

# *Relations, Méthodes, Références, et Classes*

---

*Lexique et Mots réservés*

# *Objets, Responsabilités, Collaborations*

---

## ◆ Objets

- de quels objets est formé le système?

## ◆ Responsabilités

- quel objet accomplit quelle action?

## ◆ Collaborations

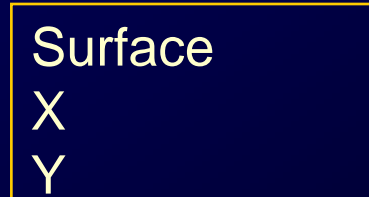
- quel objet fournit un service à quel autre objet?
- quelles sont les relations entre objets?
  - Permet de déterminer le flow de contrôle à l'exécution
  - Aide à corriger la conception

# Relations

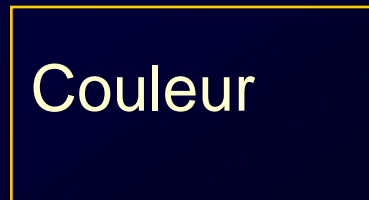
---

- ◆ Identifier les Responsabilités
- ◆ **is-a** (est-un)
  - sous-classe / super-classe (construction, spécialisation)
  - responsabilités vont à la super-classe
    - *un cercle coloré **est-un** cercle*
- ◆ **has-a** (possède)
  - un objet est formé de plusieurs autres objets (attributs)
  - pas de relation d'héritage (partition, aggrégation, composition)
    - *un corps **possède** des bras, une tête, des jambes, ...*

Cercle



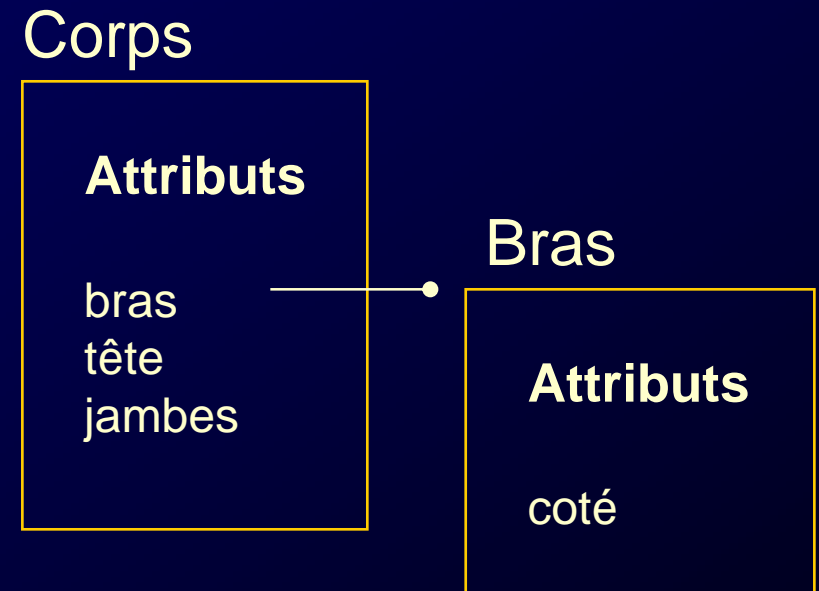
CercleColoré



# Relations

---

- ◆ Identifier les Collaborations
- ◆ **is-part-of** (fait-partie-de)
  - classes composite
    - *un bras fait-partie-de un corps*
    - classe corps est responsable de ses parties
      - on accède aux parties en passant par le corps
  - classes container
    - *un élément fait-partie-de un tableau*
    - classe tableau stocke uniquement les éléments
      - on accède aux éléments sans passer par le tableau



# Relations

---

## ◆ has-knowledge-of (sait-que)

- Avoir la connaissance d'une classe sans être composé d'éléments de cette classe
  - *le cerveau donne des ordres à l'estomac pour digérer, mais le cerveau n'est pas « formé » de l'estomac*

## ◆ depends-upon (dépend-de)

- Un objet doit changer si un autre objet change
  - *A l'écran un objet A masque un objet B, si A disparaît, alors B doit se redessiner complètement*

# Spécialisation vs Composition

---

## ◆ Question:

- Comment faire pour obtenir une classe Cube à partir d'une classe Carré?
- Faut-il choisir: **is-a** ou **has-a** ?

## X Spécialisation (is-a)

- classe Cube est une sous-classe de Carré, on ajoute la méthode Volume()

## Composition (has-a)

- classe Cube est une nouvelle classe composée de 6 faces, chacune étant un Carré.

# Exemple

---

```
class Carre{  
  int x, y; float cote; Ecran ec;  
  
  public Carre(int a, int b, float h,  
    Ecran e)  
    {x=a; y=b; cote = h; ec = e;}  
  
  public float Surface()  
    { return cote*cote;}  
  
  public int X() { return x;}  
  public int Y() { return y;}  
  
}
```

# Spécialisation vs Composition

```
class Cube extends Carre{  
  
    public Cube(int a, int b, float h,  
        Ecran e)  
        {x=a; y=b; cote = h; ec = e;}  
  
    public float Surface()  
        { return cote*cote*6;}  
  
    public float Volume()  
        {return cote*cote*cote;}  
}
```

```
class Cube {  
    int x, y; float cote; Ecran ec;  
    Carre c1,c2,c3,c4,c5,c6;  
    public Cube(int a, int b, float h, Ecran e)  
        {x=a; y=b; cote = h; ec = e;  
        c1= new Carre(a,b,h,e); c2 = ....}  
  
    public int X() { return x;}  
    public int Y() { return y;}  
    public float Surface()  
        { return c1.Surface()*6;}  
    public float Volume()  
        {return c1.Surface()*cote;}  
}
```



# *Spécialisation vs Composition*

---

- ◆ Choisir la Spécialisation si ...
  - L'interface de la super-classe est une interface autorisée pour la sous-classe (is-a s'applique au niveau des types)
  - Prototypage
- ◆ Choisir la Composition si ...
  - Has-a s'applique
  - Certaines opérations de la classe à étendre ne sont pas désirées
  - La classe de départ procure une implémentation particulière de la nouvelle classe à produire (ce n'est pas un vrai is-a)
  - On désire une interface à utiliser clairement définie

# *Spécialisation vs Composition*

---

## ◆ A méditer:

- Etant donné de l'eau et de la pollution
- Qu'est-ce que de l'eau polluée?
  - Un nouveau type d'eau (spécialisation d'eau avec un degré de pollution)
  - Un mélange eau et pollution (héritage multiple ou nouvelle classe avec composition)

# Méthodes

---

## ♦ Messages

- nom de la méthode + arguments

```
result = obj.meth(arg1, arg2, arg3)
```

- *c = new Carre(1,2,3.0,e)*
- *Carre / 1 2 3.0 e / Carre*

## ♦ Signature

- **nom** + **types des paramètres** + **type de la valeur de retour**
  - *public Carre(int a, int b, float h, Ecran e)*
  - *Carre / (int, int, float, Ecran) / Carre*
- définit ce qui est nécessaire pour utiliser la méthode

# Méthodes Particulières

---

## ◆ Constructeurs

- Méthodes particulières pour créer et initialiser des instances
  - *public Carre(int a, int b, float h, Ecran e)*
  - *c = new Carre(1,2,3.0,e)*

## ◆ Setter, Getter

- Méthodes particulières pour accéder à un attribut
  - *public void setX(int a) { x = a; }*
  - *public int getX() { return x; }*
  - *c.setX(2); c.getX();*

# Références vs Valeurs

---

- ◆ Passage de paramètres par valeur
  - *c.setX(2)*
- ◆ Passage de paramètres par référence
  - *c = new Carre(1,2,3,e);*
- ◆ Java
  - Types primitifs: passage par valeur
  - Types composés: passage par référence

# Références

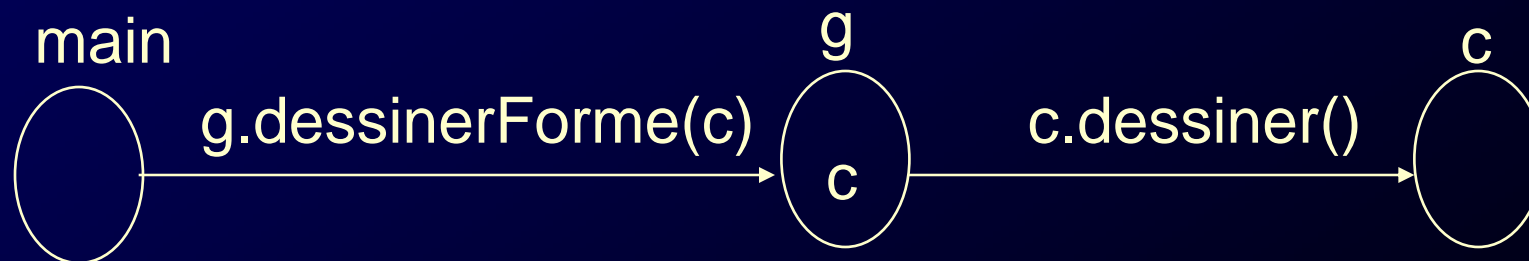
---

- ◆ Tout objet qui possède une référence sur un objet O peut lui envoyer un message !
  - Un objet est une donnée partagée mais avec plus de sémantique
    - Un pointeur sur une structure Cercle peut être utilisé pour lire et écrire les données dans la structure
    - Mais une référence sur un objet Cercle peut être utilisée pour dessiner ou calculer la surface du Cercle
      - Seule la classe de l'objet décide comment l'objet est modifié ou lu
      - Une utilisation du cercle est toujours correcte

# Références

```
class Test{  
  
    main() {  
        g = new GestionnaireEcran();  
        c = new Carre(1,2,3.0,e);  
        g.dessinerForme(c);  
    }  
}
```

```
class GestionnaireEcran {  
  
    public GestionnaireEcran() {super();}  
  
    public void dessinerForme(Forme c)  
        {c.dessiner();}  
}
```



# Information Hiding

---

- ◆ La programmation objet rend plus facile l'intégration de composants
  - Un composant a simplement besoin de connaître l'interface d'un autre composant pour l'utiliser
  - On peut donc modifier l'implantation d'une classe sans toucher aux autres composants de l'application
  - Notamment, le choix des variables dans une classe ainsi que les algorithmes choisis pour les méthodes
- ◆ *Information hiding*
  - Si un programmeur n'a pas besoin de savoir alors il ne doit pas savoir



# Information Hiding

---

- ◆ Interface définit ce qui est visible et invisible
- ◆ Mots réservés Java
  - **public**: la méthode fait partie de l'interface
  - **protected**: la méthode est visible dans le même package et dans les sous-classes
  - **private**: la méthode n'est visible que dans la classe, elle ne fait pas partie de l'interface (pas visible des sous-classes)
  - **void**: la méthode ne retourne rien
  - **static**: méthode de classe
    - invoquée par l'intermédiaire du nom de la classe
      - `Carre.maMethodeStatique()`

# La protection

---

- ◆ Les erreurs dans un objet sont contenues
  - Elles ne peuvent pas nuire aux variables dans d'autres objets
  - P.ex: si dans l'objet **c**, l'exécution de la méthode **surface** provoque un débordement de valeur, **c2** et **c1** ne sont pas touchés
  - Cela n'est pas le cas avec le langage C à cause des pointeurs !
- ◆ Mais l'appelant de **c.Surface()** est bloqué en attente d'une réponse
  - Il faut un mécanisme supplémentaire : *le traitement d'exceptions* (plus tard dans le cours)

# Classes

---

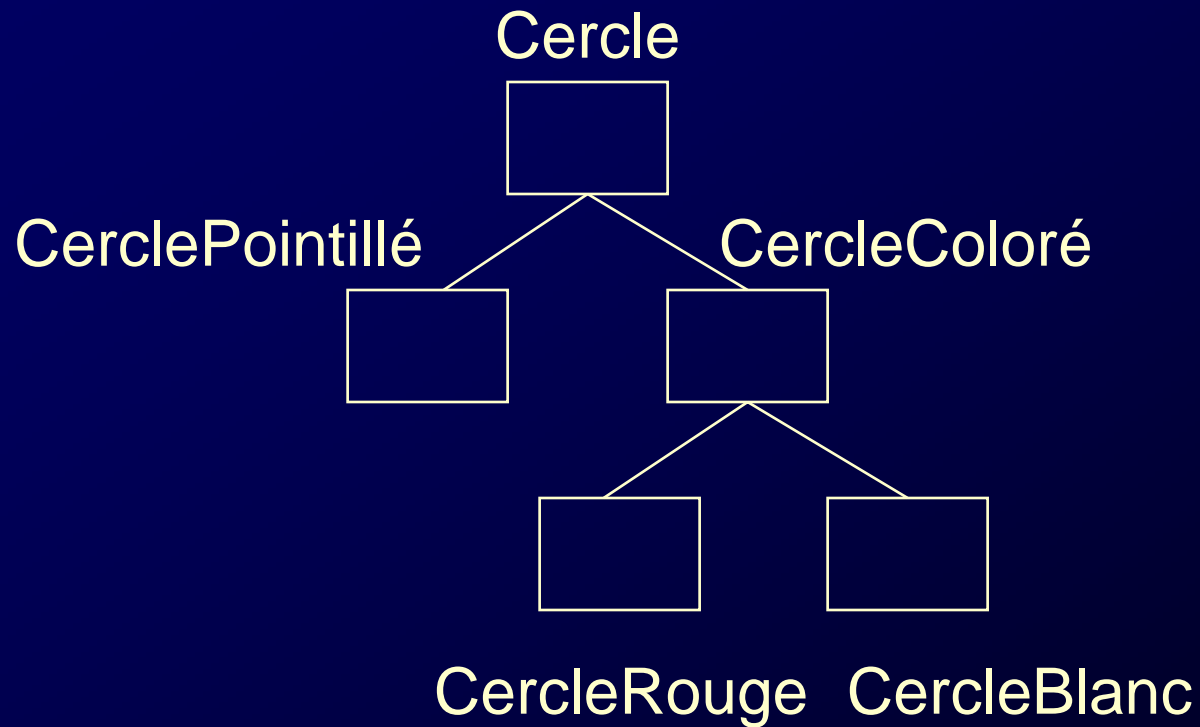
- ◆ Généraliser
- ◆ Super-Classes
- ◆ Parent
- ◆ Ancêtres
- ◆ Spécialiser
- ◆ Sous-Classes
- ◆ Enfant
- ◆ Descendant

*On utilise le terme **sous**-classe pour indiquer que les objets qui correspondent à ce type forment un **sous**-ensemble des objets de la classe parente*

*Un CercleColoré est un Cercle, mais tout Cercle n'est pas forcément un CercleColoré*

# Classes

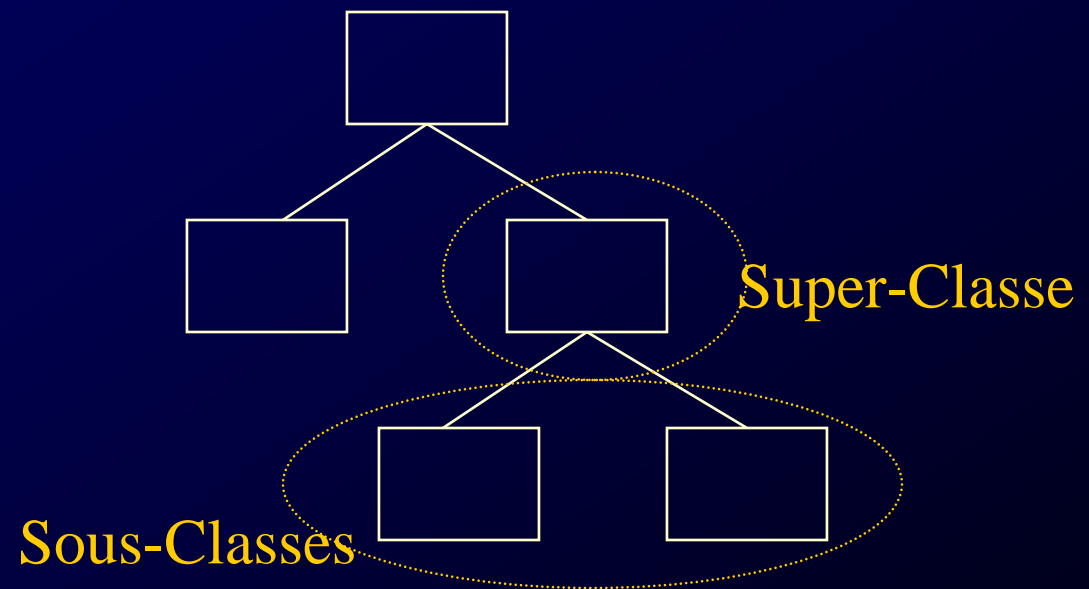
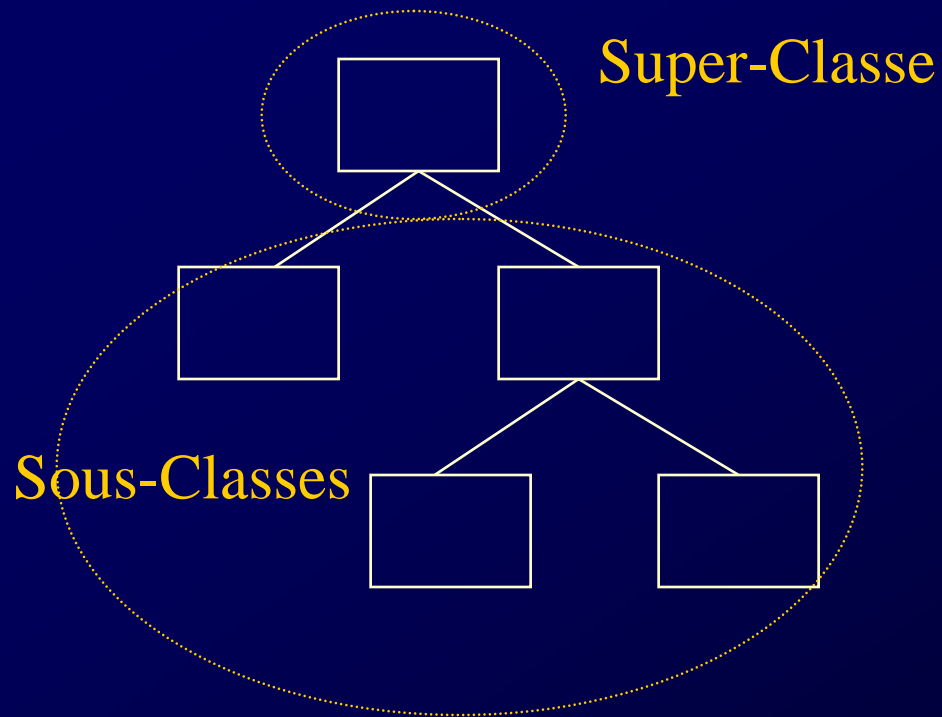
---



- ◆ Mot réservé Java: extends

# Classes

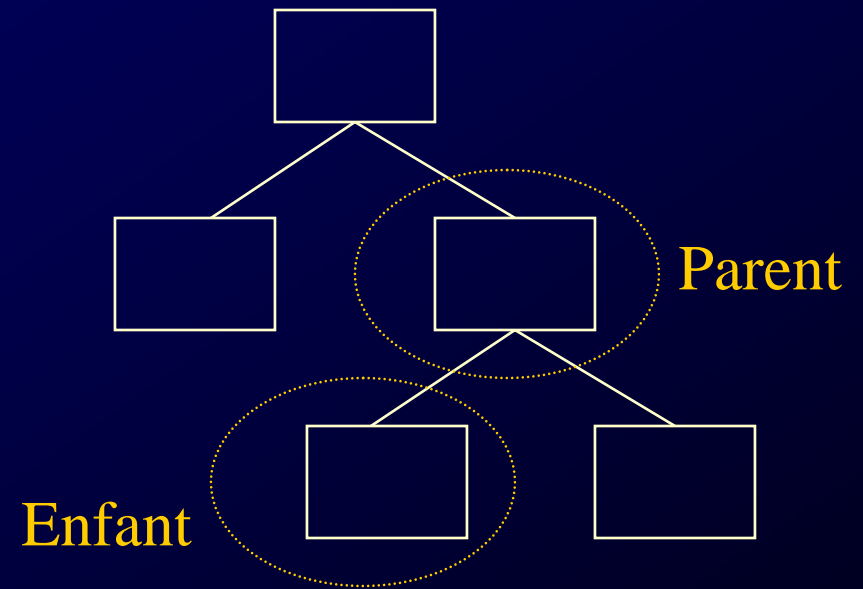
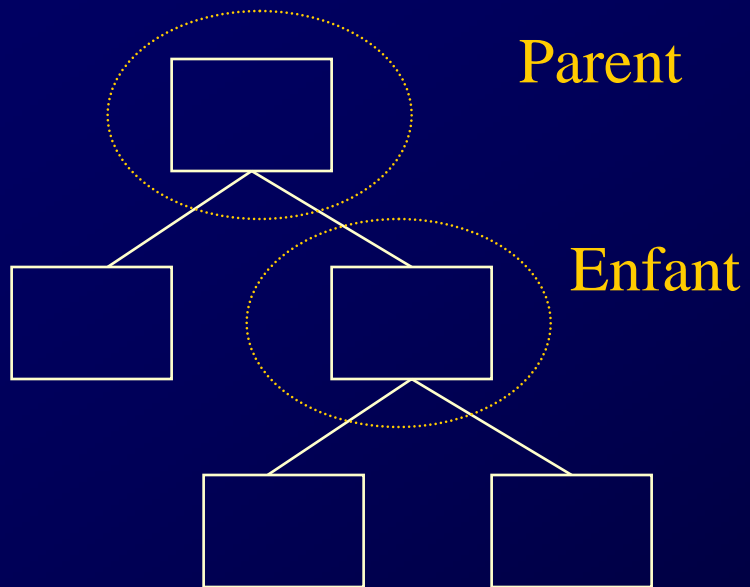
---



- ◆ Classe racine en Java: Class **Object**

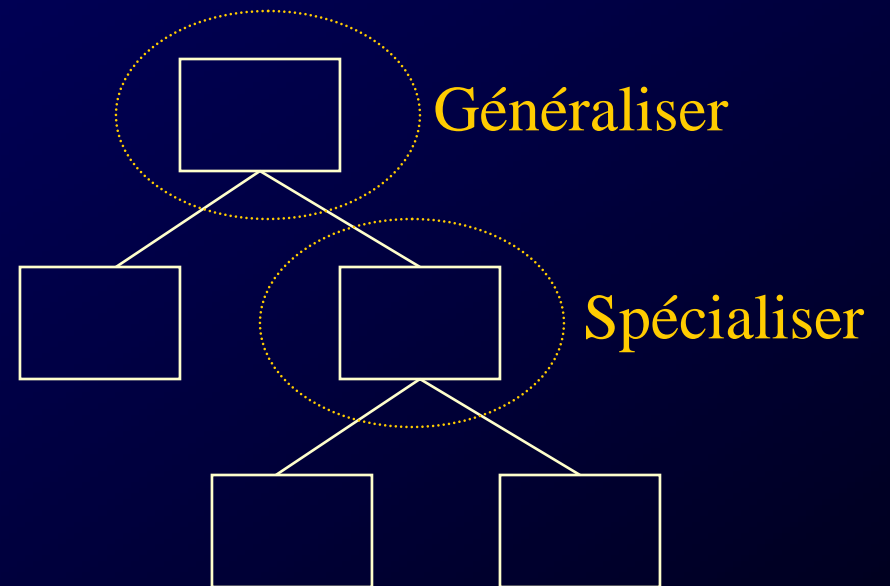
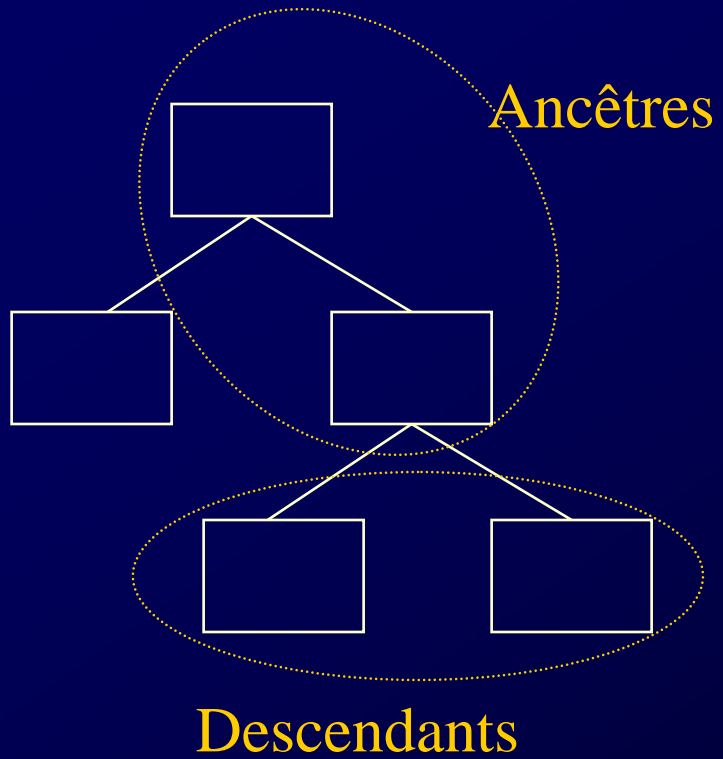
# Classes

---



# Classes

---



# *Sous-classer pour ...*

---

## ◆ ... pour Généraliser

- sous-classe ajoute de la fonctionnalité qui ne respecte pas le sous-typage
  - Faut-il sous-classer CercleNoirEtBlanc par CercleColoré
- À éviter, revoir la hiérarchie des classes

## ◆ ... pour Spécialiser

- is-a s'applique
- *un chien est-un animal*
- rappel: utiliser la composition si ...
  - une pile (stack) est-un tableau ?
  - Non, une pile est implémentée à l'aide d'un tableau



# *Classes Abstraites*

---

- ◆ Une classe abstraite est utilisée pour l'héritage, elle doit être sous-classée
- ◆ On ne peut pas créer d'instances d'une classe abstraite
- ◆ Différence avec la notion d'interface
  - Classe abstraite peut implémenter des méthodes
- ◆ Mot réservé Java: **abstract**

# *Classes Concrètes*

---

- ◆ Classes où aucune méthode n'est abstraite
- ◆ Sous-classes de classes abstraites
- ◆ Instances
  - sont créés à partir des classes concrètes

# Généricité

---

- ◆ Paramétriser des classes

EnsembleDeCercles

Cercle c1,c2, ...;

EnsembleDeCarres

Carre c1,c2, ...;

EnsembleDe[X]

X c1,c2, ...;

Test

EnsembleDe[Cercle] s1;

EnsembleDe[Carre] s2;

# Généricité en Java – Type Générique

```
public interface List <E> {  
    void add(E x);  
    Iterator <E> iterator();  
}
```

```
public class LinkedList <E>  
    implements List <E> {  
    // implementation  
}
```

```
main() {
```

```
    LinkedList<Integer> list = new  
        LinkedList <Integer> ();
```

```
    list.add(new Integer(1));
```

```
}
```

# *Généricité en Java*

---

- ◆ Types Génériques
- ◆ Méthodes Génériques
- ◆ <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>
- ◆ <http://java.sun.com/developer/technicalArticles/J2SE/generics/>

# *Mots Réservés Java*

---

- **abstract**: classe, interface ou méthode
- **extends**: sous-classe extends super-classe
- **implements**: classe implements interface
- **package**: déclaration de package
- **private**: visibilité restreinte à la classe
- **protected**: visibilité restreinte au package et aux sous-classes
- **public**: visible de partout
- **static**: champ existant en un exemplaire unique indépendamment du nombre d'instances de la classe
- **super**: invocation de la super-classe
- **void**: méthode qui n'a pas de valeur de retour
- **(rien)** : visible dans le package

# *Conventions de Programmation*

---

## ◆ Pourquoi?

- Faciliter la maintenance
  - Généralement réalisée par quelqu'un d'autre que l'auteur
- Amélioration de la lecture et de la compréhension du code
- Source code livré est bien organisé

# *Conventions de Programmation*

---

## ◆ Organisation des fichiers

- Commentaires
- Déclaration de package et/ou import
- Classe ou déclaration d'interface
  - Commentaire documentaire
  - Déclaration de la classe ou de l'interface (class Cercle )
  - Commentaire d'implémentation
  - Liste des variables statiques (public, protected, private)
  - Liste des instances (public, protected, private)
  - Constructeurs
  - Méthodes (groupées par fonctionnalité)



# *Conventions de Programmation*

---

## ◆ Noms

- Classes / Interfaces
  - Noms commençant par une Majuscule
  - Chaque Mot intermédiaire commençant également par une Majuscule
    - `class Cercle`
    - `class CercleColoré`
- Méthodes
  - Verbes commençant par une minuscule
  - Chaque Mot intermédiaire commençant par une Majuscule
    - `dessiner()`
    - `dessinerCarré()`

# *Conventions de Programmation*

---

- Variables
  - Commencent par une **minuscule**, mot intermédiaire en **Majuscule**
  - Noms formés d'un caractère sont à utiliser pour les variables temporaires uniquement (entiers: i,j,k,m,n; char:c,d,e)
    - `int i;`
    - `Cercle monCercle;`
- Constantes
  - tout en **MAJUSCULE** et mots séparés par « `_` »
    - `static final int LARGEUR_MIN = 4;`

# Exemple avec les conventions

```
class Carre{
int x, y; float cote; Ecran ec;

public Carre(int a, int b, int h,
             Ecran e)
    {x=a; y=b; cote = h; ec = e;}

public float Surface()
    { return cote*cote;}

public int X() { return x;}
public int Y() { return y;}
}
```

```
class Carre{
int coordX, coordY; float cote;
Ecran ecran;

public Carre(int i, int j, float k,
             Ecran e)
    {coordX=i; coordY=j;
     cote = k; ecran = e;}

public float calculerSurface()
    { return cote*cote;}

public int getX() { return coordX;}
public int getY() { return coordY;}
}
```

# *Conventions de Programmation*

---

- ◆ *... et encore*
  - Commentaires
  - Indentation
  - Déclarations
  - Espaces

Pour en savoir plus:

<http://java.sun.com/docs/codeconv>