

# Programmation JAVA

Classes, héritage, polymorphisme, exceptions

Pierre Ramet

`ramet@labri.fr`

IUT Bordeaux 1

Stage Programmation Licence, 2014

# Plan de l'exposé

- 1 Classes, Objets
- 2 Associations
- 3 Héritage et polymorphisme
- 4 Exceptions

# Plan de l'exposé

- 1 Classes, Objets
- 2 Associations
- 3 Héritage et polymorphisme
- 4 Exceptions

# Principe de la programmation objet I

- Type de données et traitements associés sont placés dans le même conteneur, appelé **classe**.
  - données d'une classe : données membres, attributs
  - traitements d'une classe : fonctions membres, méthodes, opérations
- Une réalisation concrète d'une classe est un **objet**. On parle d'**instance** de classe.
- Les données sont **encapsulées** dans la classe. En général, seules les méthodes de l'objet ont accès à ses attributs.
- Les objets communiquent par **messages**
  - plusieurs objets collaborent pour réaliser une tâche complexe

## Principe de la programmation objet II

- envoi de message  $a \xrightarrow{\text{msg}(x,y)} b$   
⇔ dans une méthode de  $a$ , appel de la méthode  $\text{msg}$  de  $b$  avec paramètres  $(x, y)$ .
- Les collaborations se matérialisent par des **relations** structurelles entre classes : associations, agrégations, etc
- Les classes peuvent être regroupées ou inversement spécialisées : **héritage**, généralisation/spécialisation.
- Des objets de types distincts, même regroupés sous un même type, peuvent conserver un comportement spécifique : **polymorphisme**

# Types primitifs et classes I

- **types primitifs** : types prédéfinis avec des opérations de base (addition, etc), non extensibles. Base pour la définition de types plus complexes.

Ex : `int`, `float`, `double`, `char`, `boolean`, ...

Variable de type primitif : *type nom\_variable*

```
{  
    int a = 10;           // déclaration et initialisation  
    a = a + a;           // opération  
}                         // la variable a n'existe plus
```

- **Classe** : encapsule les données dans ses **attributs** et contient des **méthodes** pour faire des traitements dessus.

# Types primitifs et classes II

```
public class Etudiant {  
    // methodes  
    public Etudiant( String n, int note1, int note2 )  
    {  
        nom = n;  
        notes = new int[ 2 ];  
        notes[ 0 ] = note1;  
        notes[ 1 ] = note2;  
    }  
    ... // attributs  
    private int[] notes;  
    public String nom;  
}
```

- type d'un attribut : type primitif ou classe de l'API ou classe définie par l'utilisateur

## Types primitifs et classes III

- méthode : *modifieur val\_retour nom ( liste\_paramètres )* {  
... code ... }  
modifieur : indicateur de portée (`public ...`), `final`,  
`abstract ...`



# Instanciation des classes I

- Une variable dont le type  $T$  est une classe est une **variable objet**
- Une variable objet **n'est pas** un objet (en JAVA), c'est un(e) **référence/pointeur** soit vide (`null`) soit vers un objet de type  $T$ .
- Une **instance** de classe  $T$  (ie un **objet** de type  $T$ ) est créée par la commande **new**.
- Lorsqu'un objet n'est plus référencé, il est détruit par un processus asynchrone, le **garbage collector**.

# Instanciation des classes II

```
Etudiant etd; // etd est une variable objet  
// Mais vaut null. Pas d'instance creee.  
if ( etd == null ) System.out.println( "Oui." );  
etd = new Etudiant( "Gerard Mer", 12, 14 );  
// Instanciation avec appel de constructeur.
```

# Plan de l'exposé

- 1 Classes, Objets
- 2 Associations**
- 3 Héritage et polymorphisme
- 4 Exceptions

# Programmation objet et associations

- Souvent plusieurs objets doivent collaborer pour réaliser une tâche. Exemples
  - Un groupe connaît tous ses étudiants pour pouvoir calculer la moyenne dans un module donné. Un étudiant connaît les modules qu'il suit.
  - Dans une interface, une fenêtre connaît les éléments qui la composent pour pouvoir se réafficher ou faire une bonne mise en page.
- Certains objets sont donc **liés**. Certaines classes sont donc **associées**.
- Les associations ne sont pas forcément symétriques et peuvent avoir différentes multiplicités.
- Quelles sont les associations que vous connaissez ?
- **agrégation, composition**

# Programmation objet et associations

- Souvent plusieurs objets doivent collaborer pour réaliser une tâche. Exemples
  - Un groupe connaît tous ses étudiants pour pouvoir calculer la moyenne dans un module donné. Un étudiant connaît les modules qu'il suit.
  - Dans une interface, une fenêtre connaît les éléments qui la composent pour pouvoir se réafficher ou faire une bonne mise en page.
- Certains objets sont donc **liés**. Certaines classes sont donc **associées**.
- Les associations ne sont pas forcément symétriques et peuvent avoir différentes multiplicités.
- Quelles sont les associations que vous connaissez ?
- **agrégation, composition**

# Associations I

- Les associations de multiplicité 0 ou 1 se font simplement en introduisant un attribut du bon type.

```
public class Societe { ... }  
public class Employe  
{ ...  
    public float salaire() { ... }  
    ...  
    Societe ma_societe;  
}
```

- Les associations de multiplicité supérieure se font avec un tableau (si taille connue) ou à l'aide d'un objet modélisant une collection : Vector, Set, ...

## Associations II

```
public class Societe  
{ ...  
    Employe[] mes_employes;  
}
```

Ecrire une méthode qui calcule la masse salariale de la société.

- Dans l'exercice précédent, l'instance de société a envoyé un message à tous les objets `Employe` associés.
- Une agrégation est parfois appelée agrégation de référence par opposition à agrégation par valeur (appelée également composition).

## Associations III

- En JAVA, association simple, agrégation et composition se mettent en oeuvre similairement. La durée de vie des objets associés est déterminée par les constructeurs et par les références actives.



# Plan de l'exposé

- 1 Classes, Objets
- 2 Associations
- 3 **Héritage et polymorphisme**
  - Héritage
  - Transtypage
  - Redéfinition ; polymorphisme
  - Interfaces
- 4 Exceptions

# Héritage I

## Héritage

B **hérite** de A : B a toutes les fonctionnalités de A et peut en rajouter d'autres


- B est une **sous-classe** de A, B dérive de A, A une super-classe de B  
Intuitivement, A est plus générale que B et B est plus spécialisée que A.

## Héritage II

- Relation d'héritage  $\Rightarrow$  généralisation, spécialisation
  - Rend compte des ressemblances entre objets au niveau conceptuel et permet de grouper des objets de comportements assez proches.
  - Délimite mieux les rôles de chaque objet
  - Permet de partager le code
- En JAVA, s'écrit : `class B extends A { ... }`

Faire la première question de l'exercice 1 de Formes.

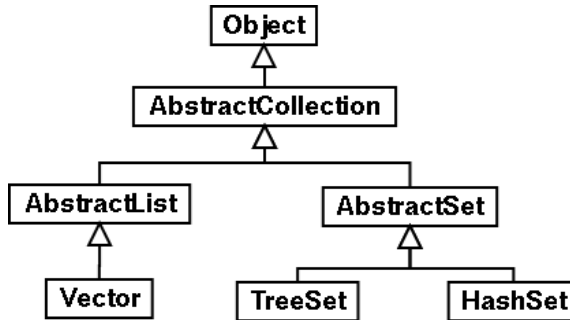
# Hiérarchie d'héritage I

- **Hiérarchie d'héritage** = graphe orienté de toutes les relations d'héritages
- Si une classe n'hérite au maximum que d'une autre  **arbre ou forêt d'héritage**.
- C'est le cas en JAVA. Une classe ne peut dériver de plusieurs classes.

En JAVA, c'est un **arbre**. Pourquoi ? Quelle est la racine ?

# Hérarchie d'héritage II

- Extrait de la hiérarchie `java.util`



# Héritage : précisions I

- si B hérite de A, alors B a accès aux attributs et méthodes `public` et `protected` de A.  
Les attributs et méthodes de(s) superclasse(s) sont utilisés simplement en donnant leur nom.

Dans `Formes`, trouvez des exemples d'utilisation de méthodes de superclasse.

- La partie `private` de A est inaccessible à partir de B.
- Toute instance de B **est** aussi une instance de A

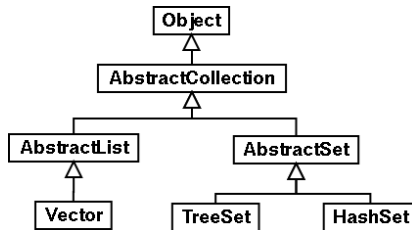
L'objet `visu`, déclaré dans `VisualiseurDeFormes.main` est une instance de `JFrame`, un peu plus spécifique.

## Héritage : précisions II

- Les constructeurs de B peuvent appeler un constructeur spécifique de A en écrivant **super**( ... ); dans leur code.

Trouvez dans `Formes` un exemple d'utilisation de `super`. A quoi sert-il ?

# Héritage : précisions I



D'après la hiérarchie précédente, combien y a-t-il d'objets instanciés par la commande :

`Objet a = new Vector(); ?`

Même question avec : `Objet b = new TreeSet();`



# Transtypage I

- Si B hérite de A, alors un objet de type B est aussi une instance de A

⇒ un objet de type B peut être utilisé comme un objet A partout !

- comme argument d'appel pour une méthode attendant un A
- une variable objet de type A peut référencer un objet B

```
public void fct( A un_a )  
...  
B b = new B();  
A a1 = new B(); // Valide  
A a2 = b;       // Valide  
fct( b );       // Valide  
fct( a1 );      // Valide
```

## Transtypage II

- Inversement, on peut rechercher à retrouver le type d'un objet à son instantiation
- **Transtypage** : opération explicite qui consiste à référencer un objet avec un type dérivé du type de la référence initiale

```
A a = new B(); // a reference en fait une instance de B
B b1 = a;      // Invalide, un A n'est a priori pas un B
B b2 = (B) a;  // Transtypage de A vers B
// Ecriture valide et exécution correcte
```

- si l'objet référencé n'était pas de ce type-là, une **exception** est levée à l'**exécution**.

# Redéfinition ; polymorphisme I

- **redéfinition** : réécriture d'une méthode d'une superclasse dans une classe dérivée (*overriding*)
- Cela permet de spécialiser certains comportements.

```
class Humain
    public String sexe()  return "Inconnu";

class Homme extends Humain
    public String sexe()  return "Masculin";

class Femme extends Humain
    public String sexe()  return "Feminin";

...
Human h1 = new Homme();
Human h2 = new Femme();
System.out.println( h1.sexe() ); // Masculin
```

# Redéfinition ; polymorphisme II

```
System.out.println( h2.sexe() ); // Feminin
```



A l'exécution, c'est le type d'**instanciation** de l'objet pointé par la variable objet qui détermine la méthode appelée.

- **Polymorphisme** : traiter plusieurs formes d'une classe comme si elles n'en étaient qu'une, sans perte de leurs spécificités
- On voit que l'héritage et la redéfinition offrent un mécanisme de polymorphisme.

Dans `Formes`, identifier des méthodes redéfinies.

# Interfaces I

- **Interface** : spécification d'un ensemble de comportements au travers de la déclaration d'une liste de méthodes.

```
public interface Affichable {  
    public void affiche();  
}
```

- Une classe peut alors **réaliser** une interface en **redéfinissant** chaque méthode de l'interface. On écrit

```
public class Etudiant implements Affichable {  
    ...  
    public void affiche() // Methode redefinie  
    {  
        System.out.println( "Mon nom est " + nom );  
    }  
}
```

## Interfaces II

- De façon identique à l'héritage, une instance d'`Etudiant` **est** aussi une instance d'`Affichable`.

```
Affichable a = new Etudiant( ... ); // Valide
```

### Interface et héritage

Le mécanisme de réalisation d'interface est une autre façon pour faire de l'héritage et du polymorphisme. Une classe hérite au maximum d'une autre mais peut implémenter un nombre quelconque d'interfaces.

Attention, une interface ne contient **jamais** du code ou des attributs.

# Interfaces III

Terminez maintenant les exercices de `Formes`.

# Plan de l'exposé

- 1 Classes, Objets
- 2 Associations
- 3 Héritage et polymorphisme
- 4 Exceptions



# Exceptions I

- Les exceptions forment un mécanisme élégant pour traiter les erreurs à l'exécution

## Exceptions II

La méthode `setSalaire` de `Employe` attend un chiffre supérieur au SMIC. Dans ce cas, on peut décider

- que le programme se termine. Rude !
- d'ignorer le problème en affectant quand même ce salaire, mais l'objet est alors invalide
- de ne pas changer le salaire de l'objet, mais l'utilisateur n'est pas prévenu du problème
- de modifier `setSalaire` pour qu'il renvoie un code d'erreur : pas très élégant
- d'essayer de prévenir l'objet appelant que son message est invalide : mais en JAVA, on ne connaît pas l'émetteur

# Exceptions III

- **Exception** : événement qui apparaît pendant l'exécution d'un programme et qui interrompt le flot normal d'exécution
- throw** Si une erreur se produit dans une méthode, celle-ci peut créer un objet dérivant de **Exception** et le donner au système qui exécute le programme. L'exception est **levée**.
- catch** Le système recherche ensuite dans la pile d'exécution (ie les méthodes appelantes), une méthode qui sait **gérer** cette exception. Le système lui donne la main et la méthode a **attrapé** l'exception.
- Une fois l'exception attrapée, le code de gestion d'exception est exécuté et le flot normal d'exécution repart de cet endroit.

## Exceptions IV

```
public class Employe {  
    public void setSalaire(float s)  
        throws Exception  
    {  
        if ( s < SMIC )  
            throw new Exception  
                ( "Salaire trop bas" );  
        m_salaire = s;  
    }  
}  
...
```

```
Employe e1; ...  
float s1; ...  
boolean loop = true;  
while ( loop ) {  
    try {  
        // s1 est renseigné  
        // par l'utilisateur  
        e1.setSalaire( s1 );  
        loop = false;  
    }  
    catch ( Exception e ) {  
        System.err.println( e );  
        System.err.println  
            ( "Resaisissez le salaire" );  
    }  
}
```

# Exceptions : précisions I

- Toute classe dérivant de `Exception` est une exception. Vous pouvez donc créer vos classes exceptions.
- Une méthode qui peut lever une exception `E` doit le déclarer avec `throws E`.
- Une méthode qui utilise cette méthode doit alors
  - soit **attraper** `E` autour de l'appel,
  - soit indiquer qu'elle peut aussi lever `E`. La gestion de l'exception est alors déléguée à la méthode appelante.

# Exceptions non vérifiées I

- Toute exception dérivant de `RuntimeException` n'a pas besoin d'être déclarée
- Si une telle exception se produit et qu'elle n'est attrapée nulle part, alors la machine virtuelle s'arrête et indique la localisation de l'exception.

Reprendre la fin de l'exercice 4 de `Formes`. Vérifiez que l'exception levée dérive bien de `RuntimeException`. Attrapez maintenant l'exception pour éviter l'erreur.