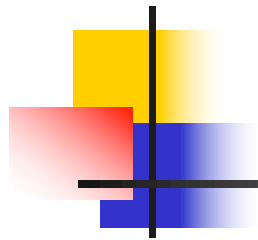


L'héritage

Dériver des classes,
constituer une hiérarchie,
redéfinir des méthodes,
utiliser les interfaces...



Objectif de ce cours

- Apprendre à utiliser au mieux l'héritage
- Comprendre comment le mettre en œuvre via l'utilisation de l'héritage simple, des classes abstraites et des interfaces
- Découvrir la classe Object, racine de la hiérarchie d'héritage en Java
 - Tracer, comparer et copier les objets
- Pour les curieux, découvrir le concept de classes internes et de classes anonymes



Pas d'héritage multiple...

- Java n'autorise pas l'héritage multiple...
- Par défaut toutes les classes dérivent de la classe `java.lang.Object`
- La classe `Object` est la racine de la hiérarchie de classe



Plan

- Introduction
- L'utilisation de l'héritage
- Dériver une classe
- Redéfinition de méthodes et champs
- Le Polymorphisme
- Hériter de la classe Object
 - Test d'égalité entre objets
- Classes abstraites
- Interfaces
 - Le clonage d'objets
- Les classes internes (+ anonymes)



Conception Objet

- Agrégation, association
 - On peut construire une instance à partir d'autres instances

```
public class Salarie{  
    private String nom;  
    private String id; // clef  
    private Adresse adresse;  
    private Salarie superieurHierarchique;  
}
```

Agrégation : cycle de vie
étroitement lié

Association : référence à une instance
relativement indépendante



Conception Objet

- Notion de classe Cliente, et d'utilisation
 - La classe Salarie est une classe cliente de la classe Adresse
 - La classe Salarie demande des services à la classe Adresse, elle l'utilise

```
public class Salarie{  
    private String nom;  
    private String id; // clef  
    private Adresse adresse;  
    private Salarie superieurHierarchique;  
}
```



Pourquoi utiliser l'héritage ?

- L'héritage permet de modéliser la relation « est un »
 - un Rectangle **est un** parallélépipède
 - un Cercle **est une** Ellipse
 - une Ellipse **est une** FormeGéométrique
 - Ils ont tous une position, une couleur,...
 - Ils peuvent tous être dessinés, déplacés,...
 - On peut calculer leur surface, leur périmètre
 - Ils ne sont pourtant pas semblables, $\pi * r^2 \neq L * l$

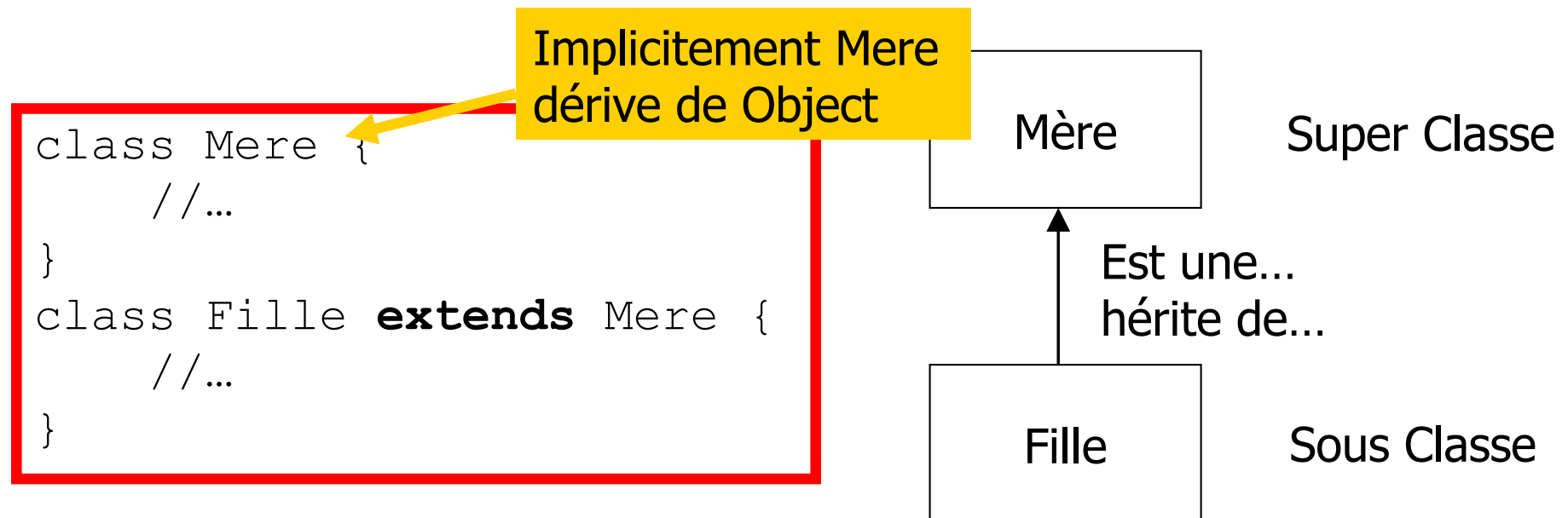


Pourquoi utiliser l'héritage ?

- Utiliser une classe existante pour définir une nouvelle classe,
 - la classe mère (super classe) permet de définir plus facilement la classe fille (sous-classe)
 - même si on a que le code compilé ".class«
 - réutilisation et minimisation des changements
- La classe fille peut être vue comme une spécialisation de la classe mère...

Dériver une classe

- Dérivation explicite en utilisant le mot clé **extends**





Quid de la classe fille ?

- Elle hérite des membres et méthodes de sa classe mère (super classe)
- Elle peut avoir en plus ses propres membres et méthodes (d'instance et de classe)
- Elle peut redéfinir, en spécialisant (*override*) les méthodes dont elle hérite
- Elle possède ses propres constructeurs
- Elle ne peut retirer aucune variable ou méthode

```
class Cercle{
    protected Point centre;
    protected double rayon = 0.0;
    public Cercle(Point c, double r){
        centre = c;
        rayon = r;
    }
    public Cercle(){this(new Point(0,0),0.0);}
    public double area(){
        return Math.PI*rayon*rayon;
    }
    public double perimeter(){
        return 2*Math.PI*rayon;
    }
    public void dessineToi(){...}
}
```

Modification en vue
d'une spécialisation

```
class CercleCouleur extends Cercle{
    private String couleur;
    public CercleCouleur(Point ce, double r, String co){
        centre = ce; rayon = r; couleur = co;}
    public String getCouleur(){
        return couleur;
    }
    public void setCouleur(String co){
        couleur = co;
    }
    public void dessineToi(){...} //redéfinition
}
```



Écriture du code de la classe fille, que faire ?

- Écrire le code (variables et/ou méthodes) spécifique au comportement de la classe fille
- Redéfinir certaines méthodes dont le comportement n'est plus approprié
- Écrire les constructeurs de la classe



Spécialisation de la classe fille

- Déclarations de champs et méthodes spécifiques

```
public class Salarie{  
    private String nom;  
    private String id; // clef  
    private Adresse adresse;  
    private Salarie superieurHierarchique;  
    ...  
}
```

Un Chef est un Salarié
avec en plus...

```
public class Chef extends Salarie{  
    //Champs spécifiques  
    private Salarie[] subordonnés;  
    //Méthodes spécifiques  
    public Salarie[] getSubordonnés() {...}  
}
```



Redéfinition des méthodes

Spécialiser et/ou adapter le
comportement des méthodes
héritées



Redéfinition des méthodes

- La sous-classe peut, redéfinir les méthodes héritées
 - Réécrire la méthode en conservant la même signature (nom + paramètres) et le même type de retour mais en modifiant le code

```
public class Chef extends Salarie{  
    ...  
    public double salaire() {  
        ...  
        return ...;  
    } ...  
}
```

Un Chef perçoit un salaire, mais il n'est pas calculé de la même façon que celui du « simple » Salarié



Redéfinition des méthodes

- Étendre le comportement d'une méthode héritée plutôt que de la redéfinir
 - utiliser la définition de la classe Mère (super) et la compléter

```
public class Personne{  
    ...  
    public String identite(){  
        return nom+prenom+  
            adresse+nTelephone;  
    }...  
}
```

```
public class Salarie extends Personne{  
    private String profession;  
    int salaire;  
    public String identite(){  
        return super.identite()+  
            profession+salaire;  
    }...}  
}
```




Redéfinition et surcharge

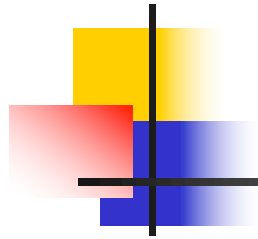
- **Attention** : redéfinition ne veut pas dire surcharge...
- Redéfinition
 - même type de retour, même nom, mêmes paramètres
- Surcharge
 - même type de retour, même nom, mais des paramètres (nombres et/ou types) différents
 - La méthode de la classe Mère n'est pas redéfinie



Limites de la redéfinition

- Les méthodes et les champs déclarés avec le modificateur *final* ne sont pas redéfinissables

```
FinalTest.java:7: Final methods can't be overridden.  
    Method void iamfinal() is final in class  
    ClassWithFinalMethod.  
        void iamfinal() {  
            ^  
1 error
```



Limites de la redéfinition

- La méthode redéfinie dans la classe fille ne doit jamais être moins accessible que la méthode originale de la classe mère
 - On ne peut pas passer d'une méthode `public` à une méthode `protected` ou `private`



La pseudo variable super

- Une sous-classe peut accéder aux méthodes redéfinies de sa super classe en utilisant la pseudo-variable `super`.

```
class A{
    public A() {...}
    public A(..., ...) {this(); ...}
    public void methode() {...}
}
class B extends A{
    public B(){super() // inutile car implicite}
    public B(..., ..., ...) {super(..., ...); ...;}
    public void methode() {...; super.methode(); ...;}
}
```

- `super` ne peut être invoquée depuis une méthode statique



Masquage de champs

- Les variables membres (d'instance ou de classe) définies dans la sous-classe masquent celles de la classe Mère ayant le même nom
- On peut cependant y accéder via le mot clé super...
- Attention : ce masquage nuit à la lisibilité du code

```
class Super{  
    Float aNumber;  
}
```

```
class Filles extends Super{  
    Float aNumber;  
    ...  
    // accès au champs de la classe Mère  
    super.aNumber;  
}
```



Masquage de champs

```
class A {  
    int x;}  
class B extends A{  
    int x;}  
class C extends B{  
    int x,a;  
    void test(){  
        a=super.x;  
        //a=super.super.x;  
        a=((B)this).x;  
        a=((A)this).x;  
    }  
}
```

Pour l'accès aux champs c'est le type déclaré qui importe



Masquage de champs

```
class A {  
    int x;}  
class B extends A{  
    int x;}  
class C extends B{  
    int x,a;  
    void test(){  
        a=super.x;           // x de B  
        a=super.super.x;    // Error; on ne peut remonter que  
                             // d'un niveau...  
        a=((B)this).x;       // x de B  
        a=((A)this).x;       // x de A  
    }  
}
```

Pour l'accès aux champs c'est
le type déclaré qui importe



Masquage des méthodes static

- On ne peut pas redéfinir une méthode déclarée `static`, on peut seulement la cacher comme on le fait avec les variables
- Cependant pour accéder à la méthode cachée, on ne peut pas utiliser `super`
- Il faut préfixer l'appel de la méthode du nom de la classe, ou utiliser un cast

```
(ClasseMere)m() ou ClasseMere.m()
```




Constructeurs des classes filles



Constructeurs des classes dérivées

- Pour construire un objet d'une sous classe (un objet spécialisé) il faut partir d'un objet de la classe Mère, la super classe (l'objet général)...
 - La première instruction du constructeur doit être un appel à un constructeur de la classe Mère ou à un autre constructeur de la classe (`this(...)`)
 - Invocation via le mot clé `super` :

```
super(); // invocation du constructeur sans arg
```

Ou

```
super(par1, par2, ..., parn) // ; constucteur à n args
```



Les constructeurs des classes dérivées

- Si il n'y a pas d'invocation explicite (`this (...)` ou `super (...)` en 1^{ère} instruction) alors il y a invocation implicite de `super ()` qui doit, sous peine d'erreur de compilation, être défini

```
class CercleCouleur extends Cercle {...
    public CercleCouleur(Point p, double r,String c) {
        super(p, r); // invocation du constructeur de Cercle
        couleur = c; //construction spécifique à CercleCouleur
    }
```

```
class CercleCouleur extends Cercle {
    public CercleCouleur(String c){
        // invocation implicite à super()
        couleur = c; //construction spécifique
    }
```



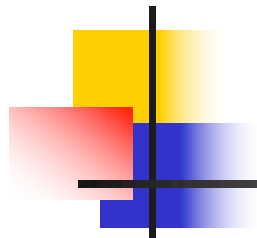
En résumé

```
public class Fille extends Mère{
    //Champs spécifiques de la classe fille
    private type champsParticulier;
    ...
    //Méthodes spécifiques
    public type méthodeTypique(type paramètres){
    ...}
    //Méthodes redéfinies
    public type mêmeNomClasseMère(){
    ...}
    //Méthodes étendues
    public type méthodeAEtendre(){
        ...
        super.méthodeAEtendre();
        ...
    }...
}
```



Polymorphisme

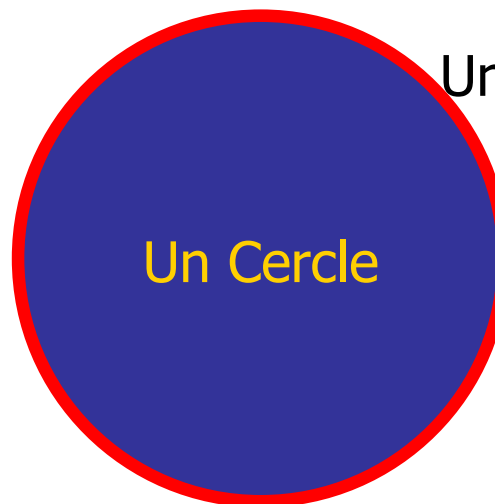
Polymorphe, du grec *polumorphos* formé à partir du grec ancien πολλοί (*polloí*) qui signifie « plusieurs » et μορφος (*morphos*) qui signifie « forme »



Le polymorphisme



Une Ellipse **est une** forme 2D



Un cercle **est une** Ellipse
est une forme 2D

Polymorphisme

- Un objet peut être vu comme une instance de n'importe laquelle des classes héritées de la classe dont il est l'instance
- Le polymorphisme permet d'écrire :

```
Forme2D uneForme = new Ellipse(2.0, 3.0);  
uneForme = new Circle(4.0);
```

- uneForme est à la fois une Forme2D et un Circle ou une Ellipse

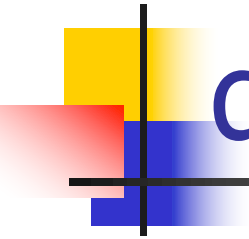


Affectation et compatibilité de types

- A un objet on peut affecter un autre objet de sa classe ou d'une de ses sous-classes

```
Salarie salarie = new Chef(); // Upcasting
Chef leChef;
Salarie[] equipe = {
    new Salarie(), new Salarie(), new Chef() };
Salarie sub = salarie.getSubordonnés();
// Error :method getSubordonnés not found in class Salarie
leChef = salarie;
// Error :Incompatible type for =. Explicit cast need
String nom = salarie.getNom();
```

Dans ce contexte comment savoir quelle est la méthode qui sera exécutée ?



Polymorphisme et liaison dynamique ou *dynamic binding*

- Lors d'une invocation de méthode c'est le type **réel** de l'objet (à l'exécution) qui importe et non son type déclaré.

```
class Shape{  
    public void draw() {  
        System.out.println("Shape draw");  
    }  
class Square extends Shape{  
    public void draw() {  
        System.out.println("Square draw");  
    }  
}
```

```
Shape s = new Square();  
s.draw();
```



Polymorphisme...

```
class A{
    int x;
    void m() {...}
}
class B extends A{
    int x;
    void m() {...}
}
```

```
class C extends B{
    int x,a;
    void m() {...}
    void test() {
        a=super.x;
        a=((B)this).x;
        a=((A)this).x;
        super.m();
        //super.super.m();
        ((B)this).m();
        A instA = new B();
        instA.m();
    }
}
```



Polymorphisme...

```
class A{
    int x;
    void m(){...}
}
class B extends A{
    int x;
    void m(){...}
}
```

```
class C extends B{
    int x,a;
    void m(){...}
    void test(){
        a=super.x;
        a=((B)this).x;
        a=((A)this).x;
        super.m();
        super.super.m();
        ((B)this).m();
        A instA = new B();
        instA.m();
    }
}
```

```
// x de B
// x de B
// x de A
// m() de B
// Error
// m() de C
// m() de B
```

Exemple d'utilisation du polymorphisme

```
class Schema{
    private Forme2D formes[];
    public void draw() {
        for (i=1;i<nbFormes;i++)
            formes[i].draw();
    }
    ...
    public static void main(String[] args){
        Schema s = new Schema(10);
        s.ajoute(new Square(...));
        s.ajoute(new Circle(...));
        s.ajoute(new Rectangle(...));
        ...
        s.draw();
    }
}
```

Les méthodes draw
appelées correspondront
aux type réel des objets
stockés dans formes[i]



S'assurer du type effectif d'une instance : l'opérateur instanceof

- Savoir si une variable fait référence à un objet d'une Classe donnée
- Opérateur booléen
- Syntaxe d'appel

```
référence instanceof NomClasse; // retourne un booléen
```

- Utilisation
 - Tester la nature d'un objet avant de décider ce qu'il convient d'en faire



L'opérateur instanceof

■ Exemples d'utilisation

```
Ellipse e = new Ellipse(2.0, 3.0) ;  
Circle c = new Circle(4.0) ;  
System.out.println(e instanceof Circle) ;  
System.out.println(e instanceof Ellipse) ;  
System.out.println(c instanceof Circle) ;  
System.out.println(c instanceof Ellipse) ;  
System.out.println(e instanceof Object) ;
```

```
Salarie[] equipe = {...};  
for (...)  
    if (equipe[i] instanceof Chef)  
        System.out.println(  
            ((Chef)equipe[i]).getSubordonnés().length);
```



L'opérateur instanceof

■ Exemples d'utilisation

```
Ellipse e = new Ellipse(2.0, 3.0) ;  
Circle c = new Circle(4.0) ;  
System.out.println(e instanceof Circle) ;      false  
System.out.println(e instanceof Ellipse) ;      true  
System.out.println(c instanceof Circle) ;      true  
System.out.println(c instanceof Ellipse) ;      true  
System.out.println(e instanceof Object) ;      true
```

```
Salarie[] equipe = {...};  
for (...)  
    if (equipe[i] instanceof Chef)  
        System.out.println(  
            ((Chef)equipe[i]).getSubordonnés().length);
```



Polymorphisme (suite...)

- Soit C la classe réelle d'un objet o sur lequel on invoque la méthode $m()$: $o.m()$
- Si la classe C contient une méthode $m()$ alors c'est cette méthode qui est exécutée
- Sinon la recherche de la méthode $m()$ se poursuit dans la classe mère de C , puis dans la classe mère de la classe mère et ainsi de suite jusqu'à trouver une méthode $m()$ qui sera alors exécutée



L'encapsulation

Université de Corse



Qu'est ce qui est hérité ?

- La classe fille hérite de tous les membres de sa classe Mère (à part les constructeurs)
- Par contre il se peut qu'elle n'ait pas accès à certains des membres dont elle a hérités (ceux déclarés `private`)
- Ces membres sont utiles pour le bon fonctionnement de la classe fille mais elle ne peut pas directement les utiliser (en particulier les modifier)



Le modificateur `protected`

- Ce modificateur, moins permissif que `public` et plus tolérant que `private`, permet à une classe de manipuler les membres dont elle a hérités
- L'accès est permis à la classe elle-même, aux classes filles et aux classes du package
- Elle ne lui permet cependant pas d'avoir accès aux membres `protected` qui définissent l'état des instances de sa classe mère.

Exemple de code protected...

Accès classe fille et package

```
package Greek;

public class Alpha{
    protected int iamprotected;
    protected void protectedMethod(){
        System.out.println("protectedMethod");
    }
}
```

```
package Greek;

class Gamma{
    void accessMethod(){
        Alpha a = new Alpha();
        a.iamprotected = 10; // legal
        a.protectedMethod(); // legal
    }
}
```

Accès permis, car la classe Gamma appartient au même package que la classe Alpha



protected...

Accès classe fille et package

```
package Latin;
import Greek.*;
class Delta extends Alpha{
    void accessMethod(Alpha a, Delta d){
        a.iamprotected = 10; // illegal
        d.iamprotected = 10; // legal
        a.protectedMethod(); // illegal
        d.protectedMethod(); // legal
    }
}
```

La classe `Delta` peut accéder au champs `iamprotected` et à la méthode `protectedMethod`, mais seulement via des objet de type `Delta` (car c'est une classe fille de `Alpha`) ou du type d'une de ses sous-classes. La classe `Delta` ne peut accéder aux membres `protected` des objets de type `Alpha` sa classe mère (car elle n'appartient pas au même package).



Exemple d'utilisation de protected

```
public class Véhicule{
    protected double vitesse;
    ...
}

public class Voiture extends Véhicule{
    private MoteurExplosion moteur;
    public Voiture(double v,MoteurExplosion m){
        vitesse = v; // on peut utiliser directement vitesse
        moteur = m;
    }
}
```



Le modificateur par défaut package

- L'accès aux champs déclarés sans modificateur (ou modificateur `package`) est permis depuis toutes les classes du même package
- L'accès est interdit aux autres

Exemple, modificateur par défaut accès package

```
package Greek;
```

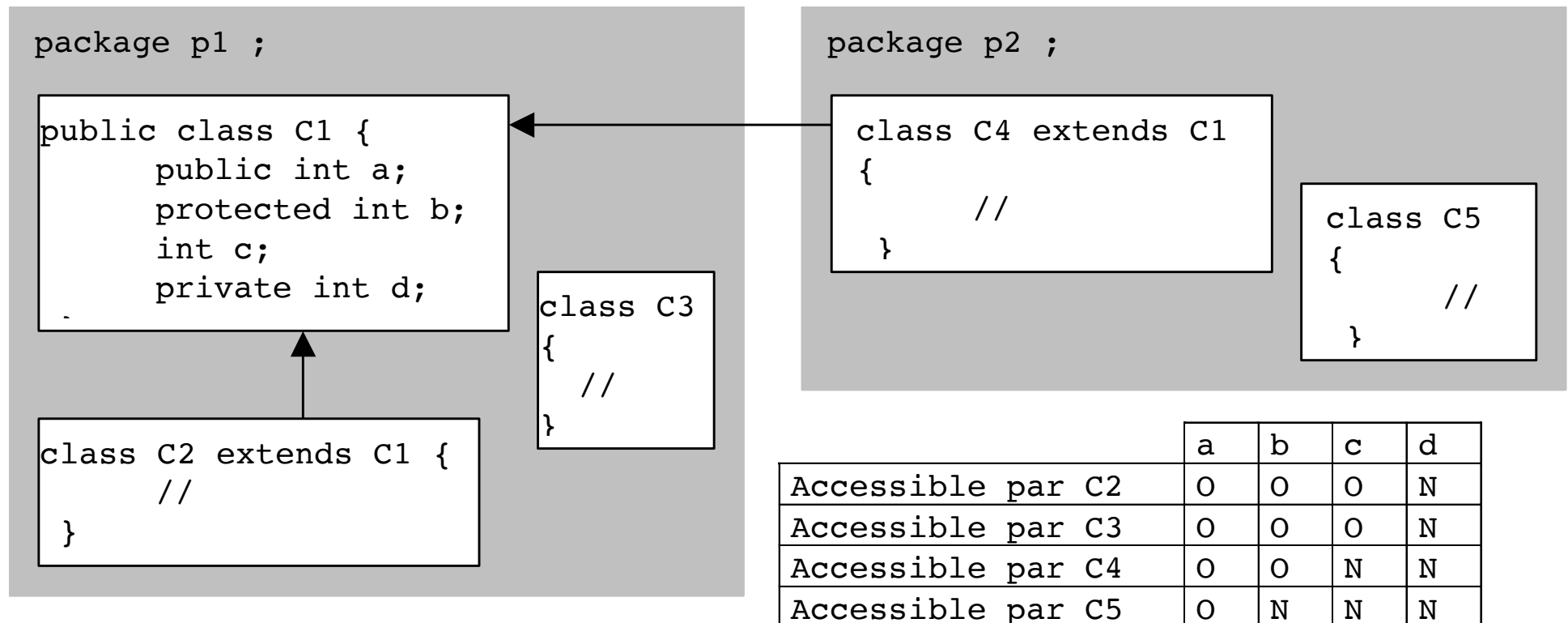
```
class Alpha{  
    int iampackage;  
    void packageMethod() {  
        System.out.println("packageMethod");  
    }  
}
```

```
package Greek;
```

```
class Beta{  
    void accessMethod(){  
        Alpha a = new Alpha();  
        a.iampackage = 10; // legal  
        a.packageMethod(); // legal  
    }  
}
```

Accès permis, car la
classe Gamma
appartient au même
package que la classe
Alpha

Résumé : l'encapsulation des membres





Déclarer un constructeur protected

- Si le constructeur d'une classe est déclaré protected
 - Il peut être appelé depuis le constructeur d'une classe fille, via l'invocation à `super (...)`
 - La classe fille ne peut pas créer directement de nouvelles instance de la classe mère par `new` (sauf dans le cas ou elle appartient au même package)



Le modificateur final

- Les classes déclarées avec le modificateur `final` ne peuvent être dérivées
 - String et Vector
- Les méthodes `final` ne peuvent être redéfinies
- Les variables `final` ne peuvent changer de valeur après leur initialisation



Hériter de la classe Object



La classe Object

- C'est la racine de l'arbre d'héritage en Java : `java.lang.Object`
- Toutes vos classe sont des descendantes de Object
- Elle n'a ni variables d'instance ni variables de classe
- Elle fournit plusieurs méthodes qui sont automatiquement héritées par toutes les autres classes



Les

méthode de la classe Object

- `String toString()`
- `boolean equals(Object o)`
- `public Class getClass()`
- `protected void finalize()`
- `protected Object clone()` **cf. Ch. Interface**
- `int hashCode()` **cf. Ch. Collection**
- `void notify()`, `void notifyAll()`, `void wait()` **cf. Ch. Thread**



Sortie texte d'une instance

`public String toString()`

- Renvoie une description de l'instance courante sous la forme d'une chaîne de caractère
- Lorsqu'une référence à un objet est passée en paramètre à la méthode `println`,
 - c'est le résultat de l'invocation à `toString` sur cette référence qui est imprimé
 - sinon c'est l'adresse de l'objet qui est affichée...
- Si vous voulez utiliser cette possibilité (très utile pour les phases de mise au point), vous devez la redéfinir dans vos classes



Exemple de code : toString()

```
public class Personne{
    ...
    public String toString(){
        return "Personne : "+nom+", "+prenom+
            "adresse : "+adresse + "tel: " + nTelephone;
    }
    ...
    public static void main(String args[]){
        Personne p = new Personne(...);
        System.out.println(p); // appel implicite à
                                // p.toString()
    }
}
```




Comparer les objets

- Attention les variables ne représentent que des références...
- Que compare-t'on ?

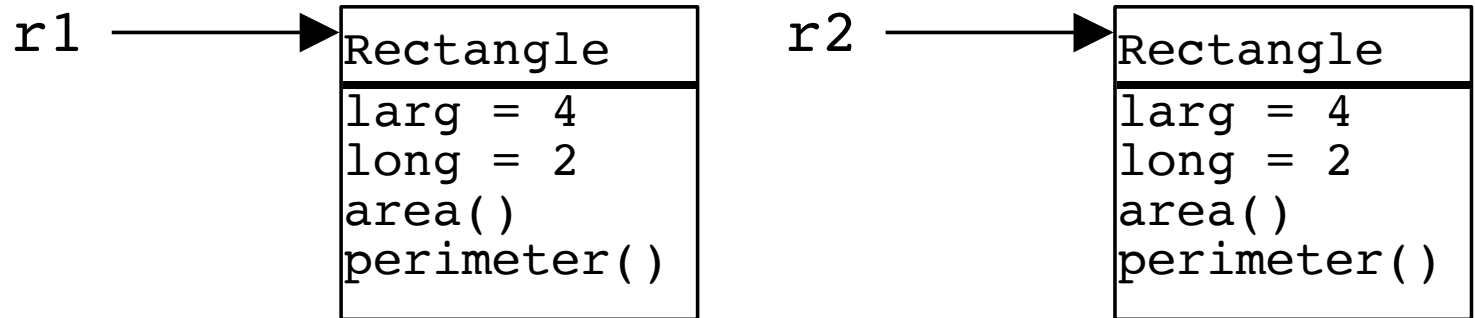
```
Rectangle r1 = new Rectangle(2,4);  
Rectangle r2 = new Rectangle(2,4);  
if (r1 == r2) ...
```

Comparer les objets

- Attention les variables ne représentent que des références...
- Que compare-t'on ?

```
Rectangle r1 = new Rectangle(2,4);  
Rectangle r2 = new Rectangle(2,4);  
if (r1 == r2) ...
```

Le test
rend
FAUX !!





Comparer les objets méthode : equals

- Permet de comparer le contenu des objets et pas seulement les références

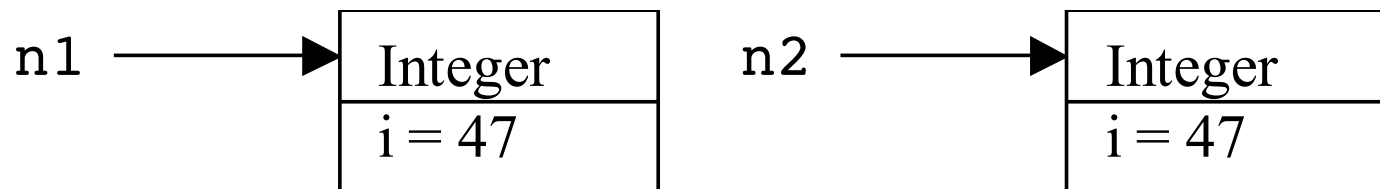
```
public class EqualsMethod {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1.equals(n2));  
    }  
}
```

Comparer les objets méthode : equals

- Permet de comparer le contenu des objets et pas seulement les références

```
public class EqualsMethod {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1.equals(n2));  
    }  
}
```

Le test
rend
VRAI !!





La méthode equals()

- `public boolean equals(Object obj)`
renvoie vrai si `obj` a la même valeur que l'instance courante (`this`). C'est-à-dire si `obj` et `this` référence le même objet
- Si ce comportement par défaut ne vous convient pas vous devez redéfinir la méthode `equals`
- Dans ce cas vous devez également redéfinir la méthode `hashCode()` ... (nous y reviendrons)

Redéfinir `hashCode` :

http://java.developpez.com/faq/java/?page=divers#DIVERS_hashCode



Exemple de code : equals()

■ Comportement par défaut

```
class Value {  
    int i;  
}  
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        Value v1 = new Value();  
        Value v2 = new Value();  
        v1.i = v2.i = 100;  
        System.out.println(v1.equals(v2));  
    }  
}
```



Exemple de code : equals()

■ Comportement par défaut

```
class Value {  
    int i;  
}  
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        Value v1 = new Value();  
        Value v2 = new Value();  
        v1.i = v2.i = 100;  
        System.out.println(v1.equals(v2));  
    }  
}
```

**Le test
rend FAUX !!**



Exemple de code : equals()

■ Redéfinition de equals

```
class Value {
    int i;
    public boolean equals(Object o) {
        boolean result = false;
        if ((o != null) && (o instanceof Value))
            result =(i == ((Value)o).i);
        return result;
    }
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
}
```




Exemple de code : equals()

■ Redéfinition de equals

```
class Value {  
    int i;  
    public boolean equals(Object o) {  
        boolean result = false;  
        if ((o != null) && (o instanceof Value))  
            result =(i == ((Value)o).i);  
        return result;  
    }  
}
```

```
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        Value v1 = new Value();  
        Value v2 = new Value();  
        v1.i = v2.i = 100;  
        System.out.println(v1.equals(v2));  
    }  
}
```

**Le test
rend VRAI !!**



getClass()

Retrouver la classe d'un objet

- La méthode `public Class getClass()` permet de renvoyer un objet représentant la classe réelle de l'objet
- La méthode `getName()` de la classe `Class` renvoie le nom de la classe

```
void PrintClassName(Object obj) {  
    System.out.println("The Object's class is " +  
        obj.getClass().getName());  
}
```

Deprecated depuis v9

Méthode : finalize()

- Automatiquement invoquée par le *GC*, avant la destruction de l'objet

```
public class Point{  
    //...  
    protected void finalize(){  
        System.out.println("Je suis garbage Collecté");  
        //... Votre code  
        super.finalize();  
        //invocation de la méthode finalize  
        //de la super classe    }  
}
```

```
Point monPoint;  
if (condition) { Point p2 = new Point(); monPoint = p2;}  
// La référence p2 n'est plus valide mais reste monPoint  
monPoint = null;  
/* l'instance n'est plus référencée. Elle peut a tout moment  
être détruite par le GC */
```

Deprecated depuis v9

Pourquoi utiliser finalize() ?

- Pour libérer les ressources système utilisées par un objet qui va être détruit : fichiers, connexions,...

```
public class ProcessFile{
    private FileInputStream file;
    public ProcessFile(String path);
    file = new FileInputStream(path);
    //...
    public void close(){
        if (file != null) {
            file.close();
            file = null;
        }
    }
}
```

```
protected void finalize()
    throws Throwable{
    close();
    super.finalize();
}
```

Deprecated depuis v9

Remarques sur `finalize()`

- Attention, ne comptez pas uniquement sur cette méthode pour libérer les ressources. En effet vous ne savez pas quand le GC va être appelé...
- La dernière instruction de votre redéfinition de `finalize()` doit toujours faire appel à `super.finalize()` pour libérer des ressources éventuellement acquise par la classe mère



Les classes abstraites

Université de Corse



Définitions

- Une classe abstraite est une classe qu'on ne souhaite pas pouvoir instancier (pas de new)
- Une classe abstraite peut ou non contenir des méthodes abstraites
- Une méthode abstraite ne possède pas de définition (pas de code...)
- Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite
- Une méthode `static` ne peut être abstraite car on ne peut redéfinir une méthode `static`



A quoi ça sert ?

- A définir une interface, un comportement commun à plusieurs classes sans rien figer
- A définir un concept
- Exemple : Forme2D...
 - On doit pouvoir calculer l'aire et le périmètre
 - Comment : ???????
- Solution : Forme2D est définie comme une classe abstraite



Syntaxe, exemple

```
public abstract class Forme2D{  
    public static final double PI =  
    3.14159;  
    private int x,y; //Position de la  
    forme
```

```
    //Méthodes identiques figées  
    void deplaceEn(int nvX,int nvY){  
        x=nvX; y=nvY;  
    }  
    //Méthodes abstraites  
    public abstract double aire();  
    public abstract double  
    perimetre();  
    public abstract void dessine()  
}
```

La classe fille doit, pour ne pas être elle même abstraite redéfinir toutes les méthodes abstraites de sa classe Mère...

```
class Rectangle extends Forme2D{  
    private int long,larg;  
    //redéfinition  
    public double aire(){  
        return long*larg;  
    }  
    public double perimetre(){  
        return 2*(long+larg);  
    }  
    public void dessine(){...}  
}
```



Classes abstraites

- Une classe abstraite peut contenir des méthodes abstraites et des méthodes "normales" (avec code)

```
abstract class NomClasse{...}
```

- Si une classe contient une méthode déclarée abstraite (dont le code n'est pas fourni) alors la classe à laquelle elle appartient doit également être déclarée abstraite

```
abstract type nomMethode (...);
```



Les interfaces

Université de Corse



Principe

- Déclare un ensemble de méthodes permettant de définir un comportement
 - Ex : spécifier une API, signifie généralement proposer un ensemble d'interfaces qui exposent des fonctionnalités
- Peut intégrer des déclarations de constantes
- Peut être vu comme une classe où la majorité des méthodes sont abstraites



Définition

- Toutes les déclaration de méthodes faites dans une interface sont par défaut publiques, le modificateur `public` est optionnel
 - Les méthodes déclarées sont abstraites, le modificateur `abstract` est optionnel
 - On peut cependant depuis la version 8 ajouter des methodes ayant un code par défaut via le modificateur `default`
 - Ainsi que des méthodes de classe `public static`
- Les déclarations de variables sont implicitement des constantes c'est-à-dire `public static final`



Déclaration

```
[modificateur] interface NomInterface{  
    //Implicitement tout est public  
  
    //Déclaration éventuelles de constantes  
    type NOM_CONSTANTE;  
  
    //Déclaration méthodes abstraites public  
    type nomMethode(types parametres);  
  
    //Depuis v8 Eventuelles méthodes de classes  
    static type nomMethodeStatic(types parametres){  
        ...}  
    //Depuis Java 8 Eventuelles méthodes avec implémentation  
    default type nomMethodeParDefaut(types parametres){  
        ...}  
}
```



Pourquoi les interfaces ?

- Permet dans une certaine mesure de « remplacer » l'héritage multiple
- Permet de se limiter à la définition d'un comportement
- Permet de définir un protocole de communication devant être suivi par les classes implémentant l'interface
- Ultérieurement les objets issus des classes implémentant l'interface peuvent être manipulés comme des instances de l'interface



Utilisation

- On ne peut pas instancier une interface
 - Pas de new...
- Une classe peut implémenter une ou plusieurs interfaces (*implements*) et en même temps hériter d'une seule classe (*extends*)
- Une interface peut hériter d'une ou plusieurs interfaces (*extends*)
- Une classe abstraite peut implémenter une ou plusieurs interfaces



Implémenter une interface

- On utilise le mot clé `implements`

```
[modificateur] class NomClasse implements NomInterface{  
    ...  
}
```

- La classe `NomClasse` doit alors implémenter toutes les méthodes abstraites (non `default`) de l'interface, sinon elle doit être déclarée `abstract`



Exemple d'utilisation

```
interface Insurable{
    void setRisk(String level);
    String getRisk();
}
public class Car extends Vehicle implements Insurable{
    private String risk = "Tier";
    private int seatNumber;
    ...
    public void setRisk(String theRisk){
        ...
        risk=theRisk;
    }
    public String getRisk(){return risk;}
    public void drive(...){
        ...}
}
public class House implements Insurable{
    private String risk;
    private int roomNumber;
    ...
    public void setRisk(String theRisk){
        ...
        risk=theRisk;
    }
    public String getRisk(){return risk;}
    ...
}
```

```
public class InsuranceFirm{
    private Insurable[] InsuredProperties;
    ...
    public void insure(Insurable i){
        ...
        i.setRisk("...");
    }
    public double compensate(Insurable i,int damage){
        ...
        i.getRisk();}
        ...
        return ...;
    }
    ...
}
```



Héritage d'interfaces

- Une interface peut hériter de plusieurs autres interfaces

```
interface NomInterfaceFille extends NomInterfaceMere1
                                   NomInterfaceMere2
                                   ...
                                   NomInterfaceMereN{
    ... //déclaration des méthodes propre à l'interface
    //fille
}
```

Exemple d'héritage

```
interface Monstre{
    void menace();
}
interface MonstreDangereux
    extends Monstre{
    void detruire();
}
interface Mortel {
    void tuer();
}
class Dragon
    implements MonstreDangereux{
    public void menace() {...}
    public void detruire() {...}
}
interface Vampire
    extends MonstreDangereux,
        Mortel{
    void BoitSang();
}
```

```
class SpectacleHorreur{
    static void u(Monstre b){
        b.menace();
    }
    static void v(MonstreDangereux d){
        d.menace();
        d.detruire();
    }
}
```

Code extrait de
ThinkInJava...

```
public static void main(String[] args){
    Dragon toto = new Dragon();
    u(toto);
    v(toto);
}
```



Grouper des constantes

- L'interface est un moyen (l'autre étant le Type Enum) de grouper des constantes puisque par défaut les variables déclarées dans une interface sont `public`, `static` et `final`

```
public interface Mois {  
    int    JANVIER = 1, FEVRIER = 2,  
          MARS = 3, AVRIL = 4, MAI = 5,  
          JUIN = 6, JUILLET = 7, AOUT = 8,  
          SEPTEMBRE = 9, OCTOBRE = 10,  
          NOVEMBRE = 11, DECEMBRE = 12;  
}
```

Utilisation :
Mois.JANVIER



Initialiser les champs dans une interface

- Les champs déclarés dans une interface sont des constantes mais ils peuvent être initialisés avec des expressions non constantes

```
import java.util.*;
public interface RandVals {
    int rint = (int) (Math.random()*10);
    long rlong = (long) (Math.random()*10);
    float rfloat = (float) (Math.random()*10);
    double rdouble = Math.random()*10;
}
```



Exemple de code

```
import java.util.*;
public interface RandVals {
    int rint = (int) (Math.random()*10);
    long rlong = (long) (Math.random()*10);
    float rfloat = (float) (Math.random()*10);
    double rdouble = Math.random()*10;
}
```

```
public class TestRandVals {
    public static void main(String[] args) {
        System.out.println(RandVals.rint);
        System.out.println(RandVals.rlong);
        System.out.println(RandVals.rfloat);
        System.out.println(RandVals.rdouble);
    }
}
```



Exercices/Questions

- Quelles sont les méthodes qu'une classe souhaitant implémenter l'interface `java.lang.CharSequence` doit implémenter ?
- Qu'est ce qui ne va pas dans cette interface ?
- Proposez une correction
- L'interface suivante est elle selon vous valide ?
A quoi pourrait elle servir ?

```
public interface SomethingIsWrong {  
    void aMethod(int aValue){  
        System.out.println("Hi Mom");  
    }  
}
```

```
public interface Marker {  
}
```


Example tutorial Oracle

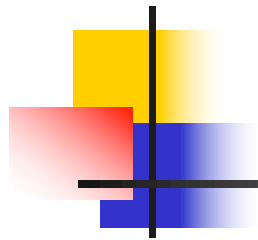
```
package defaultmethods;

import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                               "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```



Les interfaces fonctionnelle

- C'est une interface qui ne contient qu'une et une seule méthode abstraite
 - Cf. lien avec les lambda expressions
- Elle peut contenir par ailleurs des méthodes default/static et des constantes.
- Avant sa déclaration on peut utiliser l'annotation `@FunctionalInterface`.



Choisir entre Interface et classe abstraite

- Une classe abstraite peut avoir un constructeur, une interface non.
- Une classe abstraite peut avoir des champs (variables d'état) une interface non.
 - Ces champs peuvent être public, private ou protected, static ou non, final ou non, alors que dans une interface les champs sont obligatoirement public, static, et final.
- Les méthodes
 - dans une classe abstraite peuvent être de toutes sortes public, private, ou protected, de classe ou d'instance, avec du code (concrètes) ou sans (abstraites)
 - pour une interface, elles peuvent de classe (static) ou d'instance éventuellement posséder une implémentation par défaut (default) elle seront généralement public, même si depuis Java 9 il est possible d'en définir des private (internes à l'interface).
- Simuler l'héritage multiple :
 - Une classe abstraite peut hériter d'une seule autre classe, par contre une classe peut implémenter plusieurs interfaces...
 - Une interface peut hériter de plusieurs interfaces

Intérêt des deux...

Polymorphisme

- L'un des intérêts d'utiliser les classes abstraites et/ou les interfaces est de pouvoir les utiliser comme type pour manipuler des références objets inconnus, mais ayant un comportement (c'est-à-dire une ou des méthodes) identifié

Intérêt des deux...

Polymorphisme, Exemple

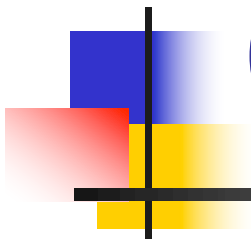
```
public interface Relatable {  
  
    // this (object calling isLargerThan)  
    // and other must be instances of  
    // the same class returns 1, 0, -1  
    // if this is greater than,  
    // equal to, or less than other  
    public int isLargerThan(Relatable other);  
}
```

```
public class RectanglePlus implements Relatable {  
    public int width = 0;  
    ...  
    public RectanglePlus(){...}  
    ...  
    public int isLargerThan(Relatable other){  
        int result = 0;  
        if other instanceof RectanglePlus{  
            RectanglePlus otherRect =  
                (RectanglePlus)other;           if (this.getArea() <  
otherRect.getArea())  
                result = -1;  
            else if (this.getArea() >  
otherRect.getArea())           result = 1;  
                return result;  
        }  
    }  
}
```

```
public Object findLargest(Object object1, Object object2) {  
    Relatable obj1 = (Relatable)object1;  
    Relatable obj2 = (Relatable)object2;  
    if ((obj1).isLargerThan(obj2) > 0)  
        return object1;  
    else  
        return object2;  
}
```

```
public Object findSmallest(Object object1, Object object2) {  
    Relatable obj1 = (Relatable)object1;  
    Relatable obj2 = (Relatable)object2;  
    if ((obj1).isLargerThan(obj2) < 0)  
        return object1;  
    else  
        return object2;  
}
```

Exemple d'interface drapeau (ou marker) : Cloneable

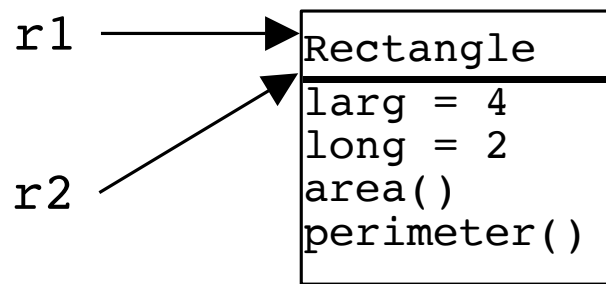


L'interface Cloneable pour copier
les objets



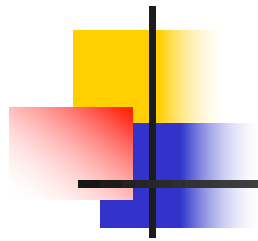
Copier les objets

- Il faut copier l'objet et pas seulement la référence...



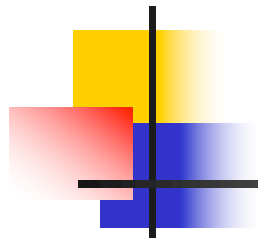
```
Rectangle r1 = new Rectangle(2, 4);  
Rectangle r2 = r1;  
if (r1==r2) ...
```

- Il n'y a pas copie, duplication, il n'y a toujours qu'un seul objet
- Si on modifie `r2`, `r1` l'est également



Cloner des objets

- Il peut être nécessaire de construire une véritable copie de l'objet on parle alors de clone
 - Conserver l'état initial d'un objet avant modification
 - Éviter de fournir à des objets extérieur une référence à un objet sensible

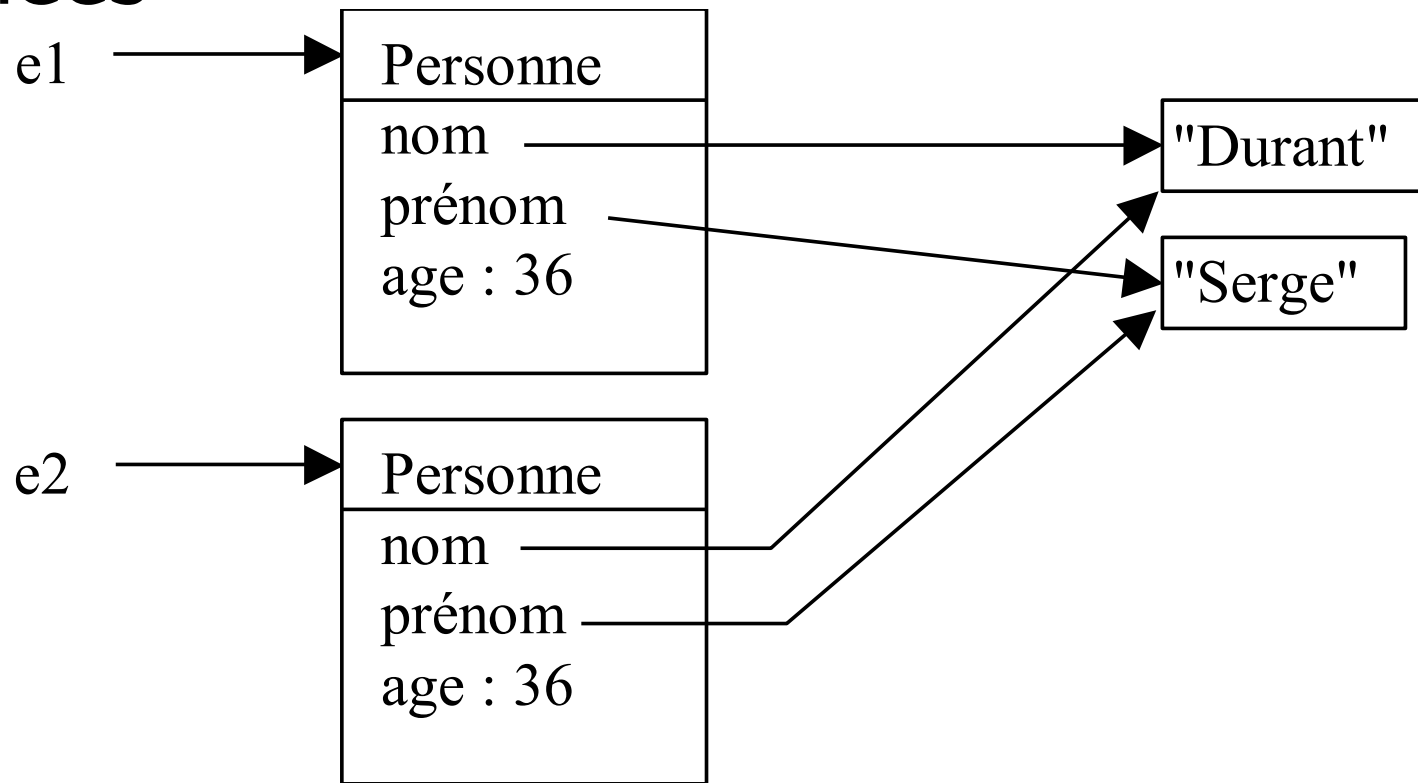


Cloner des objets

- Deux types de clonage
- Dans le cas où les variables membres de l'objet comportent des références à d'autres objets.
 - Surface ou superficielle, ou *shallow copy* : Ces références sont simplement copiées
 - Profondeur ou *deep copy* : Les objets référencés sont également clonés

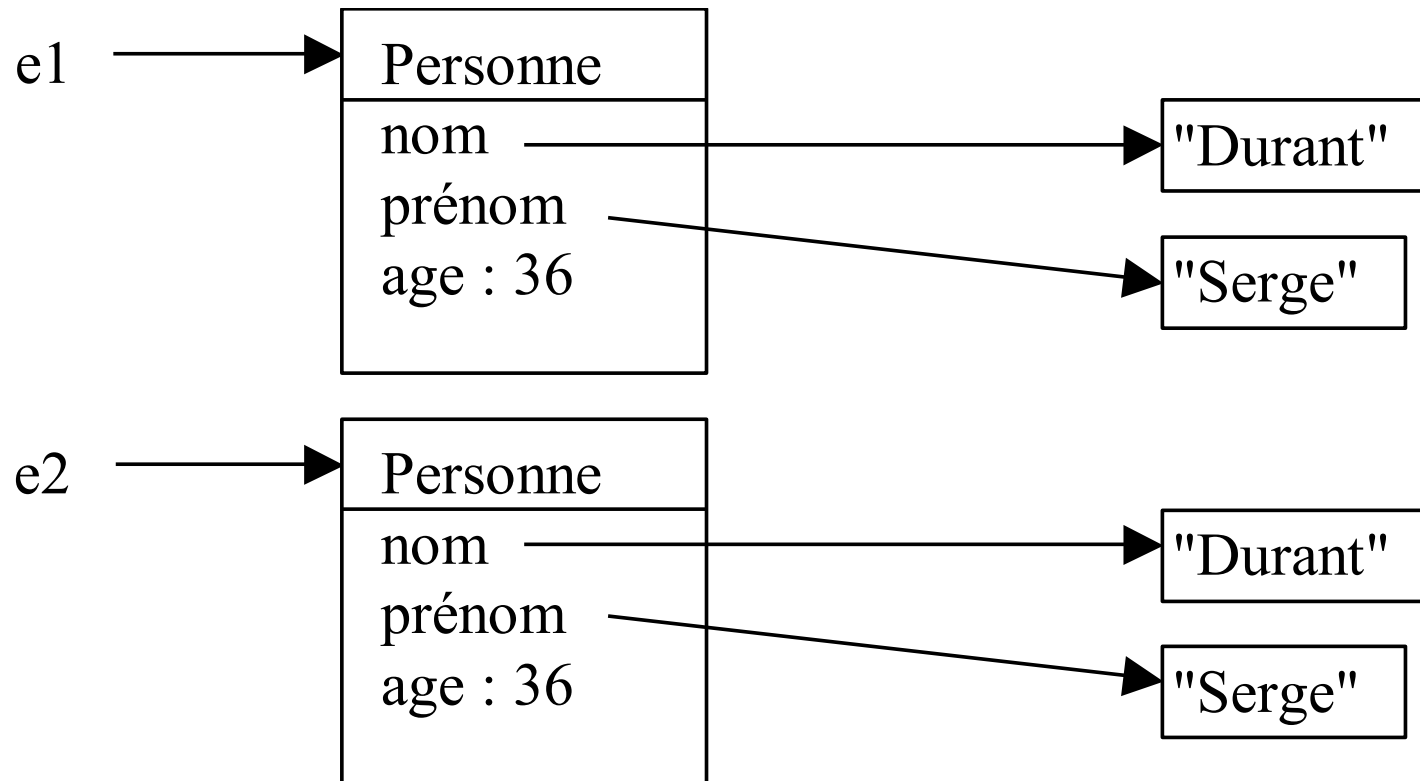
Exemple de clonage de surface, shallow copy

- Les références aux objets sont simplement copiées



Exemple de clonage en profondeur

- Les objets référencés sont également clonés





Copier les objets, la méthode clone()

- Cette méthode issue de la classe `Object` permet de faire une véritable copie de l'objet courant

```
protected Object clone() throws CloneNotSupportedException
```

- Cette méthode est `protected`, une routine extérieure à la classe ne peut l'appeler directement, il nous faut donc créer une sous-classe et la redéfinir
- Par défaut elle effectue un clonage de surface



Copier les objets, l'interface Cloneable

- Un objet ne peut être cloné que s'il est instance d'une classe implémentant l'interface `Cloneable`
- Dans le cas contraire la méthode `clone` de `Object` lève une `CloneNotSupportedException`
- Cette interface est vide, elle a seulement un rôle de marqueur pour indiquer si un objet peut être cloné ou non

```
...  
if (obj instanceof Cloneable)...
```



Comportement de clonage

- Vous avez trois possibilités
 - Décider que le clonage par défaut (clonage de surface) est approprié.
 - Adapter le clonage par défaut en appelant la méthode clone sur les instances rattachées à votre objet (clonage de profondeur)
 - Abandonner la possibilité de clonage, ne pas l'autoriser, donc ne pas implémenter l'interface `Cloneable`



Permettre le clonage

- Votre classe doit alors
 - Implémenter l'interface `Cloneable`
 - Redéfinir la méthode `clone` suivant le comportement souhaité (surface, profondeur)
 - Rendre `public` la méthode `clone()` (pour pouvoir l'invoquer depuis n'importe quelle classe)
 - propager ou gérer les `CloneNotSupportedException` si la copie profonde rencontre un objet non clonable



Exemple de code de clonage profond

```
public class Stack implements Cloneable {
    private Vector items; // code for Stack's methods and
                          // constructor not shown protected
    public Object clone() {
        try {
            // clone the stack
            Stack s = (Stack)super.clone();
            // clone the vector
            s.items = (Vector)items.clone();
            return s; // return the clone
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen because Stack is
            // Cloneable
            throw new InternalError();
        }
    }
}
```




Exemple de code, copie locale

```
import java.util.*;
class MyObject implements Cloneable {
    int i;
    MyObject(int ii) { i = ii; }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("MyObject can't clone");
        }
        return o;
    }
    public String toString() {
        return Integer.toString(i);
    }
}
```

Clonage de
surface

```
public class LocalCopy {  
    static MyObject g(MyObject v) {  
        v.i++;  
        return v;  
    }  
    static MyObject f(MyObject v) {  
        v = (MyObject)v.clone();  
        v.i++;  
        return v;  
    }  
    public static void main(String[] args) {  
        MyObject a = new MyObject(11);  
        MyObject b = g(a);  
        if(a == b) System.out.println("a == b");  
        else System.out.println("a != b");  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        MyObject c = new MyObject(47);  
        MyObject d = f(c);  
        if(c == d) System.out.println("c == d");  
        else System.out.println("c != d");  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
    }  
}
```

**Déroulez ce
code à la main
et afficher le
résultat**



Remarque sur le clonage

- Pendant l'opération de clonage, la méthode `clone()` n'appelle aucun constructeur
- `clone()` crée une copie de l'objet plus rapidement que ne le ferait un appel à `new` et à un constructeur
- La méthode `clone()` retourne toujours un `Object`. Vous devez donc forcer son type de retour



Clonage, cas particulier

- Une classe permet le clonage
- Vous dérivez cette classe, et vous ne voulez pas que votre classe dérivée accepte le clonage
- Redéfinissez la méthode clone et faites lui lever **une** `CloneNotSupportedException`

```
class Subclass extends SuperClass
{
    protected Object clone () throws CloneNotSupportedException
    {
        throw new CloneNotSupportedException ();
    }
}
```



Comment invoquer le clone ?

```
public static void main(String[] args) {  
    SuperClass sc = new SuperClass();  
    SuperClass c_sc;  
    try{  
        c_sc=sc.clone();  
    } catch(CloneNotSupportedException e){  
        System.err.println("Ne peut pas arriver !");  
    }  
}
```



Les classes internes, ou *inner class*

Pour les curieux...

Apparition dans le JDK 1.1

Permet entre autre de définir des classes
anonymes



Définition

- Une classe définie à l'intérieur d'une autre classe
- Comme membre d'une classe (au même titre que ses champs et méthodes)

```
class Englobante{  
    ...  
    class Interne{...}  
    ...  
}
```

- À l'intérieur d'un bloc local (dans une méthode)



Quand utiliser les « inner »?

- Pour renforcer le lien entre deux classes
- Si la classe interne n'a de sens que dans le contexte de sa classe englobante
- Si une instance de cette classe ne doit exister qu'à l'intérieur de sa classe englobante



Pourquoi utiliser les « inner »?

- Un objet d'une classe interne peut accéder à l'implémentation de l'objet qui l'a créé
- Les classes internes peuvent être cachées aux autres classes du même package (c'est la seule possibilité)
- Particulièrement utiles pour la gestion des événements dans les GUI (Nous y reviendrons avec les AWT et Swing)

Exemple de classes internes

```
public class Colis{
    class Contenu{
        private int i = 11;
        public int valeur(){return i;}
    }
    class Destination{
        private String label;
        Destination(String adresse){
            label = adresse;
        }
        String getLabel(){return label;}
    }
    public void transporter(String dest) {
        Contenu c = new Contenu();
        Destination d = new Destination(dest);
    }
    public static void main(String[] args) {
        Colis c = new Colis();
        c.transporter("Tanzanie");
    }
}
```

Un contenu de colis
ou une destination de
colis n'ont de sens que
dans le contexte d'un
colis !



Classes internes comme membre d'instance ou de classe

Définies en lieu et place d'une variable ou
d'une méthode

Inner : d'instance

Nested : de classe (classes imbriquées)

Accessibilité

des classes internes membres

- Même accessibilité que les autres membres en respect avec le modificateur associé à sa déclaration
 - `public`, `protected`, `package` (défaut), `private`
- peuvent être « de classe » : *nested*
 - `static`
- peuvent être abstraites ou finales
 - `abstract`, `final`

Accessibilité

des classes internes membres

- A l'extérieur de la classe englobante on peut citer une classe interne (non déclarée `private`) en la préfixant du nom de sa classe englobante

```
ClasseEnglobante.ClasseInterne
```



Exemple de code : Pile et Maillons

```
class PileEnt {  
    private class Maillon {  
        int elt;  
        Maillon suivant;  
        Maillon(int e, Maillon s){  
            elt=e;  
            suivant=s;  
        }  
    }  
    private Maillon sommet;  
    public PileEnt(){  
        sommet=null;  
    }  
}
```

©Ungaro
Rennes1

```
    public void empiler(int e){  
        sommet=new Maillon(e,sommet);  
    }  
    public void depiler(){  
        sommet=sommet.suivant;  
    }  
    public int sommet(){  
        return sommet.elt;  
    }  
}
```



Propriétés des classes internes

- Accès illimités aux membres des classes
 - La classe interne a accès à tous les membres de sa classe englobante même ceux déclarés `private`
 - La classe englobante a accès à tous les membres de sa ou ses classes internes
- Une classe interne est associée à une instance de sa classe englobante
 - Il y a pratiquement une nouvelle classe interne par objet de sa classe englobante
 - L'instance de la classe englobante est accessible depuis le code de la classe interne par `Englobante.this`
- Pour créer une nouvelle instance de la classe interne il faut passer par une instance existante de la classe englobante

```
Englobante nICE = new Englobante();  
Englobante.Interne nICI = nICE.new Interne();
```



Exemple de code : Pile

```
//Doter la pile d'un moyen de parcours de ses éléments
//(plusieurs parcours sur une même pile doivent être possible)
class PileEnt {
    private int s;
    private int p[] = new int[100];
    public PileEnt() {s=-1;}
    public void empiler(int e){
        s=s+1; p[s]=e;}
    public void depiler() {s=s-1;}
    public int sommet() {return p[s];}
    public class Parcours{
        private int courant;
        public Parcours() {courant=s;}
        public int element() {
            return p[courant];}
        public void suivant(){courant--;}
        public boolean estEnFin(){
            return courant==s;}
    }
}
```

```
PileEnt p= new PileEnt();
...
// deux parcours sur p :
PileEnt.Parcours parc1=
    p.new Parcours();
PileEnt.Parcours parc2=
    p.new Parcours();
parc1.element();
parc1.suivant();
parc2.element();
parc2.suivant();
```

©Ungaro
Rennes1



Propriétés des classes internes

- Une classe interne peut, de la même manière qu'une classe « normale » implémenter une ou plusieurs interfaces (`implements`) et dériver une autre classe (`extends`)
- Une classe interne ne peut avoir le même nom que sa classe englobante
- Une classe interne non `static` ne peut pas déclarer de membres `static` (car elle est liée à une instance)
- Pour importer toutes les classes internes d'une classe utiliser l'instruction `import Englobante.*;`



Reprise de l'exemple précédent avec implémentation d'interface

- Il existe une interface `java.util.Enumeration` qui définit le protocole à suivre pour énumérer tous les éléments d'un ensemble quelconque. Elle définit les deux méthodes
 - `public boolean hasMoreElements()`
 - `public Object nextElement()`
- Permet de définir une boucle de parcours de tous les éléments

```
while (hasMoreElements()) {  
    nextElement()  
}
```



Classe Stack

- On désire modéliser une classe `Stack` pour laquelle deux parcours simultanés et indépendants d'une même pile doivent être possibles
- Pour rendre cela possible on utilise une classe utilitaire `StackEnum` implantée sous la forme d'une classe interne qui implémente l'interface `Enumeration`



Classe Stack et StackEnum

```
public class Stack{
    private Vector items;
    ...//code for Stack's methods and constructors not shown...
    public Enumeration enumerator(){
        return new StackEnum();
    }
    class StackEnum implements Enumeration{
        int currentItem = items.size()-1;
        public boolean hasMoreElements(){
            return (currentItem >= 0);
        }
        public Object nextElement(){
            if (!hasMoreElements()) throw new NoSuchElementException();
            else return items.elementAt(currentItem--);
        }
    }
}
```



Classe interne membre de classe : *nested class*

- Les *nested* classes sont aussi appelées classe imbriquées
- On utilise une classe interne `static` pour cacher une classe dans une autre, sans avoir besoin de fournir à la classe interne une référence à un objet de sa classe englobante
- Si la classe interne est déclarée avec le modificateur `static` elle a accès à toutes les variables `static` de la classe englobante même, les `static private`
- Elle n'a pas par contre accès aux variables d'instances



Exemple classe interne static

- Calcul des valeurs min et max d'un tableau en une seule passe

```
double min = tab[0];  
double max = tab[0];  
for (int i = 1; i<tab.length;i++){  
    if (min>tab[i]) min=tab[i];  
    if (max<tab[i]) max=tab[i];  
}
```

- La méthode doit renvoyer une paire

■ Création d'une classe interne `static Pair` qui stocke deux valeurs

```
public class TabDouble{
    private double tab[];
    ...
    public static class Pair{
        private double first, second;
        public Pair(double f,double s){
            first=f; second=s;
        }
        public double getFirst(){return first;}
        public double getSecond(){return second;}
    }
    ...
    public Pair getMinMax(){
        if (tab.length==0) return null;
        double min = tab[0];
        double max = tab[0];
        for (int i = 1; i<tab.length;i++){
            if (min>tab[i]) min=tab[i];
            if (max<tab[i]) max=tab[i];
        }
        return new Pair(min,max);
    }
}
```

```
//Utilisation de getMinMax
// à l'extérieur de la classe
TabDouble t;
...
TabDouble.Pair p =
    t.getMinMax();
System.out.println("Min : "+
    p.getFirst());
System.out.println("Max : "+
    p.getSecond());
```

Classes internes définies dans un bloc local et classes anonymes



A l'intérieur d'une méthode,
D'un bloc quelconque



Classes internes définies dans une méthode

- Une classe interne peut être définie à l'intérieur d'une définition de méthode
 - Vous avez un problème compliqué à résoudre, vous créez une classe interne pour vous aider (cuisine interne)
 - Vous avez juste besoin de créer une classe interne pour faire retourner un résultat complexe à une de vos méthodes (ne sert qu'une fois, localement)



Exemple : classe interne locale

```
interface Personne{  
    public String identité();  
}
```

```
public class Avion{  
    private Personne[] listePilotes = new Personne(3);  
    private int nb=0; // nbPilotes  
    public Personne creePilote(){  
        class Pilote implements Personne{  
            ...  
            Pilote(String nom,...) {  
                ...  
                if (nb<3) listePilotes[nb++]=this;  
            }  
            public String identité(){...}  
        }  
        return new Pilote("Mermoz",...);  
    } // fin de la méthode creePilote  
    ...  
}
```

La classe n'est utilisable que dans le bloc où elle a été définie



Visibilité des variables pour les classes internes locales

- Elles ont accès à toutes les variables d'instances et de classe de la classe englobante
- Aux paramètres `final` et aux variables locales `final` de la méthode
- Elles n'ont par contre pas accès aux autres variables locales de la méthode



Les classes anonymes

- Ce sont des classes internes locales auxquelles on a pas donné de nom

```
public Personne creePilote() {  
    return new Personne(){// classe anonyme  
        private String nom = "St Exupery";  
        private Date dateNaissance = new Date(3,5,1905);  
        public String identite(){  
            return nom + dateNaissance.toString();  
        }  
    }; //fin de la classe anonyme, attention au ";"  
} // fin de la methode creePilote
```



Classes anonymes

- Elles n'ont pas de constructeurs
 - Les constructeurs doivent avoir le même nom que la classe et la classe n'en a pas !
 - Les paramètres de construction sont donnés au constructeur de la « pseudo » super classe

```
new ClasseMere(listeParamètre){ // définition de la classe}
```

- Il n'y a pas de paramètre dans le cas de l'utilisation d'une interface (comme Personne dans l'exemple prec. Enumeration dans le suiv.)

```
new Interface(){ // définition de la classe}
```

Retour

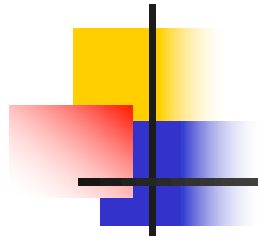
Classe Stack et StackEnum

```
public class Stack{
    private Vector items; .
    ..//code for Stack's methods and constructors not shown...
    public Enumeration enumerator(){
        return new StackEnum();
    }
    class StackEnum implements Enumeration{
        int currentItem = items.size()-1;
        public boolean hasMoreElements(){
            return (currentItem >= 0);
        }
        public Object nextElement(){
            if (!hasMoreElements()) throw new NoSuchElementException();
            else return items.elementAt(currentItem--);
        }
    }
}
```



Retour exemple Stack, classe anonyme

```
public class Stack {  
    private Vector items;  
    ...//code for Stack's methods and constructors not shown...  
    public Enumeration enumerator(){  
        return new Enumeration(){  
            int currentItem = items.size()-1;  
            public boolean hasMoreElements(){  
                return (currentItem >= 0);  
            }  
            public Object nextElement(){  
                if (!hasMoreElements()) throw new NoSuchElementException();  
                else return items.elementAt(currentItem--);  
            }  
        };  
    }  
}
```



Remarque Classes anonymes

- Elles sont utiles dans certains cas pour faciliter l'écriture de certaines méthodes (gestion des événements)
- Par contre elles rendent le code plus difficile à lire et à comprendre
- On utilise généralement des classes anonymes ayant peu de lignes de code



Bibliographie - Webographie

- Tutorial de Sun
 - Object-Oriented Programming Concepts
 - <http://java.sun.com/docs/books/tutorial/java/concepts/index.html>
 - Interfaces and Inheritance
 - <http://java.sun.com/docs/books/tutorial/java/IandI/index.html>
- Cours en ligne Sun Academic Initiative
 - Java Programming Language » : Getting Started (WJ-2751-SE6)
 - Understanding the Building Blocks (WJ-2752-SE6)
- Cours en français
 - Penser en Java (Traduction de ThinkInJava de Bruce Eckel)
 - <http://penserenjava.free.fr/>
 - <http://www.mindview.net/Books/TIJ4>