

# HÉRITAGE ET POLYMORPHISME



# Motivation : Regrouper ces stars dans des classes

2



# Représentation des stars par des classes

3

## Tennisplayer

### Attributs :

- String nom
- String prénom
- int age
- int classement
- float revenu\_annuel

### Méthodes :

public void rencontrer(Star )  
void participerTournoi(String  
titreTournoi)  
.....

## Coach

### Attributs :

- String nom
- String prénom
- int age
- String club
- Int palmares
- float revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void entrainer(String equipe)  
.....

## Footballer

### Attributs :

- String nom
- String prénom
- Int age
- String club
- float revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void jouerMatch(String adversaire)  
.....

## Actor

### Attributs :

- String nom
- String prénom
- int age
- String dernier\_film
- float Revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void jouerRole(String film)  
.....

## Singer

### Attributs :

- String nom
- String prénom
- int age
- String dernier\_album
- float revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void chanter(String titreChanson)  
.....

# Points communs (**colorés en rouge**) entre ces différentes classes

4

## Tennisplayer

### Attributs :

- String nom
- String prénom
- int age
- int classement
- float Revenu\_annuel

### Méthodes :

public void rencontrer(Star )  
void participerTournoi(String titreTournoi)  
.....

## Coach

### Attributs :

- String nom
- String prénom
- int age
- String club
- Int palmarès
- float revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void entraîner(String equipe)  
.....

## Footballer

### Attributs :

- String nom
- String prénom
- int age
- String club
- float Revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
Void jouerMatch(String adversaire)  
.....

## Actor

### Attributs :

- String nom
- String prénom
- int age
- String dernier\_film
- float Revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void jouerRole(String film)  
.....

## Singer

### Attributs :

- String nom
- String prénom
- int age
- String dernier\_album
- float revenu\_annuel

### Méthodes :

public void rencontrer(Star)  
void chanter(String titreChanson)  
.....

# Problèmes de duplication du code et de maintenance

5

L'implémentation de ces classes à l'aide d'un langage de programmation évoquera plusieurs problèmes dont:

- ❑ La duplication du code à plusieurs reprises.
- ❑ Perte du temps.
- ❑ Problèmes de maintenance.

# Problème de maintenance (exemple)

6

→ **Opération de maintenance** : Changer le type de l'attribut `revenu_annuel` par double au lieu de float.

## Tennisplayer

### Attributs :

- String nom
- String prénom
- int age
- int classement
- double **revenu\_annuel**

### Méthodes :

public void rencontrer(Star)  
void participerTournoi(String titreTournoi)

## Coach

### Attributs :

- String nom
- String prénom
- int age
- String club
- Int palmares
- double **revenu\_annuel**

### Méthodes :

public void rencontrer(Star)  
void entrainer(String equipe)

.....

## Footballer

### Attributs :

- String nom
- String prénom
- int age
- String club
- double **Revenu\_annuel**

### Méthodes :

public void rencontrer(Star)  
void jouerMatch(String adversaire)  
.....

## Actor

### Attributs :

- String nom
- String prénom
- int age
- String dernier\_film
- double **revenu\_annuel**

### Méthodes :

public void rencontrer(Star)  
void jouerRole(String film)

.....

## Singer

### Attributs :

- String nom
- String prénom
- int age
- String dernier\_album
- double **revenu\_annuel**

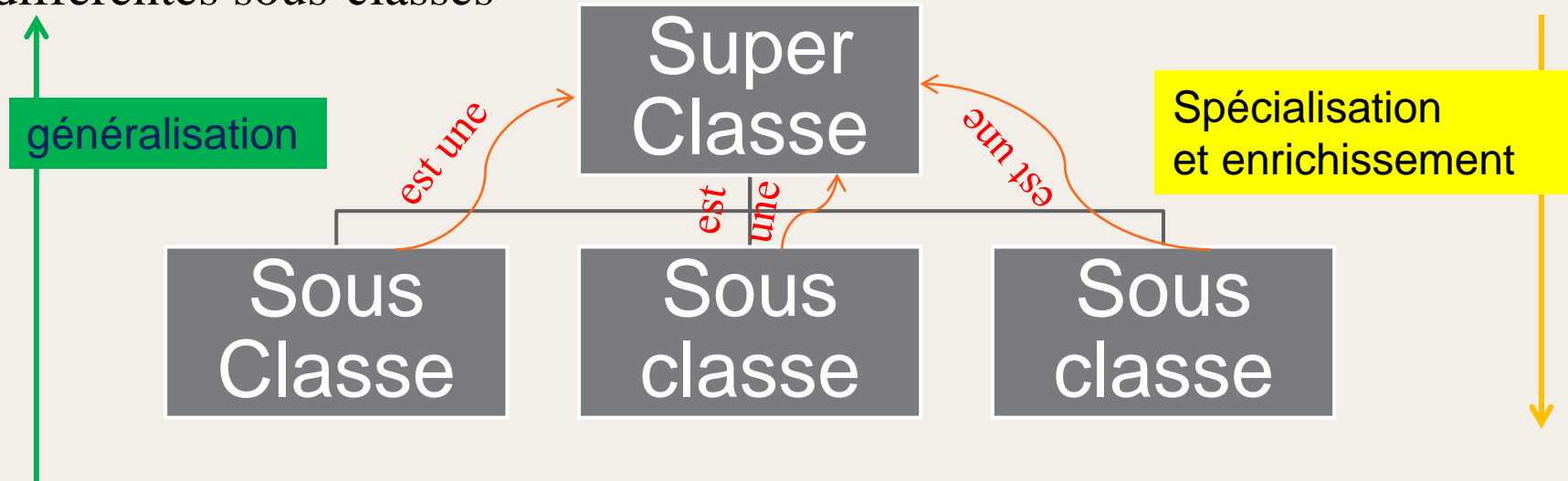
### Méthodes :

public void rencontrer(Star)  
void chanter(String titreChanson)  
.....

# Solution : L'héritage

7

- ❑ L'héritage est l'un des piliers de la programmation orientée objet.
- ❑ La notion de l'héritage représente la relation « **est un(e)** ».
- ❑ Il permet de créer des classes spécialisées appelées **sous-classes** à partir de classes plus générales déjà existante, appelées **super-classes**
- ❑ Une sous-classe « est une » super-classe. L'inverse est incorrecte.
- ❑ La super-classe encapsule les attributs/méthodes communs entre les différentes sous-classes





# Représentation des classes après application de l'héritage

8

Super  
classe

**Star**

**Attributs :**

- String nom
- String prénom
- int age
- double revenu\_annuel

**Méthodes :**

public void rencontrer(Star unStar)

**Coach**

**Attributs :**

- String club
- Int palmares

**Méthodes :**

void entraîner(String  
equipe)  
.....

Sous classes

**Tennisplayer**

**Attributs :**

- int classement

**Méthodes :**

void  
participerTournoi(String  
titreTournoi)  
.....

**Actor**

**Attributs :**

- String dernier\_film

**Méthodes :**

void jouerRole(String film)

**Singer**

**Attributs :**

- String dernier\_album

**Méthodes :**

void chanter(String  
titreChanson)  
.....

**Footballer**

**Attributs :**

- String club

**Méthodes :**

void jouerMatch(String  
adversaire)  
.....



# Principe de l'héritage

9

Lorsque une sous **classe B** est créée à partir d'une **super-classe A** :

- Le type est hérité : **B** est un **A**
- B va hériter de A l'ensemble :
  - des attributs de A
  - des méthodes de A (Sauf les constructeurs)

```
A A1 = new A(.....);  
B B1 = new B(.....);  
A1=B1 ;
```

Remarque : Les attributs et méthodes de A vont être disponible pour B sans que l'on ait besoin de les redéfinir explicitement dans B.

Par ailleurs :

- Des attributs et/ou méthodes supplémentaires peuvent être définis par la sous classe B : **enrichissement**
- Méthodes héritées de A peuvent être redéfinis dans B : **spécialisation**

# Exemple

10

## Star

### Attributs :

- String nom
- String prénom
- int age
- double Revenu\_annuel

### Méthodes :

public void rencontrer(Star unStar)

```
Star S1 = new Star(.....);  
Star S2 = new Actor(.....);  
Tennisplayer S3 = new Tennisplayer(.....);  
S1=S2;  
S2.rencontrer(S3);  
S2.age=25;
```

## Tennisplayer

### Attributs :

- int classement

### Méthodes :

void participerTournoi(String titreTournoi)  
.....

## Actor

### Attributs :

- String dernier\_film

### Méthodes :

Void jouerRole(String film)  
.....

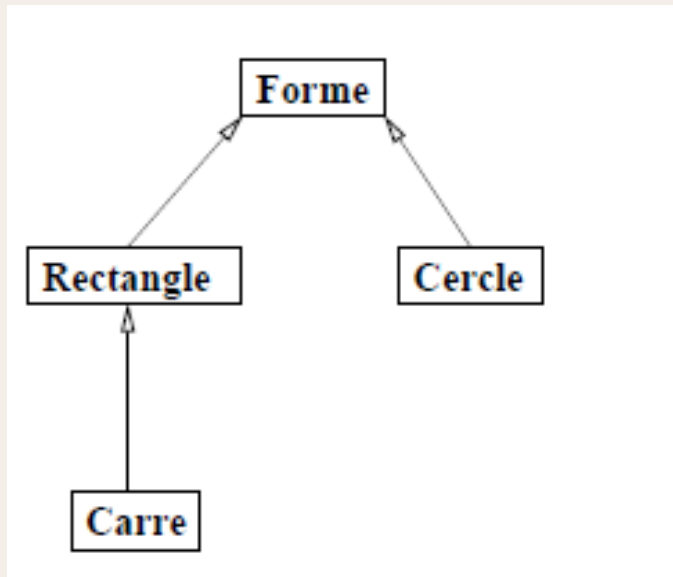
# Transitivité dans l'héritage

11

Par transitivité, les instances d'une sous-classe possèdent :

- ❑ Les attributs et méthodes(hors constructeurs) de l'ensemble des classes parentes(super-classe, super-super-classe, etc.) **de l'hérarchie de classe.**

Exemple :



- ❑ L'instance de la classe **Carre** hérite les attributs et méthodes (sauf les constructeurs) de la classe **Rectangle** et de la classe **Forme**.

# Héritage : syntaxe java

12

Définition d'une sous classe sous java :

Syntaxe :

```
Class Nomsousclasse extends NomSuperClasse
```

```
{
```

```
/*Déclaration des attributs et méthodes spécifiques à  
la sous-classe*/
```

```
}
```

# Synthèse : super classe et sous classe

13

## Une super-classe :

- Est une classe « parente »
- Déclare les attributs/méthodes communs
- Peut avoir plusieurs sous-classes

## Une sous-classe est :

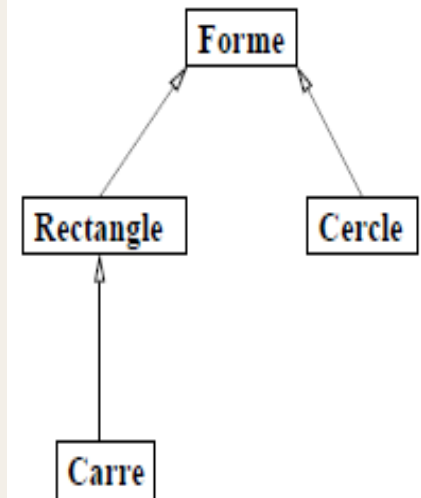
- Une classe « enfant »
- Étend une seule super-classe
- Hérite des attributs, des méthodes et du type de la super-classe

**Remarque** : Un attribut/une méthode hérité(e) peut s'utiliser comme si il/elle était déclaré(e) dans la sous-classe de la super-classe (**en fonction des droits d'accès**).

# Exemple

14

```
public class Rectangle extends Forme {  
  
    private int largeur;  
    private int longueur;  
  
    public Rectangle(int x, int y) {  
        this.largeur = x;  
        this.longueur = y;  
    }  
  
    public int getLargeur() {  
        return this.largeur;  
    }  
  
    public int getLongueur() {  
        return this.longueur;  
    }  
  
    public int surface() {  
        return this.longueur * this.largeur;  
    }  
  
    public void affiche() {  
        System.out.println("rectangle " + longueur + "x" + largeur);  
    }  
}
```



# Héritage et la classe **Object**

15

- ❑ En Java, toutes les classes héritent de `java.lang.Object`, directement ou indirectement.
- ❑ **Directement** : si l'on déclare une classe sans héritage, le compilateur rajoute `extends Object`.
- ❑ **Indirectement** : si l'on hérite d'une classe celle-ci hérite de `Object` (directement ou indirectement)



# Classe Object et ses méthodes

16

```
public class Object {  
    public Object() {...} // constructeur  
  
    public String toString() {...}  
  
    protected native Object clone() throws CloneNotSupportedException {...}  
  
    public equals(java.lang.Object) {...}  
    public native int hashCode() {...}  
  
    protected void finalize() throws Throwable {...}  
  
    public final native Class getClass() {...}  
  
    // méthodes utilisées dans la gestion des threads  
    public final native void notify() {...}  
    public final native void notifyAll() {...}  
  
    public final void wait(long) throws InterruptedException {...}  
    public final void wait(long, int) throws InterruptedException {...}  
}
```

# Problème

17

**Classe\_A**

Private double x;

.....



**Classe\_B**

double y;

.....;

public boolean m()

{  
If x>1 && y>1  
return true;

.....

}

.....

Erreur de compilation, car l'accès à l'attribut x se fait juste dedans sa classe A

Solution : pour résoudre ce problème, on doit utiliser les méthodes getters et setters prévus dans la super classe.

# Rappel : Modificateurs d'accès

18

Modificateur du membre	<code>private</code>	<code>aucun</code>	<code>protected</code>	<code>public</code>
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui

# Rappel : droit d'accès protected

19

**Classe\_A**

**protected** Double x;

.....

Cette fois l'accès à l'attribut x n'engendrera pas une erreur de compilation

**Classe\_B**

private double y;

.....;

Public boolean m()

{

If x>1 && y>1

return true;

.....

}

.....

# Synthèse

20



- ❑ Une sous-classe n'a pas de droit d'accès aux membres (attributs ou méthodes) privés hérités de ses super-classes. Elle doit alors utiliser les getters/setters prévus dans la super-classe.
- ❑ L'utilisation du modificateur **protected** dans la super classe n'offrira pas seulement l'accès aux sous classe mais aussi aux classes appartenant au même package de la super classe.
- ❑ La définition d'attributs protégés (**protected**) nuit à une bonne encapsulation.
- ❑ Les attributs protégés (**protected**) sont d'un usage peu recommandé en java.

# Classe finale

21

Si une classe est déclarée **final**, il est impossible de l'étendre par des sous-classes

```
1 package com.javaguides.corejava.keywords.finalkeyword;
2
3 final class Person {
4     private String firstName;
5     private String lastName;
6     public String getFirstName() {
7         return firstName;
8     }
9     public void setFirstName(String firstName) {
10         this.firstName = firstName;
11     }
12     public String getLastName() {
13         return lastName;
14     }
15     public void setLastName(String lastName) {
16         this.lastName = lastName;
17     }
18 }
19
20 class Employee extends Person {
21     |
22 }
```

 The type Employee cannot subclass the final class Person  
1 quick fix available:  
 [Remove 'final' modifier of 'Person'](#)

Press 'F2' for focus

# Reprenons le premier exemple

22

Super  
classe

**Star**

**Attributs :**

- String nom
- String prénom
- int age
- double Revenu\_annuel

**Méthodes :**

public void rencontrer(Star unStar)

Sous classes

**Tennisplayer**

**Attributs :**

- int classement

**Méthodes :**

void  
participerTournoi(String  
titreTournoi)  
.....

**Actor**

**Attributs :**

- String dernier\_film

**Méthodes :**

void jouerRole(String film)

**Singer**

**Attributs :**

- String dernier\_album

**Méthodes :**

void chanter(String  
titreChanson)  
.....

**Footballer**

**Attributs :**

- String club

**Méthodes :**

void jouerMatch(String  
adversaire)  
.....



# Problème : comportement différent d'une méthode héritée par les sous classes

23

Pour un star non acteur, il invoque la méthode rencontrer pour **prendre café** avec un autre Star

```
public void rencontrer(Star unStar)
{
    prendreCafé(leStar);
}
```

Pour un star acteur (Actor), il invoque la méthode rencontrer pour **prendre selfie** avec un autre Star:

```
Public void rencontrer(Star unStar){
    prendreSelfie(leStar);
}
```

Faut-il reconcevoir toute la hiérarchie?

**Réponse :** Non, on ajoute simplement la méthode spéciale `Public void rencontrer(Star unStar)` qui permet de prendre selfie dans la sous-classe Actor.

# Solution : Redéfinition

24

Super  
classe

**Star**

**Attributs :**

- String nom
- String prénom
- int age
- double Revenu\_annuel

**Méthodes :**

**public void rencontrer(Star unStar)**

**Coach**

**Attributs :**

- String club
- Int palmares

**Méthodes :**

**void entraîner()**  
.....

Sous classes

**Tennisplayer**

**Attributs :**

- int classement

**Méthodes :**

**void**  
**participerTournoi(String**  
**titreTournoi)**  
.....

**Actor**

**Attributs :**

- String dernier\_film

**Méthodes :**

**Public void rencontrer(Star UnStar)**  
**void jouerRole(String film)**

**Singer**

**Attributs :**

- String dernier\_album

**Méthodes :**

**void chanter(String**  
**titreChanson)**  
.....

**Footballer**

**Attributs :**

- String club

**Méthodes :**

**void jouerMatch(String**  
**adversaire)**  
.....

Redéfinition de la méthode **rencontrer(Star Un Star)** dans la sous classe Actor pour l'adapter au comportement des acteurs (prendre selfie).

# Redéfinition dans une hiérarchie de classes

25

- ❑ Redéfinition : pour les méthodes « overriding ».
- ❑ Redéfinition : une méthode déjà définie dans une super-classe a une nouvelle définition dans une sous-classe.
- ❑ La même méthode peut être redéfinie sur plusieurs niveaux dans une hiérarchie de classes.
- ❑ La redéfinition est très courante pour **les méthodes**.

# Exemple

26

```
public class Rectangle extends Forme {  
  
    private int largeur;  
    private int longueur;  
  
    public Rectangle(int x, int y) {  
        this.largeur = x;  
        this.longueur = y;  
    }  
  
    public int getLargeur() {  
        return this.largeur;  
    }  
  
    public int getLongueur() {  
        return this.longueur;  
    }  
  
    public int surface() {  
        return this.longueur * this.largeur;  
    }  
  
    public void affiche() {  
        System.out.println("rectangle " + longueur + "x" + largeur);  
    }  
}
```

```
public class Carre extends Rectangle {  
  
    public Carre(int cote) {  
        super(cote, cote);  
    }  
  
    public void affiche() {  
        System.out.println("carré " + this.getLongueur());  
    }  
}
```

Redéfinition de la méthode  
affiche dans la sous classe  
Carre

On doit afficher «carré» au moment du calcul de la surface d'une instance de la sous classe Carre .  
La solution est de redéfinir la méthode affiche() de la classe Rectangle.

# Masquage

27

- ❑ Comme pour les méthodes, les attributs d'une super-classe peuvent être déclarer une autre fois dans les sous-classes. On appelle cette opération le **masquage**.
- ❑ Masquage (shadowing) : un identificateur d'attribut de la sous-classe qui en cache un autre de la super-classe.
- ❑ Le masquage est peu courant.

# Exemple de masquage de l'attribut

28

```
public class Cercle extends FormeGeometrique{  
    final double rayon;  
    Color couleur;  
  
    public String toString() {  
        ...  
        System.out.printf("%d", super.couleur);  
        ...  
    }  
}
```

L'attribut **couleur de type Color** de la classe Cercle masque l'attribut **couleur de type int** de la classe FormeGeometrique

```
public class FormeGeometrique{  
    int couleur;  
    Point centre;  
  
    public int getColor() {  
        return this.couleur;  
    }  
}
```

# Accès à une méthode redéfinie ou attribut masqué

29

- ❑ Cette fois, l'acteur (Actor) ne souhaite pas seulement prendre un selfie avec un autre star mais également prendre café.
- ❑ Comment appeler la méthode générique `public void rencontrer(Star)` alors qu'elle a été déjà redéfinie dans la sous-classe `Acteur(Actor)`?

**Solution** : le mot clé `super`.

- ❑ La notation `super` permet d'avoir accès aux membres(attributs et méthodes) **non static** de la superclasse.
- ❑ `super` ne s'utilise que dans la sous-classe.

Pour un star acteur (Actor) :

```
Public void rencontrer(Star unStar){  
    super.rencontrer(unStar); //invoquer la méthode générique de la super classe  
    prendreSelfie(unStar); //invoquer la méthode redéfinie de la sous classe  
}
```



# Méthode finale

30

- ❑ Si une méthode est déclarée final, celle-ci ne peut être redéfinie
  - ❑ Important en terme de sécurité.

```
29 class UNIX {
30
31     protected final void whoAmI () {
32         System.out.println("I am UNIX");
33     }
34
35 }
36
37 class Linux extends UNIX {
38
39     public void whoAmI () {
40         System.out.println("I am Linux");
41     }
42 }
```

Cannot override the final method from UNIX

1 quick fix available:

[Remove 'final' modifier of 'UNIX.whoAmI'\(..\)](#)

Press 'F2' for focus

# Constructeurs et héritage

31

- ❑ Lors de l'instanciation d'une sous-classe, il faut initialiser :
  - Les attributs propres à la sous-classe.
  - Les attributs hérités des super-classes.
- ❑ Mais il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui même l'initialisation des attributs hérités.
- ❑ L'accès à ces attributs pourraient notamment être interdit (**private**).

Solution : l'initialisation des attributs hérités doit se faire en invoquant **les constructeurs des super-classes**.

# Constructeurs et héritage

32

- ❑ L'invocation du constructeur de la super-classe se fait au début du corps du constructeur de la sous-classe au moyen du mot clé `super`.

Syntaxe :

```
SousClasse(liste de paramètres )
```

```
{
```

```
    Super(Arguments);
```

```
//Arguments : liste d'arguments attendus par un constructeur de la  
super-classe de sous-classe
```

```
}
```

- ❑ Lorsque la super-classe admet un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe **n'est pas obligatoire**.
- ❑ Le compilateur se charge de réaliser l'invocation du constructeur par défaut.

# Exemple

33

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel(int x, int y) {  
        this.x = newX;  
        this.y = newY;  
    }  
    // ...  
}
```

```
public class ColoredPixel extends Pixel {  
    private byte[] rgb;  
    public ColoredPixel() {  
        // super(); // Constructeur Pixel() is undefined  
        super(0,0); // OK; notez que x et y sont private!  
        rgb = new byte[3];  
    }  
}
```

# Exemple : sous-classe sans ses propres attributs

34

```
public class Rectangle extends Forme {  
  
    private int largeur;  
    private int longueur;  
  
    public Rectangle(int x, int y) {  
        this.largeur = x;  
        this.longueur = y;  
    }  
  
    public int getLargeur() {  
        return this.largeur;  
    }  
  
    public int getLongueur() {  
        return this.longueur;  
    }  
  
    public int surface() {  
        return this.longueur * this.largeur;  
    }  
  
    public void affiche() {  
        System.out.println("rectangle " + longueur + "x" + largeur);  
    }  
}
```



```
public class Carre extends Rectangle {  
  
    public Carre(int cote) {  
        super(cote, cote);  
    }  
  
    public void affiche() {  
        System.out.println("carré " + this.getLongueur());  
    }  
}
```

# Synthèse

35

1. Chaque constructeur d'une sous-classe doit appeler `super(...)`.
2. Les arguments fournis à `super` doivent être ceux d'au moins un des constructeurs de la super-classe.
3. L'appel doit être la toute 1<sup>re</sup> instruction.
4. Erreur si l'appel vient plus tard ou 2 fois.
5. Aucune autre méthode ne peut appeler `super(...)` pour initialiser le constructeur de la super classe.

## Remarque :

Si on oublie l'appel à `super(...)` alors appel automatique à `super()`. Pratique parfois, mais erreur si le constructeur par défaut n'existe pas.

# Problème

36

## Code1

```
Public static void main(String[] args) {  
    Star leStar = new Star(...);  
    Tennisplayer T1=new Tennisplayer();  
    Actor A1=new Actor();  
    Footballer F1=new Footballer();  
    Singer S1=new Singer();  
    T1.renconter(leStar);  
    A1.renconter(leStar);  
    F1.renconter(leStar);  
    S1.renconter(leStar);  
}
```

**Question :** comment procéder de telle façon à optimiser le code1?



# Sous typage

37

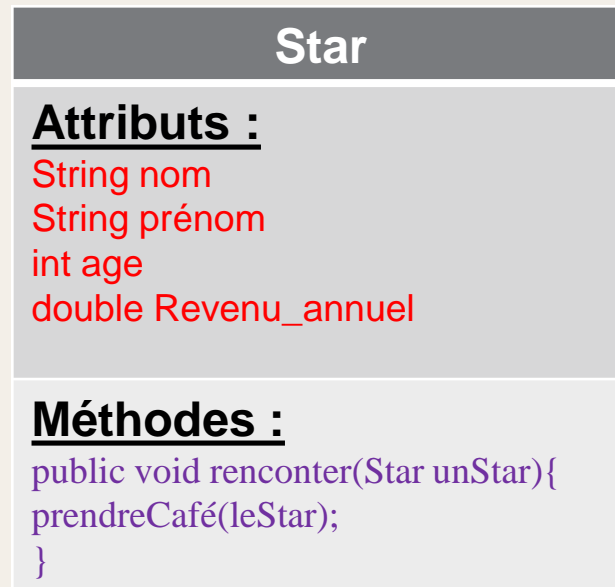
**réponse** : Le sous-typage permet de réutiliser un code (générique) qui a été écrit en typant les références avec un super-type et en appelant le code avec un sous-type.

Syntaxe : **TypeSuperclass** reference = new sousclass(...)

Code1 : Typage réel	Code2: Sous Typage
<pre>Public static void main(String[] args) {   Star leStar = new Star(...);   Tennisplayer T1=new Tennisplayer();   Actor A1=new Actor();   Footballer F1=new Footballer();   Singer S1=new Singer();   T1.renconter(leStar);   A1.renconter(leStar);   F1.renconter(leStar);   S1.renconter(leStar); }</pre>	<pre>Public static void main(String[] args) {   Star leStar = new Star(...);   <b>Star [] stars = new Star[4];</b>   <b>stars[0]=new Tennisplayer();</b>   <b>stars[1]=new Actor();</b>   <b>stars[2]=new Footballer();</b>   <b>stars[3]=new Singer();</b>   <b>for (int i=0; i&lt;stars.length; i++) {</b>   <b>stars[i].rencontrer(leStar); }</b> }</pre>

Problème : Quelle méthode sera invoqué par l'instruction `stars[1].rencontrer(leStar)` ?

38



```
Public static void main(String[] args) {  
    Star leStar = new Star(...);  
    Star [] stars = new Star[4];  
    stars[0]=new Tennisplayer();  
    stars[1]=new Actor();  
    stars[2]=new Footballer();  
    stars[3]=new Singer();  
    For (int i=0; i<stars.length; i++) {  
        stars[i].rencontrer(leStar);  
    }  
}
```

**Tennisplayer**

**Attributs :**  
int classement

**Méthodes :**  
void  
participerTournoi(String  
titreTournoi)  
.....

**Actor**

**Attributs :**  
String dernierFilm

**Méthodes :**  
public void rencontrer(Star unStar){  
    prendreSelfie(leStar);  
}  
void jouerRole(String film)

**Singer**

**Attributs :**  
String dernierAlbum

**Méthodes :**  
void chanter(String  
titreChanson)  
.....

**Footballer**

**Attributs :**  
String club

**Méthodes :**  
void jouerMatch(String  
adversaire)  
.....

# Solution : Polymorphisme par sous typage

39

- ❑ Le polymorphisme par sous typage 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.
- ❑ 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.

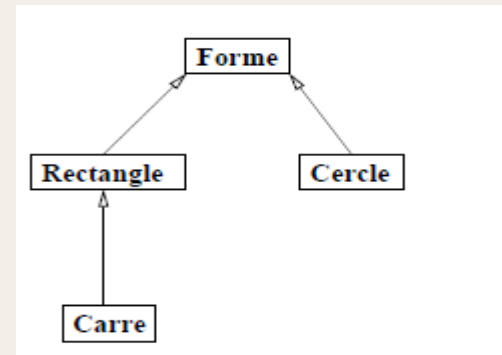
# Exemple de polymorphisme par sous typage

40

L'opérateur **instanceof** peut être utilisé pour tester l'appartenance d'un objet(instance) à une classe comme suit

```
Forme[] tableau = new Forme[4];  
tableau[0] = new Rectangle(10,20);  
tableau[1] = new Cercle(15);  
tableau[2] = new Rectangle(5,30);  
tableau[3] = new Carre(10);
```

```
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 Cercle)  
        System.out.println("element " + i + " est un cercle");  
    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 carré");  
}
```



**Résultat:**

element[0] est une forme

element[0] est un rectangle

element[1] est une forme

element[1] est un cercle

element[2] est une forme

element[2] est un rectangle

element[3] est une forme

element[3] est un rectangle

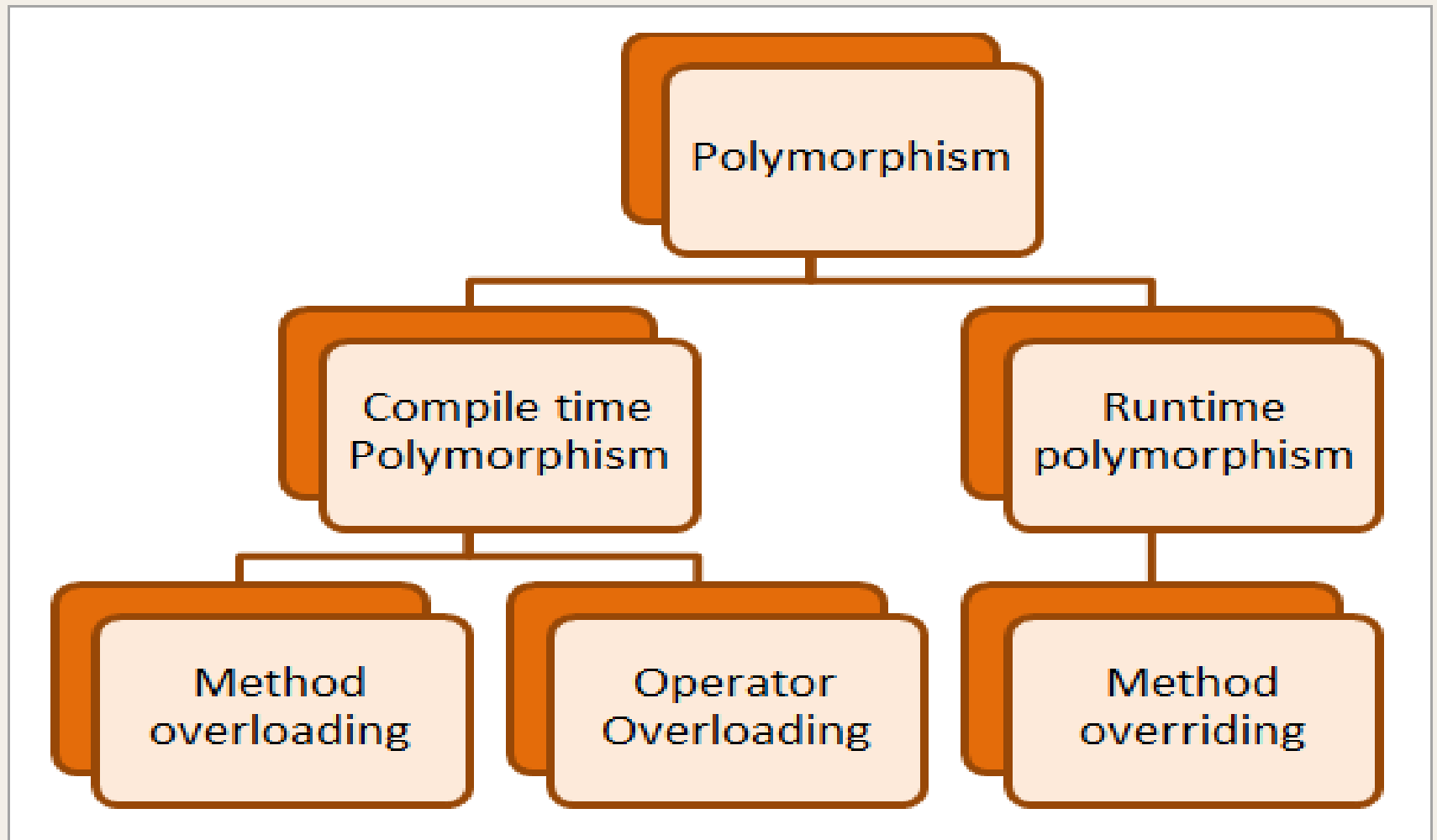
element[3] est un carré

# Polymorphisme de surcharge

41

- ❑ La **surcharge** correspond à avoir des méthodes de même nom mais des entêtes différents dans une même classe.
- ❑ À ne pas confondre avec la redéfinition.
- ❑ Les méthodes comme les constructeurs peuvent être surchargées (overloaded) :
  - Leur nom est le même;
  - Le nombre ou le type de leurs paramètres varie.

```
public class A {  
    public void m(int a) {...}    // surcharge  
    public void m(double a) {...} // surcharge  
}  
  
public class B extends A {  
    public void m(long a) {...}    // surcharge  
    public void m(double a) {...} // redéfinition  
}
```





Merci de votre attention