

Héritage

Réutilisation de code et Sous-typage

Héritage - Rappel

- ◆ L'héritage permet de définir une classe B à partir d'une classe existante A
 - B est une sous-classe de A
 - Evite la redondance (codes de Cercle et Carre se répètent)
 - Permet d'obtenir des classes qui correspondent à 100% aux besoins
- ◆ B peut comprendre plus de messages que A
- ◆ B peut définir ses propres versions des méthodes qui sont définies dans A

Exemple de Carré et Cercle

- ◆ Les classes Carré et Cercle partagent
 - Un positionnement sur la grille
 - Les variables coordX et coordY
 - Les méthodes getX(), getY(), dessiner(), et bouger()
 - Différence: méthode calculerSurface()
- ◆ Approche programmation objet
 - On définit une classe FormeGéométrique qui contient les variables et les méthodes communes
 - Carré et Cercle héritent de la classe FormeGéométrique
 - Carré et Cercle définissent séparément calculerSurface()

Exemple: classe *FormeGéométrique*

```
class FormeGéométrique{

    int coordX, coordY;
    Ecran ecran;

    public FormeGéométrique(int a, int b;
        Ecran e)
        { coordX=a; coordY=b;ecran=e;}

    public void bouger(int a,int b)
        {coordX = a; coordY = b;}

    /* La méthode calculerSurface() */
    public float calculerSurface()
        {return 0.0;}
```

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

    public int getY()
        { return coordY;}

    public void dessiner()
        { ecran.afficher(this);}
}
```

Exemple: classe Cercle

```
class Cercle extends FormeGéométrique{
```

```
    int rayon;
```

```
    public Cercle(int a, int b, int r, Ecran e){
```

```
        super(a,b,e);
```

```
        rayon = r;
```

```
    }
```

```
    public float calculerSurface()
```

```
    { return 3.14*rayon * rayon;}
```

```
    }
```

les variables ecran, coordX et coordY sont définies implicitement

On invoque le constructeur de la super-classe

Les méthodes bouger(), getX(), getY(), dessiner() sont implicitement définies

Exemple: classe Carre

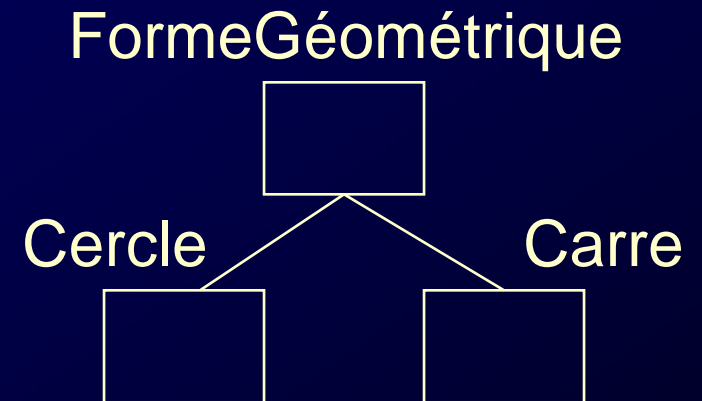
```
class Carre extends FormeGéométrique{

float cote;

public Carre(int a, int b, float r, Ecran e){
    super(a,b,e);
    cote = r;
}

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

}
```



Note : Cercle et Carre redéfinissent la méthode calculerSurface()

Utilisation de l'héritage

- ◆ Regrouper les fonctionnalités communes
 - P.ex: la classe FormeGéométrique regroupe la fonctionnalité commune aux classes Carré et Cercle
 - En Java, toute classe hérite directement ou indirectement de la classe Object
 - **Tous** les objets peuvent invoquer les méthodes publiques et protégées de la classe Object.
 - equals(), getClass(), notify(), wait(), toString(), ...
- ◆ L'héritage est donc un moyen important pour la réutilisation du code et de l'expérience
 - *Outil important de développement*

Utilisation de l'héritage

- ◆ Les spécificités sont décrites dans les sous-classes
 - L'héritage permet la **spécialisation** d'un composant
 - Classe Cercle est une sorte spécialisée de FormeGéométrique
 - L'héritage définit une relation **est-un** entre classes
 - Cercle **est-une** FormeGéométrique
 - On peut donc utiliser un Cercle au lieu d'une FormeGéométrique
 - Voilà pourquoi les objets de la classe Cercle doivent comprendre tous les messages qui sont compris par la classe FormeGéométrique
 - *Outil important d'exécution*

Utilisation de l'héritage

```
Cercle c1 = new Cercle(2,3,4,e);
```

```
Carre c2 = new Carre (3,6,7,e);
```

```
FormeGéométrique f;
```

```
f = c1; f.dessiner();
```

```
System.out.print(«Surface est» +  
    f.calculerSurface());
```

```
f = c2; f.dessiner();
```

```
System.out.print(«Surface est» +  
    f.calculerSurface());
```

Dessine un cercle

Surface du cercle

Dessine un carre

Surface du carre

Utilisation de l'héritage

◆ La relation **est-un**

- spécifie quand un composant peut être utilisé à la place d'un autre

◆ Problème

- Message “f.calculerSurface()” est un appel légal, mais on ne sait pas à la compilation, quel est le code qui sera exécuté:
 - *La méthode calculerSurface() de FormeGéométrique, de Carré ou de Cercle*

Liaisons Statique et Dynamique

◆ Liaison statique

- la liaison entre le message et la méthode se fait sur la base des type *déclaré* de la variable (caractéristiques statiques)
 - *Les librairies en C où le code est recherché à la compilation*

◆ Liaison dynamique:

- la liaison message-méthode s 'effectue sur la base du type de la *valeur* que prend la variable à l 'exécution
 - *La variable f peut référencer un objet de la classe FormeGéométrique, de la classe Carré, ou de la classe Cercle au cours de son existence*

Liaisons Statique et Dynamique

- ◆ La liaison dynamique est la base de l'extensibilité dans la programmation objet et de la maintenance des systèmes
 - Du nouveau code (via des nouvelles classes) peut être introduit dynamiquement sans même arrêter l'application qui tourne
 - Ceci permet de remplacer un composant (p.ex: un objet `FormeGéométrique`) par un autre (p.ex: un objet `Cercle`)

Etendre =

♦ Redéfinir (overriding)

- Prévoir une implémentation différente de la méthode lorsqu'elle s'applique à des instances de la sous-classe
- **Remplacer**: changer complètement le code
 - *calculerSurface()* appliqué à une instance de *FormeGéométrique* retourne 0.0; appliqué à une instance de *Carré* retourne *cote*cote*;
- **Raffiner**: garder le code prévu dans la super-classe et ajouter quelques instructions
 - *utilisation de super(); ... ; (autres instructions)*
 - *Constructeur Carre(a,b,r,e)*

Etendre =

◆ Renommer

- Modifier le nom de la méthode lorsqu 'elle s 'applique aux instances de la sous-classe
 - **rename** en Eiffel
 - n'existe pas en Java

◆ Ajouter

- Sous-classe définit de **nouveaux** attributs, de **nouvelles** méthodes
 - *CercleColoré définit l'attribut Couleur, et méthode getColor()*

◆ Surcharger (overloading)

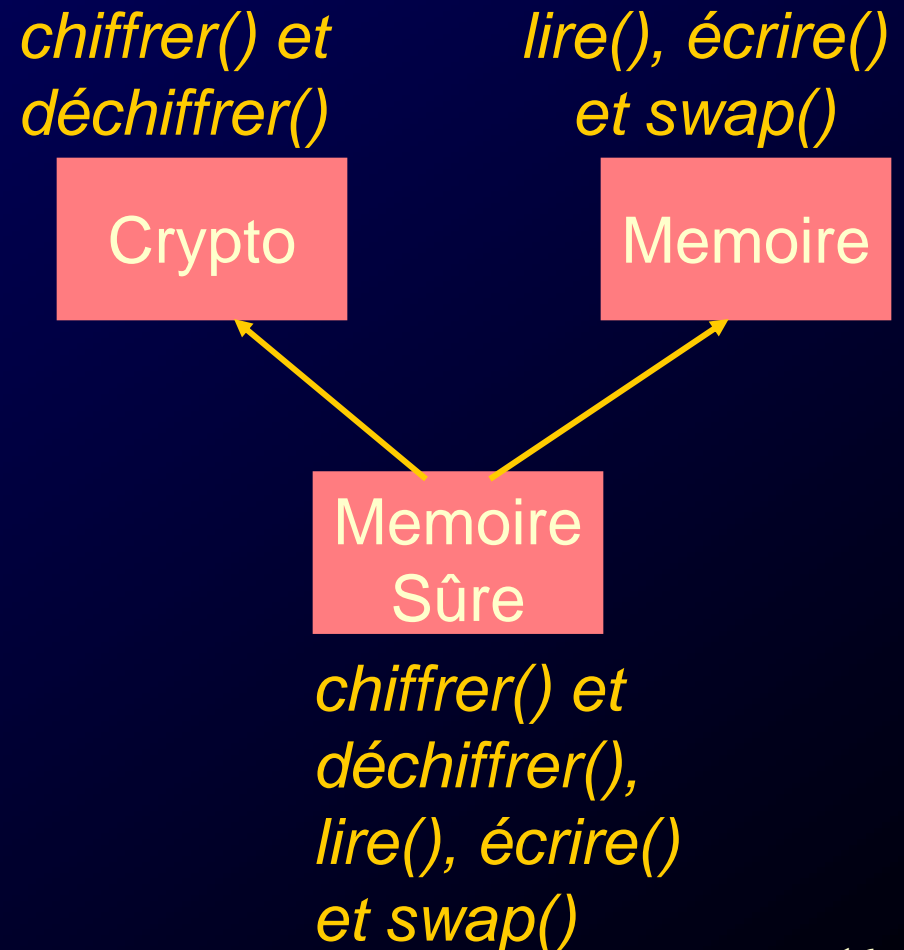
- attacher plus d'un sens à un identificateur (attribut, méthode)
 - même méthode avec des paramètres différents
 - redéfinir la méthode dans une sous-classe

Etendre =

- ◆ Comment la bonne méthode est-elle trouvée?
 - Partir de la classe de l'instance et chercher la méthode
 - Si elle est trouvée: elle est appliquée
 - Sinon, chercher la méthode dans la classe parente
 - Ainsi de suite, jusqu'à ce que la méthode soit trouvée
- ◆ Attention à l'effet yo-yo:
 - rechercher une méthode à l'exécution peut forcer à parcourir **plusieurs fois** la hiérarchie des classes
 - Lorsqu'une méthode est définie à l'aide d'autres méthodes on parcourt à nouveau toutes les classes en partant de celle de l'instance

Héritage multiple

- ◆ Dans quelques langages à objets, une classe peut hériter de plusieurs classes
 - Une classe peut ainsi être définie comme une **spécialisation** de plusieurs classes
 - C++ et Eiffel mettent en œuvre de l'héritage multiple
 - Java: héritage simple
- ◆ La classe **Mémoire-Sûre** représente une page de mémoire qu'on peut chiffrer et déchiffrer



Les 2 buts de l'héritage (multiple)

- ◆ 1. Réutiliser du code de plusieurs sources dans une nouvelle classe
- ◆ 2. Construire une classe qui peut être utilisée à la place de plusieurs classes
 - P.ex: un objet Mémoire-Sûre peut être utilisé à la place d'un objet Crypto, ou bien à la place d'un objet Mémoire

Héritage multiple

- ◆ L'héritage multiple pose des problèmes pour le gestion des noms
- ◆ Variables
 - Supposons que la classe Mémoire et la classe Crypto définissent une variable *r*
 - Selon le principe de l'héritage, toute variable déclarée dans une classe est implicitement déclarée dans une sous-classe
 - Alors, quelle est la signification de *r* dans la classe Mémoire-Sûre?
 - Est-ce que c'est le même objet? Quelle est sa valeur?

Héritage multiple

◆ Méthodes

- Si les deux super-classes définissent la même méthode
- Quelle méthode va être exécutée? Une seule ou les deux?

◆ Solutions:

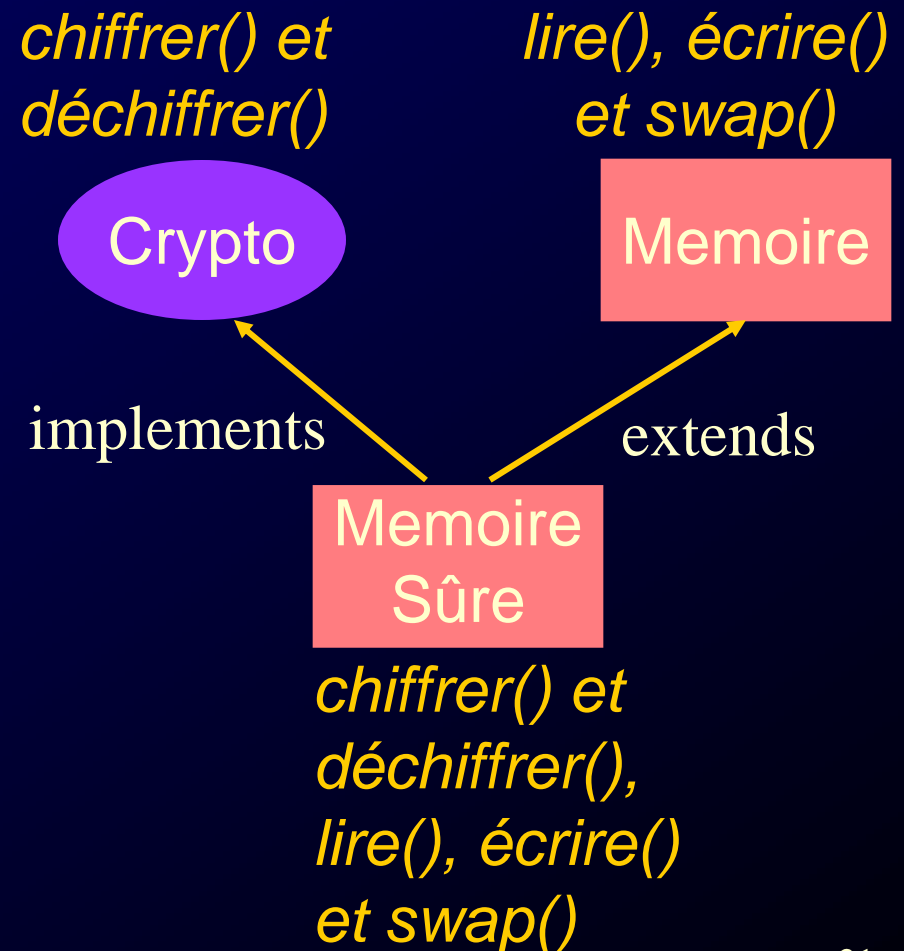
- définir les structures de données le plus bas possible dans la hiérarchie des classes et les encapsuler (privé)
- redéfinir toutes les méthodes héritées à double par les super-classes

Héritage en Java

- ◆ Ces problèmes ont poussé les concepteurs de Java à n'utiliser que de l'héritage **simple**
 - pas de problème de gestion de noms
 - réutilisation de toutes les définitions (non privées) de la super-classe
- ◆ En Java, une classe peut étendre au plus une classe, mais peut implanter plusieurs interfaces

Héritage « Multiple » en Java

- ◆ Définir une interface Crypto
- ◆ Définir une classe Memoire
- ◆ La classe **Mémoire-Sûre** implémente les méthodes de l'interface et représente bien une page de mémoire qu'on peut chiffrer et déchiffrer



La classe Object

- ♦ Java possède une seule hiérarchie des classes
 - Comme Smalltalk, Objective-C (arbre)
 - À la différence de C++ (forêt)
- La classe *Object* se trouve à la racine et implante les méthodes de base
 - P.ex: *equals()*, *hashCode()*, *clone()*, *getClass()*
 - Donc, tout objet comprend un minimum de messages
 - Carre: *c1.clone()*;
 - P.ex: Une classe Chiffrement peut contenir une méthode
 - **public Crypto chiffrer(Object obj)** qui opère sur n'importe quel objet !! De n'importe quelle classe ! Par exemple des carrés ...
 - *o = chiffrement.chiffrer(c1)*;

Coût de l'héritage

- ◆ Un programme objet est généralement plus lent que le même programme écrit en C
 - Principalement à cause de la liaison dynamique de code
 - et de l'effet yo-yo
- ◆ Complexité des programmes :
 - Le comportement attendu d'un programme peut être défait par des sous-classes dynamiquement liées à l'application

Sous-typage et Polymorphisme

*Le rapport entre **int** et **long***

*ou les classes **FormeGéométrique** et **Carré***

Typage - Rappel

- ◆ Chaque classe définit un **type**:
 - valeurs + opérations
- ◆ Chaque sous-classe définit un **sous-type**
 - l'ensemble des valeurs que peut prendre le sous-type est un **sous-ensemble** de l'ensemble des valeurs que peut prendre le super-type
 - *un CercleColoré est un Cercle*
 - *le type CercleColoré est un sous-type du type Cercle*

Sous-typage

- ◆ De manière plus générale
- T1 et T2 deux types
- Si T1 comprend au moins les mêmes opérations que T2
- Ou si T1 comprend moins de valeurs que T2
 - Dans ce cas, T1 est un sous-type de T2 ($T1 \angle T2$)
- *P.ex: [3..7] avec l'opération (==) est une version plus spécialisée de [1..10] avec l'opération (==)*
 - Où [a..b] représente l'ensemble des entiers entre a et b
- *P. ex: [1..10] pour des int comprend moins de valeurs que [1..10] pour des float*

Rôle du sous-typage

◆ 1. Substitution

- Un CercleColoré est un Cercle
 - Les messages envoyés aux cercles sont compris par les objets de CercleColoré
 - Si $U \angle V$, alors on peut utiliser un objet de type U lorsqu'un objet de type V est attendu

◆ 2. Validité

- Programme qui utilise ($==$, $<$) pour des valeurs comprises dans $[1..10]$ marche aussi correctement pour des valeurs dans $[3..7]$

Types Statique et Dynamique

- ♦ V est le type [1..10] (==,<)
U est le type [3..7] (== <)
 - donc, $U \angle V$
- ♦ x possède un type V à la *compilation*
 - V est le type *statique* de x
- ♦ Pendant l'*exécution*, x peut désigner un objet d'un type U
 - U est le type *dynamique* de x

```
Var x: range (1..10) of int;  
Var y: range (3..7) of int;
```

```
Begin
```

```
  x:= y; /* Légal */
```

```
  y:= x; /* illégal */
```

```
End.
```

Le cadre d'objets

- ♦ Si la classe B hérite de la classe A
 - $B \angle A$ puisque B
« comprend les messages
(méthodes) de A »
- ♦ La classe d'une *variable* est appelée la classe **statique**
- ♦ La classe de la *valeur* que prend une variable est appelée la classe **dynamique**

```
Cercle c1 = new Cercle(2,3,4,e);
```

```
/* Classe Statique de f */
```

```
FormeGéométrique f;
```

```
/* Classe Dynamique de f est  
Cercle */
```

```
f = c1;
```

Le cadre d'objets

- ◆ Toute méthode de B doit être un sous-type d'une méthode de A
 - Le *type* d'une *méthode* ?
 - Une méthode (avec un seul paramètre) prend un objet du type S et rend un objet du type D
 - Le type de la méthode est alors la fonction $(S \rightarrow D)$!!

La règle de contra-variance

- ◆ B hérite de A, la variable **a** est de type A, **b** est de type B, la variable **y** est de type D1, la variable **x** est de type S1
 - Est-ce que la commande **y=a.m(x)** valide?
 - Est-ce que la commande **y=b.m(x)** valide?
- ◆ **m** dans A est du type $(S1 \rightarrow D1)$ public D1 m(S1 x);
- ◆ **m** dans B est du type $(S2 \rightarrow D2)$ public D2 m(S2 x);
- ◆ Pour permettre la mise à jour des composants de manière transparente:
 - Une valeur retournée par **m** dans B ($\in D2$) doit être comprise dans les valeurs retournées par **m** dans A
 - $D2 \angle D1$: donc l'affectation à **y** reste légale
 - une valeur acceptée par **m** dans A ($\in S1$) doit être acceptée par **m** dans B
 - $S1 \angle S2$: pour pouvoir appeler **m** (dans B) avec le même **x** (dans A)
- ◆ Si $S1 \angle S2$ et $D2 \angle D1$, alors $(S2 \rightarrow D2) \angle (S1 \rightarrow D1)$

Le cadre d'objets

◆ Si $B \angle A$

- toute méthode de classe A doit posséder un sous-type dans classe B

ColPoint p1;

Point p2,p3;

if(p2.equal(p3)) ..



p2 = p1;



if(p1.equal(p3)) ...



Class Point

```
int x,y;
```

```
equal(Point p)
```

```
    return(p.x ==x &&  
           p.y ==y)
```

Class ColPoint is Point

```
int colour;
```

```
equal(ColPoint c)
```

```
    return (c.x ==x &&  
           c.y ==y &&  
           c.colour==colour)
```


Le cadre d'objets

- ◆ Explication informelle
 - 2eme cas: on cherche la couleur d'un Point !
- ◆ Explication selon la règle de contra-variance
 - $\text{ColPoint} \angle \text{Point}$
 - $\text{equal-p: Point} \rightarrow \text{Bool}$, $\text{equal-c: ColPoint} \rightarrow \text{Bool}$
 - alors, $\text{equal-c} \angle \text{equal-p}$ est faux !
 - Donc, on peut exécuter equal-p avec un ColPoint mais on ne peut pas exécuter equal-c avec un Point !
 - La contra-variance empêche des erreurs dans la définition des nouvelles classes !

Traitement de la contra-variance

◆ C++ et Java

- Le type d'une méthode est sa signature
 - Une sous-classe peut redéfinir le code d'une méthode mais **ne peut pas** changer son type
 - Si on garde le nom mais on change le type ce n'est plus considéré comme la même méthode

◆ Eiffel

- Permet de contourner la contra-variance
 - Plus de flexibilité ; plus d'erreurs

◆ Les règles donnent de la sécurité

Typage et Liaison Dynamique

- ◆ Typage dynamique
 - sélection à l'exécution du type d'une variable
- ◆ Liaison dynamique
 - sélection à l'exécution de la variante de la méthode redéfinie
- ◆ Polymorphisme
 - la possibilité qu'une référence soit associée au moment de l'exécution à des occurrences de différentes classes

Polymorphisme

- ◆ Une variable peut être liée aux objets de différents types au cours de sa vie
 - Variables *polymorphes*
 - Souplesse cruciale dans la programmation orientée-objet
 - Combiner les mérites de réutilisation (héritage) avec une notion de correction (sous-typage)
 - Un programme peut manipuler des types qui ne sont même pas encore inventés
 - Rend typage statique difficile car on n'est pas sûr à l'avance des classes utilisées

2 formes de « polymorphisme »

◆ Polymorphisme par Inclusion

- Polymorphisme via sous-typage de l'héritage
- choix de hiérarchie unique ou hiérarchie multiples

◆ Une hiérarchie unique

- La classe *Object* est la super-classe de toute classe
 - Toute classe comprend donc un minimum de méthodes
 - On peut donc écrire des méthodes qui manipulent n'importe quel type de donnée

– `public void manipuler(Object o) { }`

◆ Les hiérarchies multiples

- Les classes ne possèdent pas une racine unique (p.ex: C++)

2 formes de « polymorphisme »

◆ **Polymorphisme Paramétrique (généricité)**

- Procédures qui prennent des paramètres de plusieurs types
 - Le polymorphisme peut exister à l'exécution
 - Une fonction en mémoire prend des objets de types différents
- Ou seulement à la compilation
 - Une fonction dans une librairie peut être créée en mémoire pour manipuler des objets de divers types

Sommaire

- ◆ Rôle d'un système de types
 - **Génie logiciel**: technique plus formelle que les commentaires
 - **Sécurité** : permet d'éviter des erreurs dans les programmes de façon automatique
 - Le sous-typage réalise un **polymorphisme** qui convient bien à la programmation par extension, cf. Internet !
 - Le polymorphisme est correctement contrôlé

Sommaire

◆ Contraintes

- Moins de flexibilité, surtout avec le typage statique
- La notion d'erreur est restreinte

◆ Mise en œuvre

- Le typage est un aspect crucial de la programmation objet
- Encore un concept qu'un système doit être capable de réaliser
 - Un prochain cours