UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Programmation Orientée Objet

Initiation à Java

Saber HENI
saber.heni02@univ-paris8.fr
http://handiman.univ-paris8..fr/~saber/

ORACLE
Certified Professional
Java SE 6 Programmer

Plan du cours

- Chapitre 1 : Introduction au langage Java
- Chapitre 2 : Classes et objets
- Chapitre 3 : Concepts de base de l'OO
- Chapitre 4 : La gestion des exceptions
- Chapitre 5 : Les classes de base dans Java
- Chapitre 6 : Les collections

UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Chapitre 3

Concepts de base de l'Orienté Objet

L'héritage - 1

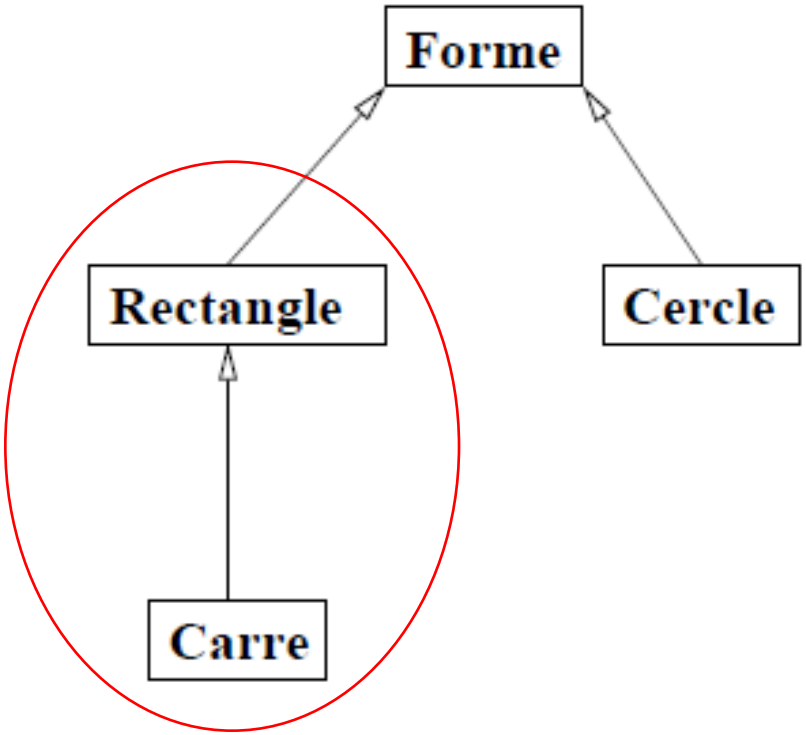
- La notion d'héritage est l'un des fondements de la programmation orientée objet.
- Grâce à elle, nous pourrons créer des **classes héritées** (aussi appelées ***classes dérivées*** ou encore ***classes filles***) de nos **classes mères** (aussi appelées ***classes de base*** ou encore ***super classes***).
- Nous pourrons créer autant de classes dérivées, par rapport à notre classe de base, que nous le souhaitons.
- De plus, nous pourrons nous servir d'une classe dérivée comme d'une classe de base pour élaborer encore une autre classe dérivée.

L'héritage - 2

- Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre.
- Les sous-classes peuvent **redéfinir** les méthodes héritées.
  - Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.
- Une classe peut avoir plusieurs sous-classes. Une classe ne peut avoir qu'une seule classe mère.
  - **Il n'y a pas d'héritage multiple en Java.**

L'héritage - 4

- Avant de passer à un exemple, on dit que en Java que tout est objet.
  - Bien évidement parce que toute les classes héritent implicitement (directement ou par héritages successifs) de la classe **Object** qui est la classe mère de toutes les classes.



L'héritage - 5

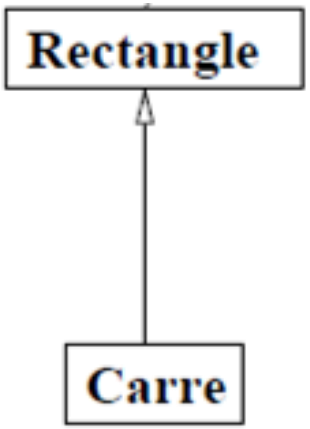
Exemple

Nous allons reprendre l'exemple du rectangle et créer la nouvelle classe carré.

- On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre.
  - En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.
- Après, on écrit le constructeur de la classe carré.

```
public class Carre extends Rectangle {

        public Carre (int cote) {
                super (cote, cote);
        }
}
```



L'héritage - 6

Nous allons ajouter une méthode à la classe Rectangle.

```
public void afficher () {
        System.out.println(”Rectangle ” + longueur + ”x” + largeur);
}
```

Nous rappelons la classe Rectangle.

```
public class Rectangle {
        private int longueur ;
        private int largeur ;

        public Rectangle (int longueur , int largeur ) {
                System.out.println ("Appel du constructeur de la classe
        "       +"Rectangle");
                this.longueur = longueur;
                this.largeur = largeur ;
        }
        public int surface () {
                return this.longueur * this.largeur ;
        }
        // ici les accesseurs et les mutateurs
}
```

L'héritage - 7

- Pour appeler le constructeur de la classe mère, il suffit d'écrire **super(*paramètres*)** avec les paramètres.
  - En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.
- Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par **super**.

```
public class Carre extends Rectangle {

        public Carre (int cote) {
                super (cote, cote);
                System.out.println ("Appel du constructeur de la classe "
                +"Carre");
                System.out.println(super.surface ());
        }
}
```

L'héritage - 8

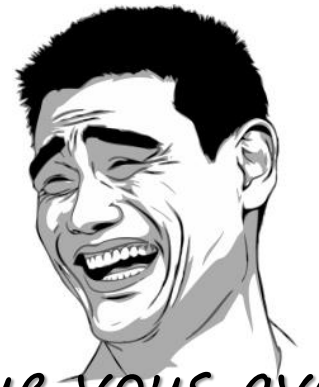
- La classe Carre peut bénéficier de la classe Rectangle et ne nécessite pas la réécriture de ces méthodes si celles-ci conviennent à la sous-classe.

- L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours en tant que première instruction ;
- Si aucun appel à un constructeur d'une classe supérieure n'est fait, le constructeur fait appel implicitement à un constructeur vide de la classe supérieure (comme si la ligne super() était présente).
- Si aucun constructeur vide n'est accessible dans la classe supérieure, une erreur se produit lors de la compilation.

L'héritage - 9

- **Exemple**

```
public class Carre extends Rectangle {
        private int cote;
        public Carre (int cote) {
                this.cote = cote;
        }
}
```



Parce que vous avez oublié ce qui est écrit sur le panneau rouge de la diapo précédente

Accès aux attributs hérités - 1

- Les variables et méthodes définies avec le modificateur d'accès **public** restent publiques à travers l'héritage et toutes les autres classes peuvent y accéder.
- Une variable d'instance définie avec le modificateur **private** n'est pas accessible directement, seulement *via* les méthodes héritées.
- Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur **protected**.
  - La variable ainsi définie sera héritée dans toutes les classes filles qui pourront y accéder librement mais ne sera pas directement accessible hors de ces classes.

Accès aux attributs hérités - 2

- Ci-dessous un tableau récapitulant la porté (Visibilité) d'un membre d'une classe marqué à chaque fois par l'un des quatre modificateurs de visibilité.

	public	protected	défaut	private
Dans la même classe	Oui	Oui	Oui	Oui
Dans une classe du même package	Oui	Oui	Oui	Non
Dans une sous-classe d'un autre package	Oui	Oui	Non	Non
Dans une classe quelconque d'un autre package	Oui	Non	Non	Non

Accès aux attributs hérités - 3

- **Exemple**

```
public class Parent {
	private int w = 1;
	public int x = 2;
	int y = 3;
	protected int z = 4;
}
```

```
public class Fille extends Parent {
	public void testerVisibilité () {
		System.out.println(w);
		System.out.println(x);
		System.out.println(y);
		System.out.println(z);
	}
}
```

Accès aux méthodes héritées - 1

- Pour la visibilité des méthodes héritées.
  - Même principe que les attributs.
- Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par super.
  - **Exemple :**

```
public class Carre extends Rectangle {

        public Carre (int cote) {
                super (cote, cote);
                System.out.println(super.surface ());
                super.afficher ();

        }
}
```

Accès aux méthodes héritées - 2

- **Suite de l'exemple :**

```
public class Main {

        public static void main (String [] args) {
                Carre c1 = new Carre (4);
        }
}
```

- **Résultat :**

1

```
Appel du constructeur de la classe Carre
Appel du constructeur de la classe Rectangle
16
Rectangle 4x4
```

Faux

2

```
Appel du constructeur de la classe Rectangle
Appel du constructeur de la classe Carre
16
Rectangle 4x4
```

Correct

Accès aux méthodes héritées - 3

```
Appel du constructeur de la classe Rectangle
Appel du constructeur de la classe Carre
16
Rectangle 4x4
```

- La méthode afficher écrit le mot "Rectangle" en début de chaîne. Il serait souhaitable que ce soit "carré" qui s'affiche.
- Le problème peut être résolu par une redéfinition de la méthode afficher dans la classe Carre.
  - **Redéfinition des méthodes?**

Redéfinition des méthodes héritées - 1

- On dit qu'une méthode d'une sous-classe redéfinit ((en) **override**) une méthode de sa classe supérieure, si elles ont la même signature mais que le traitement effectué est ré-écrit dans la sous-classe.
  - Voici le code de la classe Carre où est résolu le problème soulevé

```
public class Carre extends Rectangle {

        public Carre (int cote) {
                super (cote, cote);
                System.out.println(super.surface ());
                this.afficher ();

        }

        public void afficher () {
                System.out.println("carre " +
                        this.getLongueur());
        }
}
```

Redéfinition des méthodes héritées - 2

- Lors de la redéfinition d'une méthode, il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure.
  - Cet accès utilise également le mot-clé **super** comme préfixe à la méthode.
  - Dans notre cas, il faudrait écrire super.afficher() pour effectuer le traitement de la méthode afficher() de la classe Rectangle.
- Enfin, il est possible d'interdire la redéfinition d'une méthode ou d'une variable en introduisant le mot-clé **final** au début d'une signature de méthode.
- Il est aussi possible d'interdire l'héritage d'une classe en utilisant **final** au début de la déclaration d'une classe (avant le mot-clé class).

Redéfinition des méthodes héritées - 3

- **Exemple 1**

```
public final class Rectangle {
        private int longueur ;
        private int largeur ;
        public Rectangle (int longueur , int largeur ) {
                this.longueur = longueur;
                this.largeur = largeur ;
        }
        public int surface () {
                return this.longueur * this.largeur ;
        }
}
```

```
public class Carre extends Rectangle { }
```

- **Résultat de compilations :**
  - La classe Rectangle compilera correctement.
  - La classe Carre affichera un message : **error : cannot inherit from final Rectangle**

Redéfinition des méthodes héritées - 4

- **Exemple 2**

```
public class Rectangle {
        private int longueur ;
        private int largeur ;
        // Le constructeur et les accesseurs sont cachés ici
        public final void afficher () {
          System.out.println("Rectangle " + longueur + "x" + largeur);
        }
}
```

```
public class Carre extends Rectangle {
        // Le constructeur est caché ici
        public void afficher () {
                System.out.println("carre " + this.getLongueur());
        }
}
```

- **Résultat de compilations :**
  - La classe Rectangle compilera correctement.
  - La classe Carre affichera un message : **error : afficher () in Carre cannot override afficher() in Rectangle … overridden method is final**

Redéfinition des méthodes héritées - 5

- **Règles de redéfinition (Overriding) des méthodes :**
  - Une méthode privée (non héritée) ne peut pas être redéfinit.
    - Par contre, elle peut être ré-déclarée dans la sous-classe.
  - Une méthode finale ne peut pas être redéfinit non plus.
  - Un constructeur ne peut pas être hérité et ne peut ni être marqué statique ni final.
  - Pour redéfinir une méthode, il faut qu'elle ait le même nom et exactement les même paramètres que la méthode de la classe mère.
    - Si le nombre et/ou le type de paramètre change, vous tomber involontairement dans cas de la surcharge (Overloading).
  - La type de retour de la méthode redéfinie peut être le même que la méthode originale ou un sous-type (Retour covariant (plus de détail dans le Polymorphisme)).
  - La visibilité de la méthode redéfinie ne doit pas être plus restrictive que celle de la classe mère.
  - Une méthode de classe (Statique) ne peut pas être redéfinit.
    - Elle aussi peut être ré-déclarée dans la sous-classe. (plus de détail dans le Polymorphisme)

Polymorphisme - 1

- Le polymorphisme est la faculté attribuée à un objet d'être une instance de plusieurs classes.
- Il a une seule classe "réelle" qui est celle dont le constructeur a été appelé en premier (c'est-à-dire la classe figurant après le new).
  - Mais il peut aussi être déclaré avec une classe supérieure à sa classe réelle.
  - Cette propriété est très utile pour la création d'ensembles regroupant des objets de classes différentes comme dans l'exemple suivant :

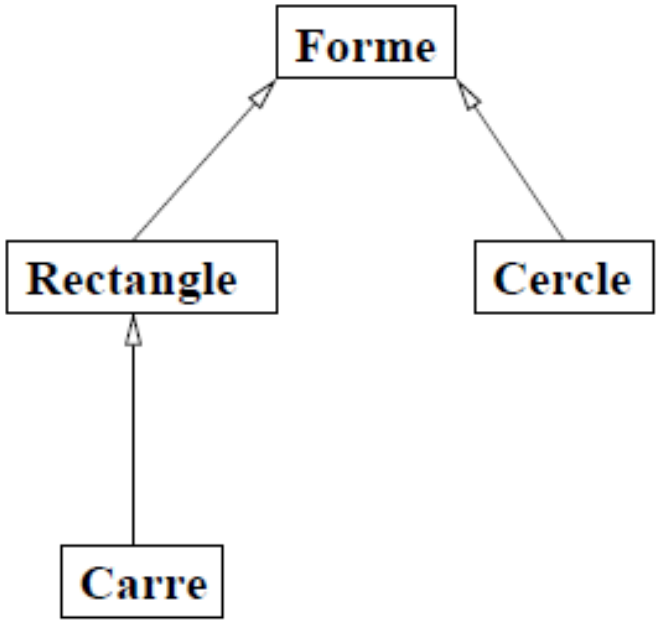
```
Forme[] tableau = new Forme[4];
tableau[0] = new Rectangle(10,20);
tableau[1] = new Rectangle(5,30);
tableau[2] = new Carre(10);
tableau[3] = new Forme();
```

Polymorphisme - 2

- Soit la classe Forme qui est la classe mère de Rectangle.

```
Public class Forme {
    public int surface () {
      return 0;
    }
    public void afficher () {
      System.out.println("Forme");
    }

}
```



- Donc, il faut changer l'entête de la classe Rectangle.

```
Public class Rectangle extends Forme {
      // Le reste est le même
}
```

Polymorphisme – 3 (instanceof)

- L'opérateur ***instanceof*** permet de savoir à quelle classe appartient une instance :

```
for (int i = 0 ; i < tableau.length ; i++) {
     if (tableau[i] instanceof Forme) {
         System.out.println("element ["+i+"] est une forme");
     }
       if (tableau[i] instanceof Rectangle) {
         System.out.println("element["+i+"] est un rectangle");
     }
     if (tableau[i] instanceof Carre) {
         System.out.println("element["+i+"] est un carre");
     }
}
```

- Donc, il faut changer l'entête de la classe Rectangle.

Polymorphisme – 4 (instanceof)

- Le code précédent produira :

```
element[0] est une forme
element[0] est un rectangle
element[1] est une forme
element[1] est un rectangle
element[2] est une forme
element[2] est un rectangle
element[2] est un carre
element[3] est une forme
```

- Donc,
  - un Carre est un Rectangle;
  - un Rectangle est une Forme;
  - un Carre est une Forme.

Polymorphisme – 5 (Redéfinition)

- Une des propriétés induites par le polymorphisme est que l'interpréteur Java est capable de trouver le traitement à effectuer lors de l'appel d'une méthode sur un objet.
  - Pour plusieurs objets déclarés sous la même classe (mais n'ayant pas la même classe réelle), le traitement associé à une méthode donné peut être différent.
    - Si cette méthode est redéfinie par la classe réelle d'un objet (ou par une classe située entre la classe réelle et la classe de déclaration), le traitement effectué est celui défini dans la classe la plus spécifique de l'objet et qui redéfinie la méthode.
  - Dans notre exemple, la méthode afficher() est redéfinie dans toutes les sous-classes de Forme et les traitements effectués sont :

Polymorphisme – 6 (Redéfinition)

- Soit le code Suivant :

```
public static void main (String [] args ) {
        Forme f1 = new Forme ();
        Rectangle r1 = new Rectangle (2, 1);
        Forme f2 = r1;
        Forme f3 = new Carre (3);

        f1.afficher ();
        f2.afficher ();
        f3.afficher ();

}
```

- Résultat :

```
Forme
Rectangle 2x1
Carre 3
```

Polymorphisme – 7 (ré-déclaration)

- Nous avons vus qu'une méthode de classe (Statique) ne peut pas être redéfinit.
  - Elle aussi peut être ré-déclarée dans la sous-classe. (plus de détail dans le Polymorphisme)
- Preuve – Si on rajoute les méthodes suivantes comme suit :

```
Public static void tester () {
        System.out.println ("Forme");
}
```

Dans la classe Forme

```
Public static void tester () {
        System.out.println ("Rectangle");
}
```

Dans la classe Rectangle

```
Public static void tester () {
        System.out.println ("Carre");
}
```

Dans la classe Carre

Polymorphisme – 8 (ré-déclaration)

- Soit le code Suivant :

```
public static void main (String [] args ) {
        Forme f1 = new Forme ();
        Forme f2 = new Rectangle (2, 1);
        Forme f3 = new Carre (3);

        f1.tester ();
        f2.tester ();
        f3.tester ();

}
```

- Résultat :

```
Forme
Forme
Forme
```

Polymorphisme – 9 (Conversion ou cast)

- Soit le code Suivant :

```
public static void main (String [] args ) {
        Forme f1 = new Forme ();
        Forme f2 = new Rectangle (2, 1);
        Forme f3 = new Carre (3);

        Rectangle r = (Rectangle) f2;
        Carre c = (Carre) f3;

        f1.tester (); // Il vaut mieux de faire Forme.tester ();
        r.tester (); // de même Rectangle.tester ();
        c.tester (); // de même Carre.tester ();

}
```

- Résultat :

```
Forme
Rectangle
Carre
```

Interfaces - 1

- Une interface est un type, au même titre qu'une classe, mais abstrait et qui donc ne peut être instancié.
- Une interface décrit un ensemble de signatures de méthodes, sans implémentation, qui **doivent être implémentées** dans toutes les classes qui *implémentent* l'interface.
- L'utilité du concept d'interface réside dans le regroupement de plusieurs classes, tel que chacune implémente un ensemble commun de méthodes, sous un même type.
- Une interface possède les caractéristiques suivantes :
  - elle contient des signatures de méthodes ;
  - elle ne peut pas contenir de variables, seulement des constantes;
  - une interface peut hériter d'une ou plusieurs interface (avec le mot-clé **extends**) ;
  - une classe (abstraite ou non) peut implémenter plusieurs interfaces.
  - La liste des interfaces implémentées doit alors figurer après le mot-clé **implements** placé dans la déclaration de classe, en séparant chaque interface par une virgule.

Interfaces - 2

- Dans notre exemple, Forme peut être une interface décrivant les méthodes qui doivent être implémentées par les classes Rectangle Carre (même si celle-ci peut profiter de son héritage de Rectangle).
- L'interface Forme s'écrit alors de la manière suivante :

```
public interface Forme {
        public int surface() ;
        public void afficher() ;
}
```

- Et Les classes Rectangle et cercle deviennent :

```
public class Rectangle implements Forme {
 // Le contenu reste le même
}
```

```
public class Carre implements Forme {
 // Le contenu reste le même
}
```

Interface - 3

- Soit le code Suivant :

```
public abstract interface Forme {
        public abstract int surface() ;
        public abstract void afficher() ;
}
```

- Qui est équivalent à celui là

```
public interface Forme {
        int surface() ;
        void afficher() ;
}
```

- **Remarques**
  - Les méthodes d'une interface sont implicitement **public** et **abstract**.
  - Une interface est implicitement abstraite (**abstract**).
  - Le polymorphisme fonctionne sur les interfaces.
  - Les méthodes d'une interface ne peuvent êtres ni **static** ni **final**.

Interface - 4

- Les variables déclaré dans une interfaces sont implicitement **public static** et **final**.
- Les méthodes d'une interface ne contiennent pas d'implémentation.
  - Elle se terminent par **";"** et non pas par **"{}"**.

```
public static final double PI = 3.14;
// Les écritures suivaantes sont équivalents à la 1ère
public final static double PI = 3.14;
public final double PI = 3.14;
public double PI = 3.14;
double PI = 3.14;
static final double PI = 3.14;
// d'autres combinaisons sont possible
```

- Si une classe implémente une interface mais que le programmeur n'a pas écrit l'implémentation de toutes les méthodes de l'interface, une erreur de compilation se produira sauf si la classe est **une classe abstraite**.

Classes abstraites - 1

- Le concept de classe abstraite se situe entre celui de classe et celui d'interface.
  - C'est une classe qu'on ne peut pas directement instancier car certaines de ses méthodes ne sont pas implémentées.
- Une classe abstraite peut donc contenir
  - des variables,
  - des méthodes implémentées
  - et des signatures de méthode à implémenter (abstraites).
- Une classe abstraite
  - peut implémenter (partiellement ou totalement) des interfaces
  - et peut hériter d'une classe ou d'une classe abstraite.

Classes abstraites - 2

- Le mot-clé **abstract** est utilisé
  - devant le mot-clé class pour déclarer une classe abstraite,
  - dans la les signatures de méthodes à implémenter.
- Lorsqu'une classe hérite d'une classe abstraite, elle doit :
  - soit implémenter les méthodes abstraites de sa super-classe en les dotant d'un corps ;
  - soit être elle-même abstraite si au moins une des méthodes abstraites de sa super-classe reste abstraite.
- Une classe peut être marqué abstraite même si toutes ses méthodes sont concrètes.
- Une classe ne peut jamais marqué final et abstract.
  - L'unique objectif d'une classe abstraite est d'être héritée.
- Une méthode abstraite ne peut être marqué ni final ni private.

Classes abstraites - 3

- Imaginons que l'on souhaite attribuer deux variables, origineX et origineY, à tout objet représentant une forme.
- Comme une interface ne peut contenir de variables,
  - il faut transformer Forme en classe abstraite comme suit :

```
public abstract Forme {
        protected int origineX;
        protected int origineY;

        public Forme (int origineX, int origineY) {
                this.origineX = oroginex;
                this.origineY = orogineY;
        }
        // Les accesseurs et les mutateurs
        public abstract int surface() ;
        public abstract void afficher() ;
}
```

Classes abstraites - 4

- De plus, il faut rétablir l'héritage des classes Rectangle et Carre vers Forme :

```
public class Rectangle extends Forme {
...
}
```

```
public class Carre extends Rectangle {
...
}
```


The End