

---

## Chapitre 3

### Héritage et Polymorphisme

05/12/03

Héritage et polymorphisme

page 1

---

## Redondance de code

- Redondance
  - traitements similaires à des endroits différents du code
  - copier/coller avec adaptations
- Fléau pour le développement et la maintenance d'applications.
- Handicap de la programmation procédurale.

05/12/03

Héritage et polymorphisme

page 5

---

## Plan

- Héritage
- Trans-typage
- Classe Object
- Redéfinition
- Mot-clef « super »
- Polymorphisme
- Mot-clef « final »
- Méthodes et classes abstraites
- Interfaces

05/12/03

Héritage et polymorphisme

page 2

---

## Passage de paramètres

- Le fondement d'un code concis et exploitable est l'introduction du mécanisme de passage de paramètres.
- Il s'agit de pouvoir définir dans le langage des actions (procédures ou fonctions) paramétrées.
- Tous les langages évolués offrent ce mécanisme.

05/12/03

Héritage et polymorphisme

page 6

---

## Héritage

05/12/03

Héritage et polymorphisme

page 3

---

## Composition

- La composition représente un lien de type :  
« has a »
- La composition consiste à utiliser des variables « références » comme champs d'une nouvelle classe.
- La nouvelle classe est cliente des classes des variables « références ».

05/12/03

Héritage et polymorphisme

page 7

---

## Réutilisation de code

- La réutilisation de code est l'une des idées directrices de la programmation.
- Elle est rendue possible grâce aux mécanismes :
  - de passage de paramètres
  - de composition
  - d'héritage
- Le paradigme objet met en œuvre le mécanisme d'héritage contrairement au paradigme procédural.
- Il est alors possible de lutter plus efficacement contre la redondance de code.

05/12/03

Héritage et polymorphisme

page 4

---

## Exemple 1/3

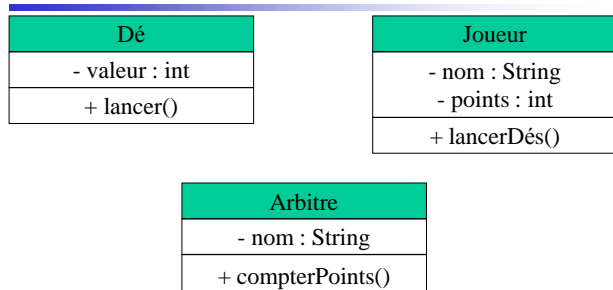
- Un jeu se compose de deux joueurs, trois dés et un arbitre.
- Il faut définir quatre classes :
  - classe Dé
  - classe Joueur
  - classe Arbitre
  - classe Jeu

05/12/03

Héritage et polymorphisme

page 8

## Exemple 2/3



05/12/03

Héritage et polymorphisme

page 9

## Exemple 1/3

```
public class Vehicule extends Object
{
    protected String nom;
    public Vehicule(String n) { nom=n; }
    public String toString()
    {
        return nom;
    }
}
```

05/12/03

Héritage et polymorphisme

page 13

## Exemple 3/3

```
class Jeu
{
    Dé dé1, dé2, dé3;
    Joueur joueur1, joueur2;
    Arbitre arbitre;
    ...
}
```

- La classe jeu est cliente des classes Dé, Joueur et Arbitre.

05/12/03

Héritage et polymorphisme

page 10

## Exemple 2/3

```
public class Bateau extends Vehicule {
    protected boolean plaisance;
    public Bateau(String nom) { this.nom=nom; }
    public Bateau(String nom, boolean b) {
        this(nom); plaisance=b;
    }
    public void quitterPort() { ... }
    public String toString() {
        return nom + (plaisance ? " (plaisance)" : "");
    }
}
```

05/12/03

Héritage et polymorphisme

page 14

## Héritage

- L'héritage représente un lien de type : « is a »
- L'héritage permet de créer une nouvelle classe en spécialisant une classe existante. On dit alors qu'elle hérite de (ou étend) celle-ci.
- La nouvelle classe est dite sous-classe, classe dérivée ou classe enfant. On dit aussi qu'elle est une spécialisation de celle-ci.
- La classe existante est dite super-classe, classe de base ou classe parent. On dit aussi qu'elle est une généralisation de celle-ci.

05/12/03

Héritage et polymorphisme

page 11

## Exemple 3/3

```
public class Avion extends Vehicule {
    protected int nbReacteurs=2;
    public Avion(String nom) { this.nom=nom; }
    public Avion(String nom, int nb) {
        this(nom); nbReacteurs=nb;
    }
    public void decoller() { ... }
    public String toString() {
        return nom + " (" + nbReacteurs + ")";
    }
}
```

05/12/03

Héritage et polymorphisme

page 15

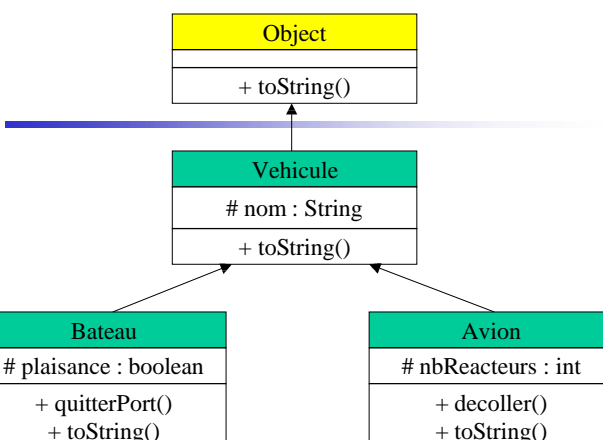
## Héritage simple

- En Java, il n'existe pas d'héritage multiple. On ne dispose que du concept d'héritage simple (et des interfaces).
- Chaque classe hérite d'une seule classe :
  - soit de la classe Object (par défaut)
  - soit d'une classe dont le nom est précisé après le mot-clef « extends ».

05/12/03

Héritage et polymorphisme

page 12



05/12/03

Héritage et polymorphisme

page 16

## Possibilités de l'héritage

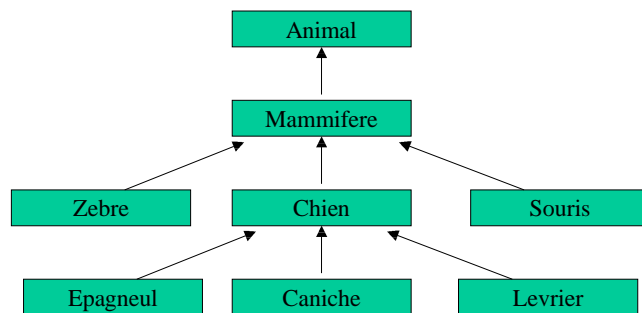
- Possibilité d'étendre la définition d'une classe héritée en ajoutant :
  - des champs
  - des méthodes (incluant des constructeurs).
- Possibilité de redéfinir des méthodes d'une super-classe.

05/12/03

Héritage et polymorphisme

page 17

## Exemple 2



05/12/03

Héritage et polymorphisme

page 21

## Intérêt de l'héritage

- Réutiliser du code
  - sans toucher au code des classes existantes,
  - et en indiquant simplement les nouvelles caractéristiques propres à la sous-classe,
  - et en redéfinissant éventuellement certaines méthodes de la super-classe.

05/12/03

Héritage et polymorphisme

page 18

## Ancêtres et Descendants

- Soit un chemin quelconque dans une hiérarchie de classes menant d'une classe D à une classe A.
  - la classe D est dite classe descendant de A,
  - la classe A est dite classe ancêtre de D.
- Par exemple,
  - VTT est classe descendant de Vehicule
  - Mammifere est classe ancêtre de Epagneul

05/12/03

Héritage et polymorphisme

page 22

## Hiérarchie de classes

- L'héritage ne se limite pas nécessairement à un seul niveau de dérivation.
- Il est possible de définir une hiérarchie de classes sur plusieurs niveaux.
- Une hiérarchie de classes s'appelle également hiérarchie d'héritage.

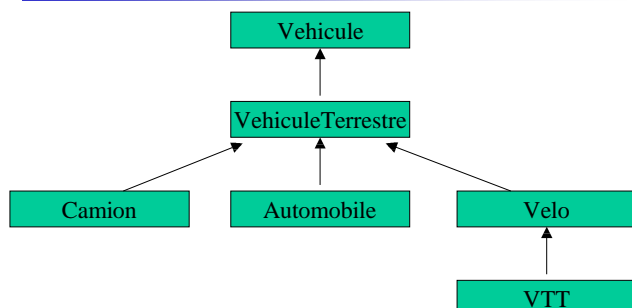
05/12/03

Héritage et polymorphisme

page 19

## Transtypage

## Exemple 1



05/12/03

Héritage et polymorphisme

page 20

## Conversion de classes

- Une variable « référence » déclarée d'une certaine classe peut parfois contenir un lien vers un objet d'une classe différente.
- En effet, il est possible d'effectuer des opérations de conversion (implicite ou explicite) de classes.
- Une telle opération est appelée transtypage.

05/12/03

Héritage et polymorphisme

page 24

## Transtypage (Casting)

- La syntaxe est : (classe) référence
- Si classe est ancêtre de la classe définie pour référence alors il s'agit d'un upcasting.
- Si classe est descendant de la classe définie pour référence alors il s'agit d'un downcasting.
- Dans les autres cas, il y a un problème.

05/12/03


Héritage et polymorphisme

page 25

## Exemple

- La séquence suivante est illégale :  

```
Vehicule v = new Avion("Airbus A320",4);  
...  
v.decoller();
```


- La séquence suivante est légale :  

```
Vehicule v = new Avion("Airbus A320",4);  
...  
((Avion)v).decoller();
```

05/12/03

Héritage et polymorphisme

page 29

## Upcasting

- Cette opération consiste à considérer une référence comme étant plus générale que ce qu'elle semble être.
- Il est toujours possible d'effectuer un upcasting de façon implicite.
- Les champs et méthodes d'une classe sont en effet valides pour toute classe dérivée.

05/12/03

Héritage et polymorphisme

page 26

## Intérêt du upcasting ?

- Sans upcasting, pas besoin de downcasting.
- Par exemple, si lors de la déclaration de v, on ne perdait pas de vue que v est un Avion, alors on pourrait éviter le downcasting.
- MAIS le upcasting est fondamental pour utiliser le polymorphisme.

05/12/03

Héritage et polymorphisme

page 30

## Exemple

L'instruction suivante :

```
Vehicule v = new Avion("Airbus A320",4);
```

est un upcasting implicite.

L'instruction suivante :

```
Vehicule v = (Vehicule) new Avion("Airbus A320",4);
```

est un upcasting explicite équivalent.

05/12/03

Héritage et polymorphisme

page 27

## La classe réelle d'une référence

- Il est toujours possible de tester la classe réelle d'une référence avec "instanceof".
- La syntaxe est : référence instanceof classe
- Il existe une alternative avec la méthode isInstance de la classe Class.

05/12/03

Héritage et polymorphisme

page 31

## Downcasting

- Cette opération consiste à considérer une référence comme étant moins générale que ce qu'elle semble être.
- Un downcasting est nécessairement explicite.
- A l'exécution,
  - un contrôle est effectué pour vérifier la cohérence du downcasting,
  - une exception de type ClassCastException peut être levée.

05/12/03

Héritage et polymorphisme

page 28

## Exemple

```
Vehicule[] vs = new Vehicule[n];  
vs[0] = new Bateau("Remorqueur",false);  
...  
vs[n-1] = new Avion("Airbus A320",4);  
...  
for (int i=0; i<vs.length; i++)  
{  
    if (vs[i] instanceof Avion)  
        ((Avion)vs[i]).decoller();  
}
```

05/12/03

Héritage et polymorphisme

page 32

---

## Classe Object

05/12/03

Héritage et polymorphisme

page 33

---

## equals()

- Une méthode equals() est définie dans la classe Object. Elle renvoie true si l'objet recevant l'appel est identique à celui passé en paramètre de la méthode.
- Initialement, dans Object, cette méthode renvoie true si les deux objets ont la même adresse.
- Pour être utile, elle doit donc être redéfinie.

05/12/03

Héritage et polymorphisme

page 37

---

## classe Object

- En java, la racine de l'arbre d'héritage des classes est la classe java.lang.Object
- Cette classe possède quelques méthodes qui sont donc héritées par toute classe.
- Cette classe ne possède aucun champ.

05/12/03

Héritage et polymorphisme

page 34

---

## Exemple

```
public class Fraction
{
    private int num,den;
    public String toString() { return num+ "/" + den; }
    public boolean equals(Object o)
    {
        if (! (o instanceof Fraction)) return false;
        return num*((Fraction)o).den ==
            den*((Fraction)o).num;
    }
}
```

05/12/03

Héritage et polymorphisme

page 38

---

## toString()

- Une méthode toString() est définie dans la classe Object. Elle renvoie une description de l'objet sous la forme d'une chaîne de caractères (String).
- Initialement, dans Object, cette méthode renvoie l'adresse en mémoire de l'instance.
- Pour être utile, elle doit donc être redéfinie dans les nouvelles classes.

05/12/03

Héritage et polymorphisme

page 35

---

## getClass()

- Une méthode getClass() est définie dans la classe Object. Elle renvoie un objet de la class Class correspondant à la classe réelle de l'objet.
- En effet, en Java, toute classe utilisée dans une application est représentée par un objet instance de la classe java.lang.Class

05/12/03

Héritage et polymorphisme

page 39

---

## Exemple

- Dans une instruction print ou println, toute référence à un objet est automatiquement remplacée par l'appel à toString sur cet objet.
- Par exemple :  
System.out.println("Vehicule = " + v);
- est traduit implicitement en :  
System.out.println("Vehicule = " + v.toString());

05/12/03

Héritage et polymorphisme

page 36

---

## Exemple

```
vehicule[] vs = new Vehicule[n];
vs[0] = new Bateau("Remorqueur",false);
...
vs[n-1] = new Avion("Airbus A320",4);
...
for (int i=0; i<vs.length; i++)
{
    System.out.println(vs[i] + " de classe ");
    System.out.println(vs[i].getClass().getName());
}
```

05/12/03

Héritage et polymorphisme

page 40

## Autres méthodes de Object

- Pour les tables de hachage :
  - hashCode()
- Pour la synchronisation :
  - wait(), notify() et notifyAll()
- Pour le clonage :
  - clone()
- Pour détruire correctement un objet :
  - finalize()

05/12/03

Héritage et polymorphisme

page 41

## Exemple 2/2

- Objet de la classe TestA

```
TestA ta = new TestA();
ta.f();           // OK      => affiche "A"
ta.g();           // OK      => affiche "AA"
ta.h();           // Error
```
- Objet de la classe TestB

```
TestB tb = new TestB();
tb.f();           // OK      => affiche "B"
tb.g();           // OK      => affiche "AA"
tb.h();           // OK      => affiche "BB"
```

05/12/03

Héritage et polymorphisme

page 45

## Redéfinition

05/12/03

Héritage et polymorphisme

page 42

## Contraintes de redéfinition

- Redéfinir une méthode implique certaines contraintes :
  - sur la signature
  - sur l'accessibilité
  - sur les exceptions
- Si ces contraintes ne sont pas respectées, une erreur est signalée à la compilation.

05/12/03

Héritage et polymorphisme

page 46

## Redéfinition

- Lorsqu'on définit une nouvelle classe par héritage, il est possible de redéfinir une (ou plusieurs) méthode.
- Il s'agit de placer dans cette classe une méthode de « même » en-tête qu'une méthode existante dans la super-classe (ou une classe ancêtre).
- La nouvelle méthode « cache » alors la méthode redéfinie.

05/12/03

Héritage et polymorphisme

page 43

## Contraintes sur la signature

- La méthode redéfinie doit posséder exactement la même signature (y compris la valeur retour) que la méthode présente dans la super-classe.
- Si une différence apparaît, alors la nouvelle méthode n'est pas considérée comme une redéfinition mais comme une surcharge.

05/12/03

Héritage et polymorphisme

page 47

## Exemple 1/2

```
class TestA
{
    public void f() { System.out.println("A"); }
    public void g() { System.out.println("AA"); }
}

class TestB extends TestA
{
    public void f() { System.out.println("B"); }
    public void h() { System.out.println("BB"); }
}
```

05/12/03

Héritage et polymorphisme

page 44

## Redéfinition ou Surcharge

- Redéfinition (overriding)
  - Il s'agit d'une méthode qui possède la même signature qu'une méthode de la classe mère.
- surcharge (overloading)
  - Il s'agit d'une méthode qui possède le même nom mais pas la même signature qu'une méthode de la classe (ou d'une super-classe)

05/12/03

Héritage et polymorphisme

page 48

## Exemple

- Ajoutons la méthode suivante à Fraction :

```
public boolean equals(Fraction f)
{
    return num*f.den == den*f.num;
}
```

- La méthode `public boolean equals(Object o)` de Fraction est une redéfinition (overriding) tandis que `public boolean equals(Fraction f)` est une surcharge (overloading).

05/12/03

Héritage et polymorphisme

page 49

## super

- Trois utilisations de "super"
  - pour appeler le constructeur de la classe mère
  - pour appeler une méthode de la classe mère
  - pour accéder à un champ de la classe mère

05/12/03

Héritage et polymorphisme

page 53

## Contraintes sur l'accessibilité

- La méthode redéfinie ne doit jamais être moins accessible que la méthode présente dans la super-classe.
- Dans Avion, il ne serait pas possible de définir `toString()` comme étant private.

05/12/03

Héritage et polymorphisme

page 50

## super pour les constructeurs

- Sur l'exemple des véhicules, on peut constater une certaine redondance dans le code.
- Les constructeurs à un argument des classes Vehicule, Bateau et Avion sont similaires.
- Il est possible d'éviter cette redondance en appelant explicitement le constructeur d'une super-classe.

05/12/03

Héritage et polymorphisme

page 54

## Contraintes sur les exceptions

- La méthode redéfinie ne peut déclarer renvoyer que des sous-classes des exceptions que déclare renvoyer la méthode présente dans la super-classe.

05/12/03

Héritage et polymorphisme

page 51

## 1<sup>ère</sup> instruction d'un constructeur

- Elle peut-être un appel à un constructeur :
  - de la super-classe : `super(...)`
  - de la classe elle-même : `this(...)`
- Dans les autres cas, un appel implicite au constructeur sans argument de la super-classe est effectué.
- Si ce dernier n'existe pas, une erreur apparaît.

05/12/03

Héritage et polymorphisme

page 55

## Mot-clef « super »

## Ordre d'exécution des constructeurs

- La 1<sup>ère</sup> instruction d'un constructeur est l'appel implicite ou explicite (hormis cas du `this`) à un constructeur de la super-classe.
- Et ainsi de suite...
- La toute première instruction exécutée est donc le constructeur sans argument de la classe Object.

05/12/03

Héritage et polymorphisme

page 52

05/12/03

Héritage et polymorphisme

page 56

## Exemple 1/2

- Nouveaux constructeurs de Bateau :

```
public Bateau(String nom) { super(nom); }
public Bateau(String nom, boolean b) {
    super(nom); plaisance=b;
}
```

- Nouveaux constructeurs de Avion :

```
public Avion(String nom) { super(nom); }
public Avion(String nom, int nb) {
    super(nom); nbReacteurs=nb;
}
```

05/12/03

Héritage et polymorphisme

page 57

## Exemple 2/2

```
public class Capitale extends Ville
{
    private String pays;
    ...
    public String toString()
    {
        return super.toString() + " et capitale de "
            + pays;
    }
}
```

05/12/03

Héritage et polymorphisme

page 61

## Exemple 2/2

L'instruction suivante :

```
Avion avion = new Avion("Airbus A320",4);
```

engendre l'exécution suivante :

```
constructeur de Object (sans argument)
constructeur de Vehicule (à 1 argument)
constructeur de Avion (à 2 arguments)
```

05/12/03

Héritage et polymorphisme

page 58

## Limites

- On ne peut remonter plus haut que la classe mère pour appeler une méthode redéfinie
  - pas de : (ClasseAncetre)m()
  - pas de : super.super.m()

05/12/03

Héritage et polymorphisme

page 62

## super pour les méthodes

- Pour appeler une méthode de la classe mère, il suffit d'utiliser la syntaxe suivante :  
super.méthode()
- En fait, on n'utilisera cette syntaxe que si cette méthode est redéfinie dans la classe fille.

05/12/03

Héritage et polymorphisme

page 59

## super pour les champs

- Pour accéder à un champ de la classe mère, il suffit d'utiliser la syntaxe suivante :  
super.champ
- En fait, on n'utilisera cette syntaxe que si ce champ est redéfini dans la classe fille.
- MAIS il faut toujours éviter de redéfinir (cacher) un champ.

05/12/03

Héritage et polymorphisme

page 63

## Exemple 1/2

```
public class Ville
{
    private String nom;
    private int nbHabitants;
    ...
    public String toString()
    {
        return nom+ " ville de " + nbHabitants
            + " habitants";
    }
}
```

05/12/03

Héritage et polymorphisme

page 60

## Exemple 1/2 (à ne pas suivre)

```
class Test
{
    int i=2;
}

class SousTest extends Test
{
    int i;

    SousTest() { i=5+super.i; }
}
```

05/12/03

Héritage et polymorphisme

page 64

## Exemple 2/2 (à ne pas suivre)

```
void display()
{
    System.out.print(" i de SousTest = " + i);
    System.out.print(" i de Test = " + super.i);
}

public static void main(String[] args)
{
    SousTest s = new SousTest();
    s.display();
}
```

05/12/03

Héritage et polymorphisme

page 65

## Polymorphisme

- Le polymorphisme est la capacité qu'offre le langage à exécuter une méthode en fonction de la classe de l'objet en question.
- Le polymorphisme est une alternative élégante à la débauche de tests et de code redondant.

05/12/03

Héritage et polymorphisme

page 69

## Et static ?

- Tout membre static d'une classe donnée est partagé par toutes les classes qui héritent de celle-ci.
- Par exemple, il est possible de compter le nombre d'instances d'une classe donnée et de ses descendants.

05/12/03

Héritage et polymorphisme

page 66

## Problème

- On souhaite écrire le code permettant de définir des figures géométriques telles que des segments, des cercles et des rectangles.
- On souhaite pouvoir dessiner de telles figures et effectuer d'autres opérations telles que l'agrandissement.
- On ne se préoccupera pas de la visibilité (private, ..., public) des classes et membres.
- On suppose donnée une classe Point.

05/12/03

Héritage et polymorphisme

page 70

## Exemple

```
public class Vehicule
{
    protected int id;
    protected String nom;
    private static int nbVehicules;
    public Vehicule(String n)
    {
        id=nbVehicule++;
        nom=n;
    }
    ...
}
```

05/12/03

Héritage et polymorphisme

page 67

## Classe Point

```
class Point
{
    int abscisse;
    int ordonne;

    Point(int abs, int ord) {
        abscisse=abs; ordonne=ord;
    }

    String toString() {
        return "(" + abs + ", " + ord + ")";
    }
}
```

05/12/03

Héritage et polymorphisme

page 71

## Polymorphisme

05/12/03

Héritage et polymorphisme

page 68

## Première tentative 1/3

```
class Figure
{
    final static int SEGMENT = 0;
    final static int CERCLE=1;
    final static int RECTANGLE=2;

    int type; // peut-être SEGMENT, ...
    Point point1;
    Point point2;

    Figure(int t, Point p1, Point p2)
    {
        type=t; point1=p1; point2=p2;
    }
}
```

05/12/03

Héritage et polymorphisme

page 72

## Première tentative 2/3

```
void dessiner()
{
    if (type == SEGMENT) { ... }
    else if (type == CERCLE) { ... }
    else if (type == RECTANGLE) { ... }
}

void agrandir()
{
    if (type == SEGMENT) { ... }
    else if (type == CERCLE) { ... }
    else if (type == RECTANGLE) { ... }
}
}
```

05/12/03

Héritage et polymorphisme

page 73

## Seconde tentative 2/5

```
class Cercle
{
    Point centre;
    int diametre;

    Cercle(Point c, int d)
    {
        centre=c; diametre=d;
    }

    void dessiner() { ... }
    void agrandir() { ... }
}
```

05/12/03

Héritage et polymorphisme

page 77

## Première tentative 3/3

```
class TesterFigures
{
    static void main(String[] args)
    {
        Figure[] figures =
            { new Figure(Figure.CERCLE, ...),
              new Figure(Figure.RECTANGLE, ...),
              new Figure(Figure.SEGMENT, ...),
              new Figure(Figure.CERCLE, ...) };

        for (int i=0; i<figures.length; i++)
            figures[i].dessiner();
    }
}
```

05/12/03

Héritage et polymorphisme

page 74

## Seconde tentative 3/5

```
class Rectangle
{
    Point coinSupGauche;
    Point coinInfDroit;

    Rectangle(Point csg, Point cid)
    {
        coinSupGauche=csg; coinInfDroit=cif;
    }

    void dessiner() { ... }
    void agrandir() { ... }
}
```

05/12/03

Héritage et polymorphisme

page 78

## Commentaires

- Le code correspondant aux différents types de figure est mélangé :-)
- Pour chaque méthode, effectuer les différents tests s'avèrent particulièrement lourd à gérer :-)
- Les champs sont utilisés à des fins diverses (e.g., p1 peut-être le coin supérieur gauche d'un rectangle ou le centre d'un cercle) :-)

05/12/03

Héritage et polymorphisme

page 75

## Seconde tentative 4/5

```
class TesterFigures
{
    static void main(String[] args)
    {
        LinkedList figures = new LinkedList();
        figures.addFirst(new Cercle(...));
        figures.addFirst(new Segment(...));
        figures.addFirst(new Rectangle(...));
        figures.addFirst(new Cercle(...));
    }
}
```

05/12/03

Héritage et polymorphisme

page 79

## Seconde tentative 1/5

```
class Segment
{
    Point extremite1;
    Point extremite2;

    Segment(Point e1, Point e2)
    {
        extremite1=e1; extremite2=e2;
    }

    void dessiner() { ... }
    void agrandir() { ... }
}
```

05/12/03

Héritage et polymorphisme

page 76

## Seconde tentative 5/5

```
Iterator it = figures.iterator();
while (it.hasNext())
{
    Object obj = it.next();
    if (obj instanceof Segment)
        ((Segment)obj).dessiner();
    else if (obj instanceof Cercle)
        ((Cercle)obj).dessiner();
    else ....
}
}
```

05/12/03

Héritage et polymorphisme

page 80

## Remarque

- Il est possible de remplacer le code précédent par :

```
for (int i=0; i<figures.size(); i++)
{
    if (figures.get(i) instanceof Segment)
        ((Segment)figures.get(i)).dessiner();
    else if (figures.get(i) instanceof Cercle)
        ((Cercle)figures.get(i)).dessiner();
    else ....
}
```
- Mais cela est beaucoup moins général et efficace.

05/12/03

Héritage et polymorphisme

page 81

## Commentaires

- Il est possible de placer les différentes figures dans un tableau :-)
- Pour exécuter une méthode, il est nécessaire de tester la classe de l'objet afin de savoir comment downcaster (celui-ci est encore indispensable ici) :-)

05/12/03

Héritage et polymorphisme

page 85

## Commentaires

- Le code correspondant aux différents types de figure est séparé :-)
- Pour exécuter une méthode, il est nécessaire de tester la classe de l'objet afin de savoir comment downcaster (celui-ci est indispensable ici) :-)

05/12/03

Héritage et polymorphisme

page 82

## Quatrième tentative 1/2

```
class Figure
{
    void dessiner() { } // corps de méthode vide
    void agrandir() { } // corps de méthode vide
}

class Segment extends Figure
{ ... } // corps identique à la tentative précédente

class Cercle extends Figure
{ ... } // corps identique à la tentative précédente

class Rectangle extends Figure
{ ... } // corps identique à la tentative précédente
```

05/12/03

Héritage et polymorphisme

page 86

## Troisième tentative 1/2

```
class Figure
{
}

class Segment extends Figure
{ ... } // corps identique à la tentative précédente

class Cercle extends Figure
{ ... } // corps identique à la tentative précédente

class Rectangle extends Figure
{ ... } // corps identique à la tentative précédente
```

05/12/03

Héritage et polymorphisme

page 83

## Quatrième tentative 2/2

```
class TesterFigures
{
    static void main(String[] args)
    {
        Figure[] figures
            = { new Cercle(...), new Segment(...),
              new Rectangle(...), new Cercle(...) };

        for (int i=0; i<figures.length; i++)
            figures[i].dessiner();
    }
}
```

05/12/03

Héritage et polymorphisme

page 87

## Troisième tentative 2/2

```
class TesterFigures
{
    static void main(String[] args)
    {
        Figure[] figures = { new Cercle(...), new Segment(...),
                              new Rectangle(...), new Cercle(...) };

        for (int i=0; i<figures.length; i++)
        {
            if (figures[i] instanceof Segment)
                ((Segment)figures[i]).dessiner();
            else if (figures[i] instanceof Cercle)
                ((Cercle)figures[i]).dessiner();
            else ....
        }
    }
}
```

05/12/03

Héritage et polymorphisme

page 84

## Commentaires

- Grâce au polymorphisme, pour exécuter une méthode, il n'est pas nécessaire de tester la classe de l'objet afin de savoir comment downcaster (celui-ci n'est plus indispensable ici) :-)
- Il est possible de construire un objet de la classe Figure :-)

05/12/03

Héritage et polymorphisme

page 88

## Cinquième tentative 1/2

```
abstract class Figure
{
    abstract void dessiner();
    abstract void agrandir();
}

class Segment extends Figure
{ ... } // corps identique à la tentative précédente

class Cercle extends Figure
{ ... } // corps identique à la tentative précédente

class Rectangle extends Figure
{ ... } // corps identique à la tentative précédente
```

05/12/03

Héritage et polymorphisme

page 89

## Sixième tentative 2/2

```
class TesterFigures
{
    static void main(String[] args)
    {
        Figure[] figures
            = { new Cercle(...), new Segment(...),
              new Rectangle(...), new Cercle(...) };

        for (int i=0; i<figures.length; i++)
            figures[i].dessiner();
    }
}
```

05/12/03

Héritage et polymorphisme

page 93

## Cinquième tentative 2/2

```
class TesterFigures
{
    static void main(String[] args)
    {
        Figure[] figures
            = { new Cercle(...), new Segment(...),
              new Rectangle(...), new Cercle(...) };

        for (int i=0; i<figures.length; i++)
            figures[i].dessiner();
    }
}
```

05/12/03

Héritage et polymorphisme

page 90

## Commentaires

- Comme Figure est une interface, il n'est pas possible de construire un objet de cette classe :-)
- Comme Figure est une interface, il est possible pour les classes Segment, Cercle et Rectangle d'hériter d'une autre classe :-)

05/12/03

Héritage et polymorphisme

page 94

## Commentaires

- Comme Figure est déclarée abstraite, il n'est plus possible de construire un objet de cette classe :-)
- Java utilisant l'héritage simple, il n'est pas possible pour les classes Segment, Cercle et Rectangle d'hériter d'une autre classe :-)

05/12/03

Héritage et polymorphisme

page 91

## Retour sur le polymorphisme

- Le polymorphisme est le fait qu'une même écriture puisse correspondre à différents appels de méthodes.
- Ce concept est une notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage.

05/12/03

Héritage et polymorphisme

page 95

## Sixième tentative 1/2

```
interface Figure
{
    void dessiner();
    void agrandir();
}

class Segment implements Figure
{ ... } // corps identique à la tentative précédente

class Cercle implements Figure
{ ... } // corps identique à la tentative précédente

class Rectangle implements Figure
{ ... } // corps identique à la tentative précédente
```

05/12/03

Héritage et polymorphisme

page 92

## Exemple de polymorphisme

- L'instruction figures[i].dessiner() :
  - correspondra à l'exécution de la méthode dessiner() de la classe Segment si figures[i] est un objet de cette classe,
  - correspondra à l'exécution de la méthode dessiner() de la classe Cercle si figures[i] est un objet de cette classe,
  - ...

05/12/03

Héritage et polymorphisme

page 96

## Liaison dynamique

- Le polymorphisme est obtenu grâce au mécanisme de liaison dynamique (ou retardée).
- Lorsqu'un objet reçoit un message, la méthode qui est exécutée est déterminée :
  - au moment de l'exécution (et non celui de la compilation)
  - par la classe de cet objet (et non celle de la variable référence désignant cet objet)

05/12/03

Héritage et polymorphisme

page 97

## Intérêt du polymorphisme

- Le polymorphisme évite d'écrire du code utilisant :
  - de nombreux embranchements (switch ou if ... else if ...)
  - des pointeurs de fonction
- Le code est alors plus clair et plus succinct.

05/12/03

Héritage et polymorphisme

page 101

## Liaison dynamique

- Soit l'appel suivant : `o.m()`;
- Si la classe C de l'objet référencé par o (et non la classe de la variable référence o) contient la définition de la méthode m() alors celle-ci est exécutée, sinon, la méthode m() est recherchée dans la classe mère de C, puis, si elle ne s'y trouve pas, dans la classe mère de cette classe mère, et ainsi de suite ...

05/12/03

Héritage et polymorphisme

page 98

## Intérêt du polymorphisme

- Le polymorphisme facilite l'extension : on peut créer de nouvelles sous-classes dont les instances pourront être prises en compte sans toucher aux autres classes.
- Par exemple, il est possible de créer une sous-classe Losange sans toucher au reste.

05/12/03

Héritage et polymorphisme

page 102

## Liaison dynamique ou statique

- Liaison dynamique  
dynamic binding, late binding, run-time binding
- Liaison statique  
static binding, early binding, compile-time binding
- Il est à noter que le mécanisme de liaison dynamique peut ralentir sensiblement l'exécution des programmes

05/12/03

Héritage et polymorphisme

page 99

## Mot-clef « final »

## final

- `System.out.println(o)` affiche une description de tout objet o grâce au polymorphisme
- Elle est en effet équivalente à `System.out.println(o.toString())` et grâce au polymorphisme c'est la méthode `toString()` de la classe de o qui est exécutée.

- Le mot-clef final peut être utilisé :
  - au niveau d'une méthode
  - au niveau d'une classe
  - au niveau d'une variable
  - au niveau d'un paramètre
- Il indique "ceci ne peut être changé".
- Son utilisation peut être motivée pour des raisons :
  - conceptuelles et/ou
  - pragmatiques

05/12/03

Héritage et polymorphisme

page 100

05/12/03

Héritage et polymorphisme

page 104

## final pour une méthode

- Il existe deux motivations pour déclarer final une méthode :
  - éviter que celle-ci ne puisse être redéfinie dans une sous-classe (motivation conceptuelle)
  - améliorer l'efficacité lors de l'appel à cette méthode (motivation pragmatique)
- Toute méthode privée est implicitement final.

05/12/03

Héritage et polymorphisme

page 105

## Exemple

```
final class A
{
    ...
}

class B extends A
{
    ...
}
```



ERREUR  
Impossible  
d'hériter de A

05/12/03

Héritage et polymorphisme

page 109

## Exemple

```
class A
{
    final void f() { ... }
}
```

```
class B extends A
{
    void f() { ... }
}
```



ERREUR  
Impossible  
de redéfinir f() de A

05/12/03

Héritage et polymorphisme

page 106

## final pour une variable primitive

- La valeur de la variable ne peut être changée après initialisation.
- L'initialisation de la "constante" peut s'effectuer :
  - à la compilation (compile-time constant)
  - à l'exécution (run-time constant)

05/12/03

Héritage et polymorphisme

page 110

## Efficacité

- La liaison dynamique nécessite la recherche des méthodes lors de l'exécution.
- Le compilateur peut éviter la liaison dynamique pour les méthodes final et améliorer ainsi les performances.
- Pour des raisons d'efficacité, on ne se permettra de déclarer final une méthode que de façon exceptionnelle.

05/12/03

Héritage et polymorphisme

page 107

## compile-time constant

- Il s'agit souvent de constantes de classes déclarées publiques.
- Par exemple,

```
public static final NORD=0;
public static final SUD=1;
```

05/12/03

Héritage et polymorphisme

page 111

## final pour une classe

- Déclarer final une classe empêche la possibilité d'hériter à partir de celle-ci.
- Toutes les méthodes d'une classe final sont implicitement final.
- Par exemple, les classes String et Vector sont déclarées final.

05/12/03

Héritage et polymorphisme

page 108

## run-time constant

- La valeur de la constante ne peut être calculée qu'à l'exécution.
- Par exemple,

```
final int i = (int)(Math.random()*20);
final int j = i/2;
```
- Des constantes (blank finals) peuvent également être initialisées via les constructeurs.

05/12/03

Héritage et polymorphisme

page 112

## blank finals

- Un champ déclaré final mais sans initialisation directe doit être initialisé dans chaque constructeur.

- Par exemple :

```
class Essai
{
    final int x;
    Essai(int xx) { x=xx; }
    ...
}
```

05/12/03

Héritage et polymorphisme

page 113

## abstract

- Le mot-clef « abstract » peut être utilisé :
  - au niveau d'une méthode
  - au niveau d'une classe
- Il indique "ceci sera précisé plus tard"

05/12/03

Héritage et polymorphisme

page 117

## final pour une variable référence

- Une variable référence final ne peut référencer un autre objet que celui qui lui est affecté initialement.
- Cependant, l'état de cet objet peut être modifié.

05/12/03

Héritage et polymorphisme

page 114

## Méthodes abstraites

- Une méthode abstraite :
  - est précédé du mot-clef abstract
  - ne possède pas de corps
- Toute méthode abstraite doit être implémentée dans une sous-classe (à moins que celle-ci ne soit elle-même abstraite).
- Une méthode static ne peut être abstraite.

05/12/03

Héritage et polymorphisme

page 118

## Exemple

```
StringBuffer s1 = new StringBuffer("le
boute du sultan remonte le confluent de la
garonne");
final StringBuffer s2 = s1;
StringBuffer s3 = new StringBuffer("hum");

s2=s3; // ERREUR
s3=s2;
Char tmp = s2.charAt(47);
s2.setCharAt(47,s2.charAt(3)); //OK
...
```

05/12/03

Héritage et polymorphisme

page 115

## Classes abstraites

- Toute classe comportant une méthode abstraite doit être déclarée abstraite.
- Toute classe ne comportant aucune méthode abstraite peut être déclarée abstraite.
- Dans tous les cas, il n'est pas possible de créer une instance d'une classe abstraite.

05/12/03

Héritage et polymorphisme

page 119

## Méthodes et classes abstraites

## Intérêt des classes abstraites

- Une classe abstraite peut être utilisée pour :
  - Structurer l'ensemble des classes
  - Factoriser du code
  - Permettre le polymorphisme

05/12/03

Héritage et polymorphisme

page 116

05/12/03

Héritage et polymorphisme

page 120

## Interfaces

05/12/03

Héritage et polymorphisme

page 121

## Implémenter une interface

- En java, pour indiquer qu'une classe B implémente l'interface I, on écrit :  
class B implements I
- Dans la classe B, on doit trouver l'implémentation ou corps de toutes les méthodes de l'interface I (sauf si B est abstraite).
- Si, par ailleurs, la classe B étend la classe A, alors on écrit :  
class B extends A implements I

05/12/03

Héritage et polymorphisme

page 125

## Interface

- Une interface représente en quelque sorte la notion de classe « purement » abstraite.
- Une interface ne peut contenir que :
  - des méthodes qui sont toujours par défaut « public abstract ».
  - des champs qui sont toujours par défaut « public static final ».

05/12/03

Héritage et polymorphisme

page 122

## Implémenter plusieurs interfaces

- En java, implémenter plusieurs interfaces ne pose aucun problème.
- Les méthodes qui doivent être implémentées sont celles qui apparaissent dans chaque interface.
- Les constantes qui sont « héritées » sont celles qui apparaissent dans chaque interface.

05/12/03

Héritage et polymorphisme

page 126

## Exemple 1

```
interface Figure
{
    void dessiner();
    void agrandir();
}
```



Implicitement,  
chaque méthode est  
public abstract

05/12/03

Héritage et polymorphisme

page 123

## Implémenter plusieurs interfaces

- En java, pour indiquer qu'une classe B implémente les interfaces  $I_1, I_2, \dots, I_n$ , on écrit :  
class B implements  $I_1, I_2, \dots, I_n$
- Si, par ailleurs, la classe A étend la classe A, alors on écrit :  
class B extends A implements  $I_1, I_2, \dots, I_n$

05/12/03

Héritage et polymorphisme

page 127

## Exemple 2

```
interface Direction
{
    int NORD = 0;
    int SUD = 1;
    int EST = 2;
    int Ouest = 3;
}
```



Implicitement,  
chaque champ est  
public final static  
(donc une constante)

05/12/03

Héritage et polymorphisme

page 124

## Intérêt des interfaces

- En java, toute classe peut donc :
  - étendre une seule classe
  - implémenter plusieurs interfaces
- Il est donc préférable de définir une interface plutôt qu'une classe abstraite chaque fois que cela est possible car implémenter une interface n'implique aucune restriction.
- Les interfaces permettent d'une certaine manière de réaliser un héritage multiple « limité ».

05/12/03

Héritage et polymorphisme

page 128

## Héritage multiple

- L'héritage multiple consiste à pouvoir étendre simultanément plusieurs classes.
- Des problèmes se présentent dans la gestion d'un héritage multiple « non limité »
- Une telle gestion peut rendre le compilateur
  - complexe comme en C++
  - peu performant comme en Eiffel

05/12/03

Héritage et polymorphisme

page 129

## Interfaces et transtypage

- Il suffit de penser qu'une interface est une classe abstraite particulière pour gérer le transtypage.

- Par exemple :

Figure f = new Rectangle(...);

...  
Rectangle r = (Rectangle)f;

Upcasting

Downcasting

05/12/03

Héritage et polymorphisme

page 133

## Héritage multiple « limité »

- En Java, l'héritage multiple est « limité » à l'extension d'une classe et l'implémentation de plusieurs interfaces.
- De cette façon, hériter :
  - de constantes de même nom
  - de méthodes de même signaturene pose aucun problème.

05/12/03

Héritage et polymorphisme

page 130

## Problème

- Imaginons que l'on souhaite écrire un algorithme de tri qui puisse être utilisé par tout type d'objet.
- On se rend compte que le code se compose :
  - d'une partie invariable (le mécanisme du tri)
  - d'une partie variable (la comparaison de 2 objets)

05/12/03

Héritage et polymorphisme

page 134

## Hériter de constantes de même nom

- Si plusieurs constantes (champs static final) homonymes sont héritées alors pour lever l'ambiguïté, il est nécessaire de préfixer le nom de la constante par le nom de la classe ou de l'interface.
- Une erreur à la compilation apparaît si cela n'est pas effectué.

05/12/03

Héritage et polymorphisme

page 131

## Solution

- La partie invariable peut être codée sous la forme d'une classe Sort offrant des méthodes de tri permettant de trier un tableau d'objets comparables entre eux.
- La partie variable peut être codée sous la forme d'une interface Comparable offrant une méthode permettant de comparer (ordonner) 2 objets.

05/12/03

Héritage et polymorphisme

page 135

## Hériter de méthodes de même signature

- Si plusieurs méthodes de même signature, sont héritées, alors aucune ambiguïté n'apparaît car au plus l'une d'entre elles (provenant d'une classe et non d'une interface) possède une implémentation.
- Si aucune ne possède d'implémentation, alors il est nécessaire de lui en fournir une (sauf si la classe que l'on définit est abstraite).

05/12/03

Héritage et polymorphisme

page 132

## Interface Comparable

- Il suffit donc d'introduire une interface comme suit :

```
interface Comparable
{
    int compareTo(Object o);
}
```

- Il est à noter que cette interface existe déjà en fait dans le SDK.

05/12/03

Héritage et polymorphisme

page 136

## Méthode compareTo

- La méthode compareTo s'utilise entre 2 objets comparables comme suit :  
obj1.compareTo(obj2)
- Cette méthode doit retourner :
  - 0 si obj1 et obj2 sont de même valeur,
  - une valeur négative si obj1 plus petit que obj2
  - une valeur positive si obj1 plus grand que obj2

05/12/03

Héritage et polymorphisme

page 137

## Trier des objets comparables

- On peut écrire une méthode de tri comme suit :

```
public static void triBulles(Comparable[] t) {
    for (int limit=t.length-1; limit>0; limit--) {
        for (int i=0; i<limit; i++) {
            if (t[i].compareTo(t[i+1]) > 0) {
                Comparable tmp = t[i];
                t[i]=t[i+1];
                t[i+1]=tmp;
            }
        }
    }
}
```

- Il est à noter qu'il existe une méthode de tri (fusion) qui s'appelle sort dans la classe Arrays.

05/12/03

Héritage et polymorphisme

page 138

## Comparer 2 rationnels

Class Rationnel implements Comparable {

```
...
public int compareTo(Object o) {
    Rationnel r = (Rationnel)o;
    double rapport1 = ((double)num)/den;
    double rapport2 = ((double)r.num)/r.den;
    if (rapport1 == rapport2) return 0;
    if (rapport1 < rapport2) return -1;
    return +1;
}
}
```

05/12/03

Héritage et polymorphisme

page 139

## Choisir entre une interface et une classe abstraite

- Une interface représente un ensemble de fonctionnalités qu'une classe implémentant celle-ci doit offrir.
- Une classe (abstraite) représente la description d'un objet qu'il est possible de créer.
- Chaque fois que cela est possible (et naturel), on choisira de définir une interface plutôt qu'une classe.

05/12/03

Héritage et polymorphisme

page 140