: des objets avec les mêmes méthodes mais chaque fois un autre état (la fonction NewCounter)
- Une classe: une séparation entre la définition des méthodes et la création de l’objet (la fonction New prend une classe comme argument: {New Counter init})
- Une syntaxe pour les classes et les objets: cela facilite la programmation et garantit qu’il n’y a pas d’erreurs de forme dans la définition des classes et l’invocation des objets



# Le polymorphisme

- Le polymorphisme est le concept le plus important (après l'abstraction!) dans la programmation orientée objet
- Des objets peuvent avoir la même interface mais une implémentation différente
  - {Line draw}, {Circle draw}, {Square draw}, ...
- La même méthode peut marcher avec tous ces objets
  - Polymorphisme: l'appel accepte un argument de types différents (ici, de classes différentes). {F draw} peut marcher quand F est une ligne, un cercle, un carré, une figure composée, etc.
  - Si chaque objet satisfait aux mêmes propriétés, cela marche!
- **Le principe de la concentration des responsabilités**
  - Chaque responsabilité est concentrée dans une partie du programme (par ex., un objet) au lieu d'être morcelée partout

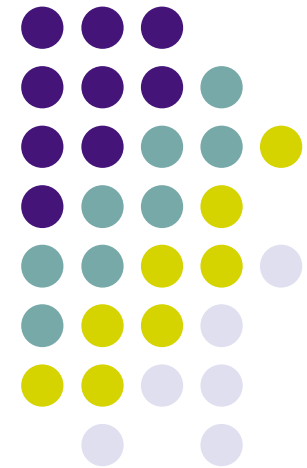


# L'héritage

- La **définition incrémentale** des classes
  - Une classe est définie en prenant une autre classe comme base, avec des modifications et des extensions
  - Lien dynamique (le bon défaut) et lien statique (pour redéfinition)
- L'héritage est **dangereux**
  - La possibilité d'étendre une classe avec l'héritage est une autre interface à cette classe, une interface qui a besoin de maintenance comme les autres!
  - **L'héritage versus la composition**: nous recommandons d'utiliser la composition quand c'est possible
- **Le principe de substitution**
  - Si A hérite de B, alors toute procédure qui marche avec  $O_B$  doit marcher avec  $O_A$
  - Avec ce principe, les dangers sont minimisés

# Introduction à Java

---



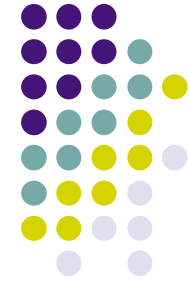




# Le langage Java

- Un langage orienté objet concurrent avec une syntaxe dérivée de C++
- “Presque pur”: presque tout est un objet
  - Un petit ensemble de types primitifs (entiers, caractères, virgules flottantes, booléens) ne l’est pas
  - Les arrays sont des objets mais ne peuvent pas être étendus avec l’héritage
- La différence de philosophie avec C++
  - C++ donne accès à la représentation interne des données; la gestion de mémoire est manuelle
  - Java cache la représentation interne des données; la gestion de mémoire est automatique (“garbage collection”)

# Un petit exemple



```
class Fibonacci {
    public static void main(String [] args) {
        int lo=1;
        int hi=1;
        System.out.println(lo);
        while (hi<50) {
            System.out.println(hi);
            hi=lo+hi;
            lo=hi-lo;
        }
    }
}
```

- Il y a toujours une méthode “public static void main”, exécutée quand l’application démarre
  - Chaque variable (argument ou variable locale) est une cellule, avec un type déclaré statiquement
  - Les entiers ne sont pas des objets, mais des types abstraits
  - Il faut initialiser les variables locales avant de les utiliser
- La méthode println est **surchargée** (il y a plusieurs méthodes avec le même nom; le langage choisit la méthode selon le type de l’argument)



# Public static void main(...)

- La méthode **main** est exécutée quand l'application démarre
- **Public**: visible dans tout le programme (donc aussi en dehors de la classe)
- **Static**: il y en a une par classe (pas une par objet)
- **Void**: la méthode ne renvoie pas de résultat (c'est une procédure, pas une fonction)
- **String[]**: un array qui contient des objets String



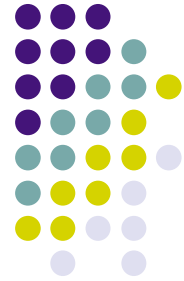
# Types

- Il y a deux sortes de types: **type primitif** et **type référence**
- **Type primitif**: booléen (1 bit), caractère (16 bits, Unicode), byte (entier de 8 bits, -128..127), short (16), int (32), long (64), float (32), double (64)
  - Entiers: une représentation en complément à 2 (!)
  - Virgule flottante: le standard IEEE754
- **Type référence**: classe, interface ou array
  - Une valeur d'un tel type est "null" ou une référence à un objet ou un array
  - Un type array a la forme `t[]` où `t` peut être n'importe quel type



# Modèle d'exécution (1)

- Modèle avec état (cellules) et concurrence (fils)
  - La concurrence en Java est basée sur les fils (“threads”) et les monitors; voir le cours INGI1131
    - C’est assez compliqué; le modèle multi-agents que nous allons voir la semaine prochaine est beaucoup plus simple!
- Typage statique
  - Les types des variables et la hiérarchie des classes sont connus à la compilation
  - Différence avec le langage du cours (Oz) qui a un typage dynamique (types et hiérarchie ne sont connus qu’à l’exécution)
  - En Java le code compilé d’une classe peut être chargé à l’exécution avec un “[class loader](#)”
  - Le langage est [conçu pour le principe de substitution](#): une routine accepte les objets des sous-classes



# Modèle d'exécution (2)

- Soutien pour l'héritage
  - L'héritage simple des classes
  - L'héritage multiple des interfaces
    - Une interface contient juste les entêtes des méthodes (la "signature", c'est comme une classe mais sans les définitions des méthodes), pas leur implémentation
- Abstractions de contrôle
  - If, switch, while, for, break, return, etc.
  - **Programmation structurée**: une série de **blocs imbriqués** où chaque bloc a des entrées et sorties; pas d'instruction "goto"
- Règles de visibilité
  - Private, package, protected, public
  - Chaque objet a une identité unique

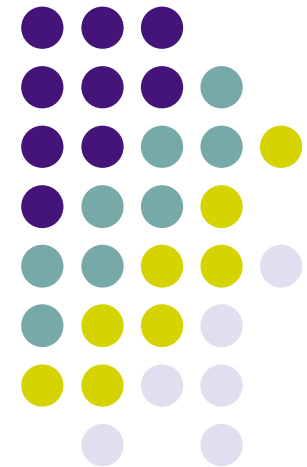


# Modèle d'exécution (3)

- Soutien pour la programmation déclarative
  - Dans ce cours, nous avons expliqué quand la programmation déclarative est préférable
  - Il y a un peu de soutien pour cela en Java
- Attributs/variables “**final**”: peuvent être affectés une fois seulement
  - Ceci permet de faire des objets immuables (sans état)
- Classes “**final**”: ne peuvent pas être étendues avec l'héritage
- “**inner classes**” (classes intérieures): une classe définie à l'intérieur d'une autre
  - Une instance d'une “inner class” est presque une valeur procédurale, mais pas complètement: il y a des restrictions (voir le livre de Arnold & Gosling)

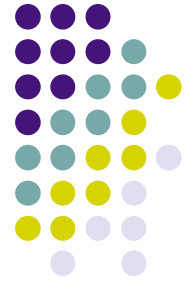
# Le passage de paramètres en Java

---





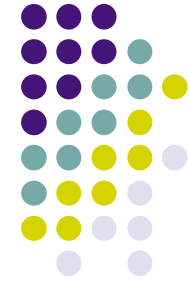
# Le passage de paramètres est par valeur (1)



```
class ByValueExample {
    public static void main(String[] args) {
        double one=1.0;
        System.out.println("before: one = " + one);
        halveIt(one);
        System.out.println("after: one = " + one);
    }
    public static void halveIt(double arg) {
        arg /= 2.0;
    }
}
```

- Qu'est-ce qui est imprimé ici?

# Le comportement de halvelt



```
public static void halveIt(double arg) {  
    arg = arg/2.0;  
}
```

```
proc {Halvelt X}  
    Arg={NewCell X}  
in  
    Arg := @Arg / 2.0  
end
```

- Voici comment on écrit halvelt dans le langage du cours
  - La définition Halvelt peut être vue comme **la sémantique de la méthode halvelt** en Java!
- Le paramètre arg correspond à **une cellule locale** qui est initialisée à l'appel de halvelt

# Le passage de paramètres est par valeur (2)



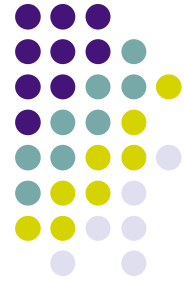
```
class Body {
    public long idNum;
    public String name = "<unnamed>";
    public Body orbits = null;
    private static long nextID = 0;

    Body(String bName, Body orbArd) {
        idNum = nextID++;
        name = bName;
        orbits = orbArd;
    }
}
```

```
class ByValueRef {
    public static void main(String [] args) {
        Body sirius = new Body("Sirius", null);
        System.out.println("bef:"+sirius.name);
        commonName(sirius);
        System.out.println("aft:"+sirius.name);
    }
    public static void commonName(Body bRef) {
        bRef.name = "Dog Star";
        bRef = null;
    }
}
```

- La classe Body a un **constructeur** (la méthode Body) et un **entier statique** (nextID)
- Le contenu de l'objet est bien modifié par commonName, mais mettre bRef à null n'a aucun effet!

# Le comportement de commonName



```
public static void commonName (Body bRef)
{
    bRef.name = "Dog Star";
    bRef = null;
}
```

```
proc {CommonName X}
    BRef={NewCell X}
in
    {@BRef setName("Dog Star")}
    BRef:=null
end
```

- Voici comment on définit commonName dans le langage du cours
- BRef est une cellule locale dont le contenu est une référence à un objet
- Quand on appelle CommonName, BRef est initialisé avec une référence à l'objet sirius



# La sémantique de Body

**declare**

**local**

NextID={NewCell 0}

**class** Body

**attr** idNum

name:"<unnamed>"

orbits:null

**meth** initBody(BName OrbArd)

idNum:=@NextID

NextID:=@NextID+1

name:=BName

orbits:=OrbArd

**end**

**end**

**end**

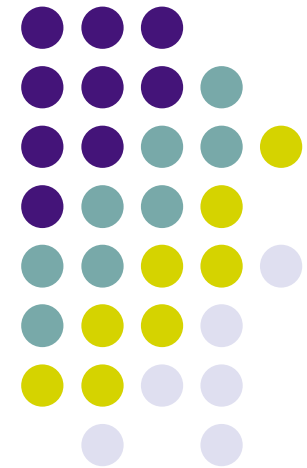
```
class Body {
    public long idNum;
    public String name = "<unnamed>";
    public Body orbits = null;
    private static long nextID = 0;

    Body(String bName, Body orbArd) {
        idNum = nextID++;
        name = bName;
        orbits = orbArd;
    }
}
```

- La définition de Body en Oz donne sa sémantique
- NextID est une **variable statique**: une cellule définie avec la classe
  - Pas comme les attributs qui sont définis avec chaque objet
- Le **constructeur** Body correspond à la méthode initBody (qui initialise un objet)

# Classes et objets en Java

---

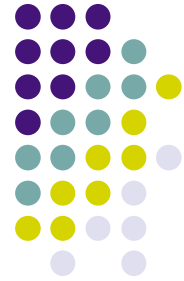




# Concepts de base

- Une classe contient des **champs** (“**fields**”, des attributs ou des méthodes), et des **membres** (“**members**”, autres classes ou interfaces)
- Il n’y a **que de l’héritage simple** des classes
  - Ceci évite les problèmes avec l’héritage multiple
- Liens statiques et dynamiques
  - Le mot-clé “**super**” permet un lien statique avec la classe juste au-dessus
  - Le mot-clé “**this**” est utilisé pour dire “self”

# Un exemple d'héritage



```
class Point {
    public double x, y;

    public void clear() {
        x=0.0;
        y=0.0;
    }
}

class Pixel extends Point {
    Color color;

    public void clear() {
        super.clear();
        color=null;
    }
}
```





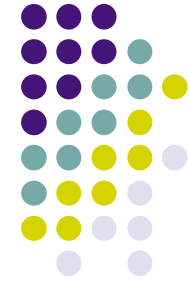
# La classe Object

- La classe Object est la **racine** de la hiérarchie

```
Object oref = new Pixel();  
oref = "Some String";  
oref = "Another String";
```

- La référence oref peut donc référencier **tout objet**
  - On regagne donc une partie de la souplesse du typage dynamique
  - (PS: les objets String sont immuables)

# Classes abstraites et classes concrètes

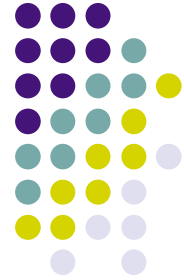


- Une **classe abstraite** est une classe qui ne définit pas toutes ses méthodes (le corps manque)
  - Elle ne peut pas être instanciée
- Une **classe concrète** définit toutes ses méthodes
  - Une classe concrète peut hériter d'une classe abstraite
  - Elle peut être instanciée
- Avec les classes abstraites, on peut faire des programmes “génériques”
  - On définit les méthodes manquantes avec l'héritage, pour obtenir une classe concrète qu'on peut ensuite instancier et exécuter

# Un exemple d'une classe abstraite



```
abstract class Benchmark {  
    abstract void benchmark();  
  
    public long repeat(int count) {  
        long start=System.currentTimeMillis();  
        for (int i=0; i<count; i++)  
            benchmark();  
        return (System.currentTimeMillis()-start);  
    }  
}
```



# Une classe abstraite

- Voici comment on peut faire la même chose avec les valeurs procédurales:

```
fun {Repeat Count Benchmark}
  Start={OS.time}
in
  for I in 1..Count do {Benchmark} end
  {OS.time}-Start
end
```

- La fonction Repeat joue le rôle de la méthode repeat dans la classe Benchmark
- L'argument Benchmark est une procédure qui joue le rôle de la méthode benchmark
- Conclusion: **avec les classes abstraites on peut faire comme si on passait une procédure en argument**
  - On utilise l'héritage pour simuler le passage d'une procédure (= valeur procédurale)



# Les classes “final”

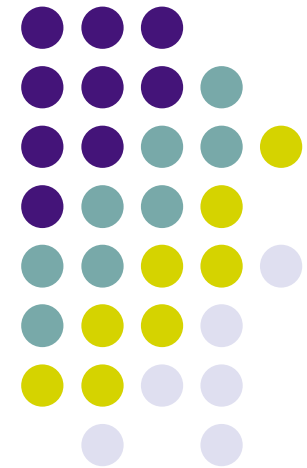
- Une classe “final” ne peut pas être étendue avec l’héritage

```
final class NotExtendable {  
    // ...  
}
```

- Une méthode “final” ne peut pas être redéfinie avec l’héritage
- C’est une bonne idée de définir toutes les classes comme “final”, sauf celles pour laquelle on veut laisser la possibilité d’extension par l’héritage
  - Question: est-ce que c’est une bonne idée de définir une classe abstraite comme final?

# Les interfaces en Java

---

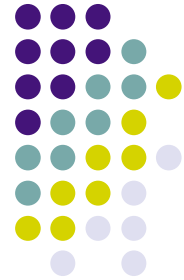




# Les interfaces

- Une **interface** en Java est comme une classe abstraite sans aucune définition de méthode
  - L'interface décrit les méthodes et les types de leurs arguments, sans rien dire sur leur implémentation
- Java permet l'héritage multiple sur les interfaces
  - On regagne une partie de l'expressivité de l'héritage multiple

# Un exemple d'une interface

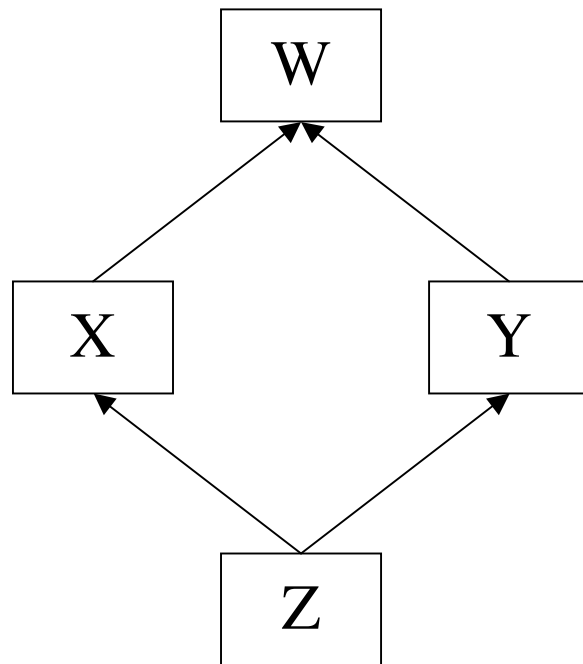
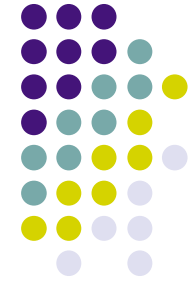


```
interface Lookup {  
    Object find(String name);  
}
```

```
class SimpleLookup implements Lookup {  
    private String[] Names;  
    private Object[] Values;  
    public Object find(String name) {  
        for (int i=0; i<Names.length; i++) {  
            if (Names[i].equals(name))  
                return Values[i];  
        }  
        return null;  
    }  
}
```

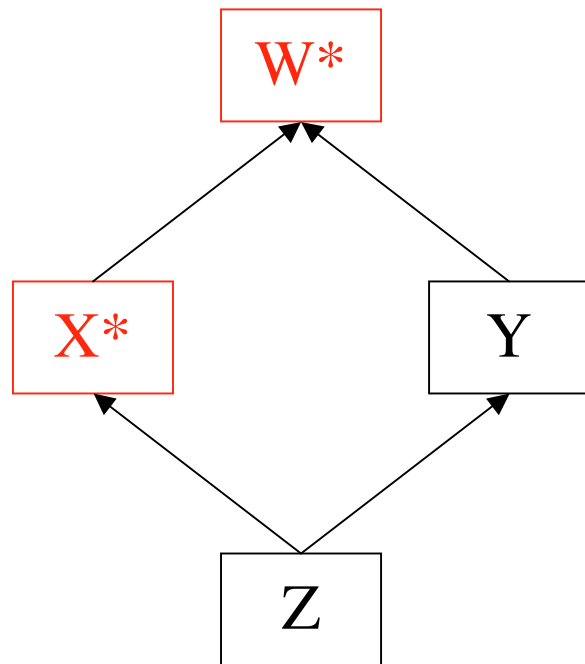


# Le problème avec l'héritage multiple des classes



- Voici un cas où l'héritage multiple classique a un problème: **l'héritage en losange**
- Quand W a de l'état (des attributs), qui va initialiser W? X ou Y ou les deux?
  - Il n'y a pas de solution simple
  - C'est une des raisons pourquoi l'héritage multiple est interdit en Java
- Nous allons voir comment les interfaces peuvent résoudre ce problème

# Une solution avec les interfaces



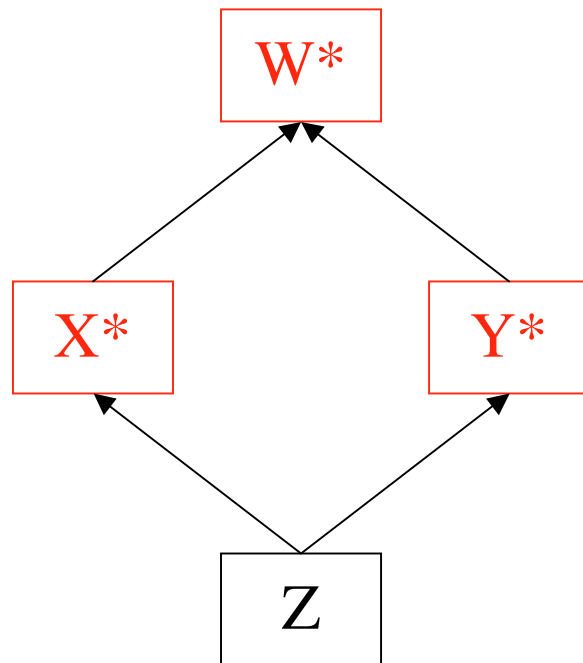
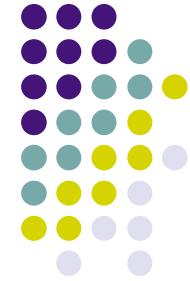
- Les interfaces sont marquées en rouge et avec un astérisque (\*)
- Il n’y a plus d’héritage en losange: la classe Z hérite uniquement de la classe Y
- Pour les interfaces, l’héritage est uniquement **une contrainte sur les entêtes des méthodes** fournies par les classes

# La syntaxe Java pour l'exemple du losange



```
interface W { }  
interface X extends W { }  
class Y implements W { }  
class Z extends Y implements X { }
```

# Une autre solution pour la même hiérarchie

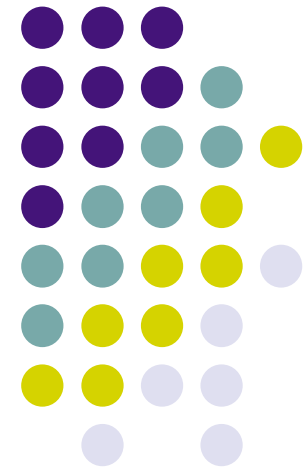


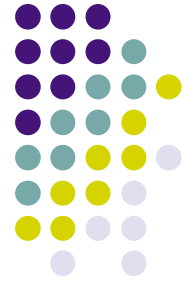
- Cette fois, Z est la seule vraie classe dans l'hiérarchie
- Voici la syntaxe:

```
interface W { }  
interface X extends W { }  
interface Y extends W { }  
class Z implements X, Y { }
```

# Les exceptions

---





# Les exceptions

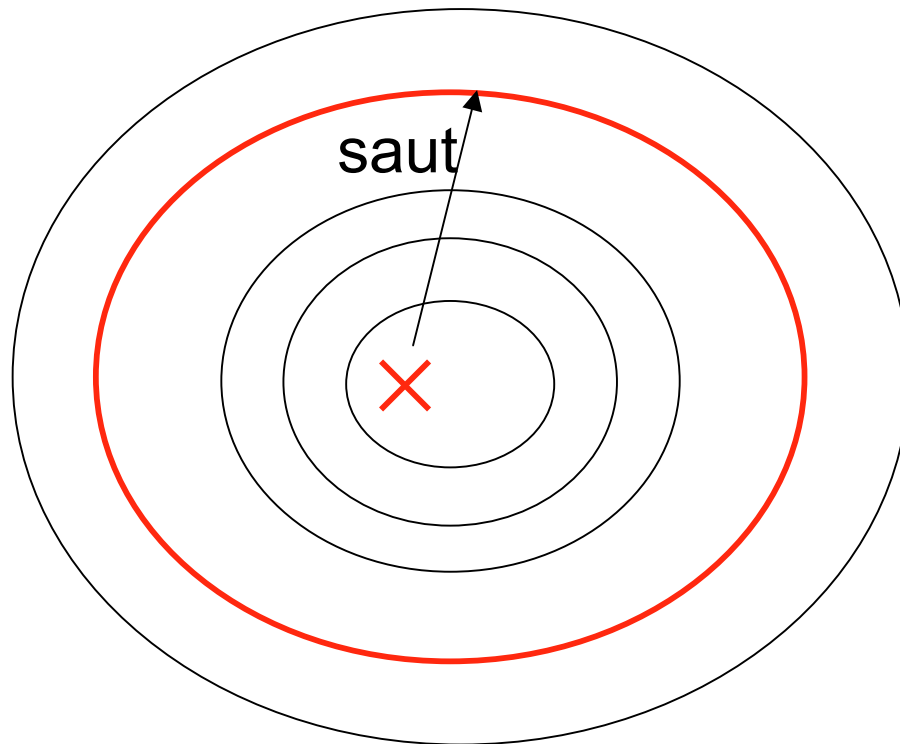
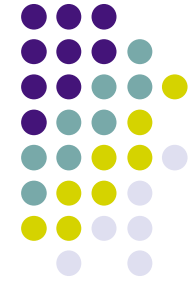
- Les exceptions sont un **nouveau concept de programmation**
  - Nous allons introduire les exceptions et ensuite expliquer comment elles sont réalisées en Java
  - Nous allons aussi donner un aperçu de la sémantique des exceptions
- Comment est-ce que l'on traite les situations exceptionnelles dans un programme?
  - Par exemple: division par 0, ouvrir un fichier qui n'existe pas
  - Des erreurs de programmation mais aussi des erreurs imposées par l'environnement autour du programme
  - En principe, des situations rares mais qu'il faut traiter
- Avec les exceptions, un programme peut gérer des situations exceptionnelles **sans que le code soit encombré** partout avec des bouts qui ne sont presque jamais exécutés



# Le principe d'endiguement

- Quand il y a une erreur, on voudrait se retrouver dans un endroit du programme d'où l'on peut récupérer de l'erreur
- En plus, on voudrait que l'erreur influence la plus petite partie possible du programme
- **Le principe d'endiguement:**
  - Un programme est une hiérarchie de contextes d'exécution
  - Une erreur n'est visible qu'à l'intérieur d'un contexte dans cette hiérarchie
  - Une routine de récupération existe à l'interface d'un contexte d'exécution, pour que l'erreur ne se propage pas (ou se propage proprement) vers un niveau plus élevé

# La gestion d'une exception (1)



✗ Une erreur qui lève une exception

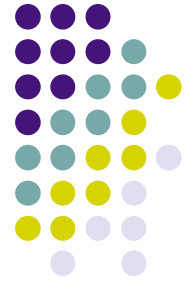
○ Un contexte d'exécution

○ Le contexte d'exécution qui attrape l'exception

Mais c'est quoi exactement un contexte d'exécution?

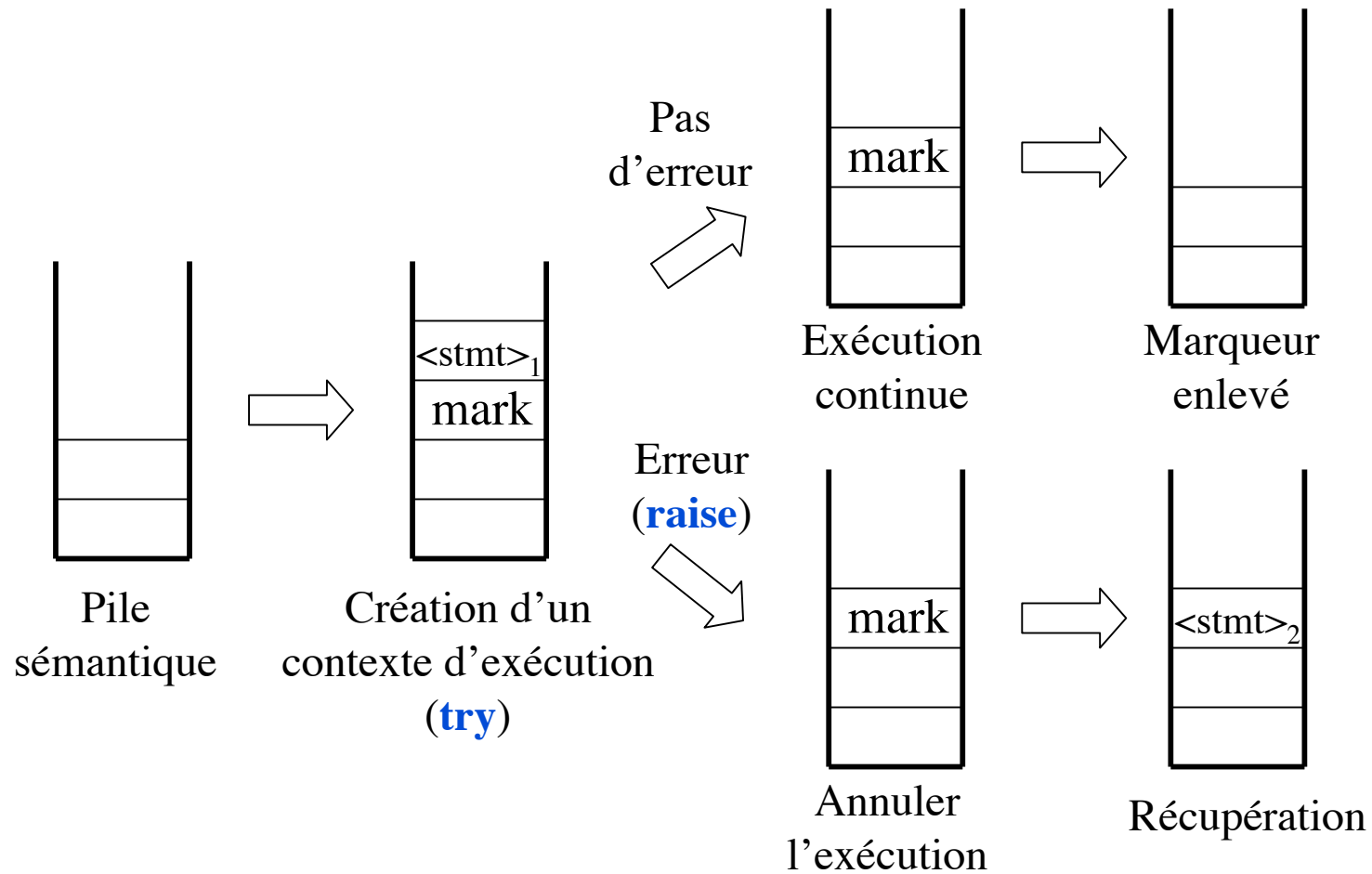


# La gestion d'une exception (2)



- Un programme qui rencontre une erreur doit transférer l'exécution à une autre partie (le gestionnaire d'exceptions) et lui donner une valeur qui décrit l'erreur (l'exception)
- Deux nouvelles instructions  
**try** <stmt><sub>1</sub> **catch** <y> **then** <stmt><sub>2</sub> **end**  
**raise** <x> **end**
- Comportement:
  - **try** met un “marqueur” sur la pile sémantique et exécute <stmt><sub>1</sub>
  - S'il n'y a pas d'erreur, <stmt><sub>1</sub> exécute normalement
  - S'il y a une erreur, le **raise** est exécuté, qui vide la pile jusqu'au marqueur (l'exécution du restant de <stmt><sub>1</sub> est donc annulée)
    - Ensuite <stmt><sub>2</sub> est exécutée et l'exception est donnée par <y>
    - La portée de <y> couvre exactement <stmt><sub>2</sub>

# La gestion d'une exception (3)





# Un contexte d'exécution

- Maintenant on peut définir exactement ce que c'est qu'un contexte d'exécution
- Un **contexte d'exécution** est une partie de la pile sémantique qui commence avec un marqueur et qui va jusqu'au sommet de la pile
  - S'il y a plusieurs instructions **try** imbriquées, alors il y aura plusieurs contextes d'exécution imbriqués!
- Un contexte d'exécution est à l'intérieur d'une seule pile sémantique
  - Avec l'exécution concurrente il y aura plusieurs piles sémantiques (une par thread): voir plus loin dans le cours!
  - En général, c'est une bonne idée d'installer un nouveau contexte d'exécution quand on traverse l'interface d'un composant

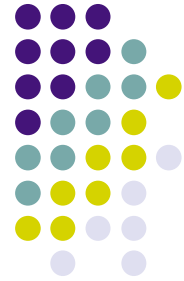
# Un exemple avec des exceptions



```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of plus(X Y) then {Eval X}+{Eval Y}
    [] times(X Y) then {Eval X}*{Eval Y}
    else raise badExpression(E) end
    end
  end
end
```

```
try
  {Browse {Eval plus(23 times(5 5))}}
  {Browse {Eval plus(23 minus(4 3))}}
catch X then {Browse X} end
```

# S'il n'y avait pas d'exceptions



```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of plus(X Y) then R={Eval X} in
      case R of badExpression(RE) then badExpression(RE)
      else R2={Eval Y} in
        case R2 of badExpression(RE) then badExpression(RE)
        else R+R2
        end
      end
    [] times(X Y) then
      % Comme avec plus
    else badExpression(E)
    end
  end
end
```

- Beaucoup plus de code!
  - Dans notre exemple, 22 lignes au lieu de 10 (plus que le double)
- En plus, le code est plus compliqué à cause du traitement de “badExpression(E)”

# Instruction avec clause “finally”



- L’instruction **try** permet aussi la clause **finally**, pour une opération qui doit être exécutée dans tous les cas de figure (erreur ou pas erreur):

```
FH={OpenFile “foobar”}
```

```
try
```

```
    {ProcessFile FH}
```

```
catch X then
```

```
    {Show “*** Exception during execution ***”}
```

```
finally {CloseFile FH} end
```



# Exceptions en Java

- Une exception est un objet qui hérite de la classe Exception (qui elle-même hérite de Throwable)
- Il y a deux sortes d'exceptions
  - **Checked exceptions**: Le compilateur vérifie que les méthodes ne lèvent que les exceptions déclarée pour la classe
  - **Unchecked exceptions**: Il y a certaines exceptions et erreurs qui peuvent arriver sans que le compilateur les vérifie. Elles héritent de RuntimeException et Error.

# Syntaxe des exceptions Java



```
throw new NoSuchElementException(name) ;
```

```
try {  
    <stmt>  
} catch (exctype1 id1) {  
    <stmt>  
} catch (exctype2 id2) {  
    ...  
} finally {  
    <stmt>  
}
```





# Un exemple en bon style

- Nous allons lire un fichier et faire une action pour chaque élément:

```
try
    while (!stream.eof())
        process(stream.nextToken());
finally
    stream.close();
```



# Un exemple en mauvais style

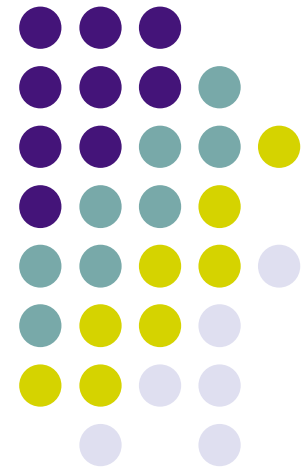
- C'est **très mauvais** d'utiliser les exceptions pour modifier l'ordre d'exécution dans une situation normale:

```
try {  
    for (;;)   
        process (stream.next());  
} catch (StreamEndException e) {  
    stream.close();  
}
```

- Trouver la fin d'un stream est tout à fait **normal**, ce n'est pas une erreur. Qu'est-ce qu'on fait si une **vraie erreur** se mélange avec cette exécution normale?

# Résumé

---

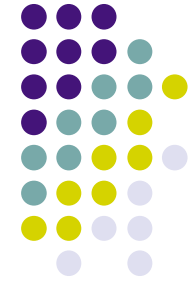


# Résumé



- Introduction au langage Java
  - Relation avec C++
- Le passage des paramètres en Java
  - Passage par valeur
  - “La référence à un objet est passée par valeur”
- Classes et objets en Java
  - L’héritage simple
  - La classe Object
  - Les classes abstraites et les classes “final” (finales)
- Les interfaces en Java
  - L’héritage multiple avec les interfaces
- Les exceptions
  - Les contextes d’exécution
  - Leur sémantique
  - Les exceptions en Java

# Bibliographie pour Java



- *The Java Programming Language, Second Edition*, par Ken Arnold et James Gosling, Addison-Wesley, 1998
  - Le livre de référence écrit par deux des concepteurs principaux du langage
- *Java Precisely*, par Peter Sestoft, MIT Press, 2002
  - Un très bon livre sur la sémantique de Java, lisible avec beaucoup d'exemples de code