



ECOLE MAROCAINE DES
SCIENCES DE L'INGENIEUR

Membre de
HONORIS UNITED UNIVERSITIES

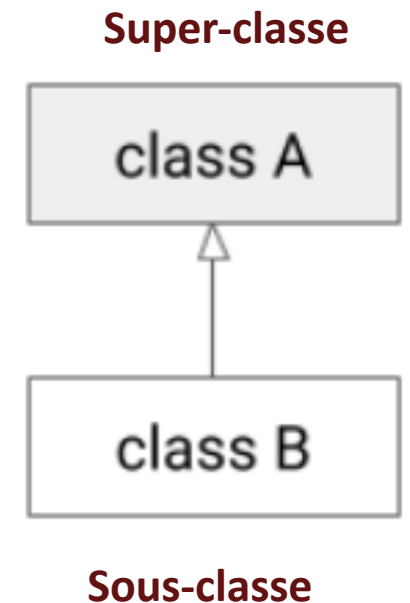


HÉRITAGE EN JAVA

3IIR - POO2

HÉRITAGE

- L'héritage est un mécanisme par lequel une classe peut hériter des attributs et méthodes d'une autre classe appelée **classe parente** ou **super-classe**.
 - La super-classe définit des attributs et comportements communs. Elle représente un type parent.
- Une classe qui hérite d'une **super-classe** est appelée **classe enfant** ou **sous-classe**.
 - La sous-classe représente un type plus spécifique (sous-type) . On dit qu'elle étend le type parent.
 - La sous-classe définit les attributs et comportements spécifiques au sous-type.

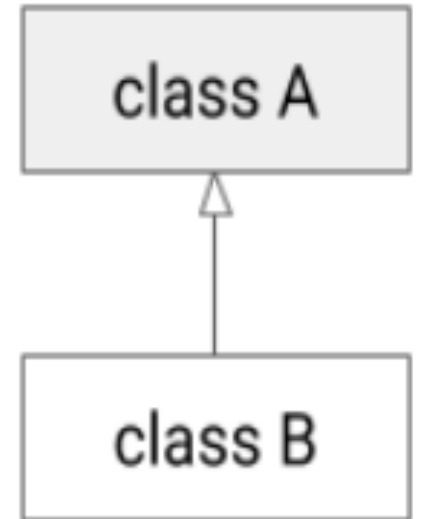


HÉRITAGE : SYNTAXE

- L'héritage en Java correspond à **l'héritage simple et public**.
 - *Une classe ne peut hériter directement que d'une seule classe .*
- Le mot clé **extends** précise la super-classe qui doit être étendue.

```
public class A {  
    ...  
}
```

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



HÉRITAGE

- Une classe déclarée **final** ne peut pas être héritée.

```
final class A {  
}
```

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



- Une classe **sealed** peut être héritée, mais uniquement par certaines classes définies avec **permits**.

```
sealed class Animal permits Chat, Chien{  
}
```

```
class Loup extends Animal {  
}
```



- Une classe qui hérite d'une **sealed** class doit être déclarée **final**, **sealed**, ou **non-sealed**.

```
final class Chien extends Animal {  
}
```



ACCÈS AUX MEMBRES HÉRITÉS

- La sous classe ne peut pas accéder aux membres **privés** de la super classe.

```
package exemples;  
public class A  
{  
    int i;  
    private int j;  
}
```



- A et B se trouvent dans le même package.*
- B hérite de i et j*

```
package exemples;  
class B extends A {  
    int total;  
    void somme()  
    {  
        total = i + j;  
    }  
}
```




ACCÈS AUX MEMBRES HÉRITÉS

Dans le même package , un membre **protected** d'une super classe, se comporte comme un membre avec l'accès par défaut.

```
package exemples;
public class A
{
    int i;
    protected int j;
}
```

```
package exemples;
class B extends A {
    int total;
    void somme()
    {
        total = i + j;
    }
}
```

 Sous classe de A

```
package exemples;
class C {
    int k;
    void meth()
    {
        A oA = new A();
        B oB = new B();
        k = oA.j + oB.j;
    }
}
```


 Classe du même package que A

ACCÈS AUX MEMBRES HÉRITÉS

À l'extérieur du package , un membre **protected** d'une classe A, n'est accessible qu'aux classes dérivées de A.

```
package exemples;
public class A
{
    int i;
    protected int j;
}
```

```
package exemples1;
import exemples.A;
class B extends A {
    int total;
    void somme()
    {
        total = i + j;
    }
}
```

 Sous classe de A

```
package exemples1;
import exemples.*;
class C {
    int k;
    void meth(int a)
    {
        A oA = new A();
        B oB = new B();
        k = oA.j + oB.j;
    }
}
```

 Classe d'un autre package que A

CONSTRUCTEUR SANS ARGUMENTS

Super-classe

```
public class Animal
{
    private String nom;
    public Animal()
    {
        System.out.println ("Animal créé");
    }
}
```

Sous-classe

```
class Chien extends Animal
{
    private String couleur;
    public Chien()
    {
        System.out.println ("Chien créé");
    }
}
```

- *Si la super classe contient un constructeur sans arguments (par défaut ou défini explicitement) , alors le constructeur sans arguments de la sous-classe invoque implicitement celui de la super-classe.*


CONSTRUCTEUR SANS ARGUMENTS

```
public class Main
{
    public static void main (String[] args)
    {
        Chien Fox = new Chien();
    }
}
```

Sortie :

Animal créé

Chien créé



Le constructeur sans arguments de la super-classe est appelé implicitement avant celui de la sous-classe.

CONSTRUCTEUR AVEC ARGUMENTS

```
public class Animal
{ ...
    public Animal(String nom)
    {
        this.nom = nom;
        System.out.println("Animal :" + nom);
    }
}
```

```
class Chien extends Animal {
    ...
    public Chien(String nom, String color)
    {
        super(nom);
        couleur = color;
        System.out.println("Chien :" + couleur);
    }
}
```

- Le constructeur avec arguments d'une sous-classe doit invoquer explicitement le constructeur avec arguments de la super-classe.
 - Cela se fait en utilisant le **mot clé super** avec la signature de constructeur correspondante.
 - Cela doit être fait **dans la première ligne de code** du constructeur de la sous-classe.
- *Si vous n'appellez pas explicitement le constructeur avec arguments de la super-classe, ce sera son constructeur sans arguments qui sera appelé implicitement.*

CONSTRUCTEUR AVEC ARGUMENTS

```
public class Main
{
    public static void main (String[] args)
    {
        Chien F= new Chien("Fox","noir");
    }
}
```

Sortie:

Animal : Fox

Chien : noir

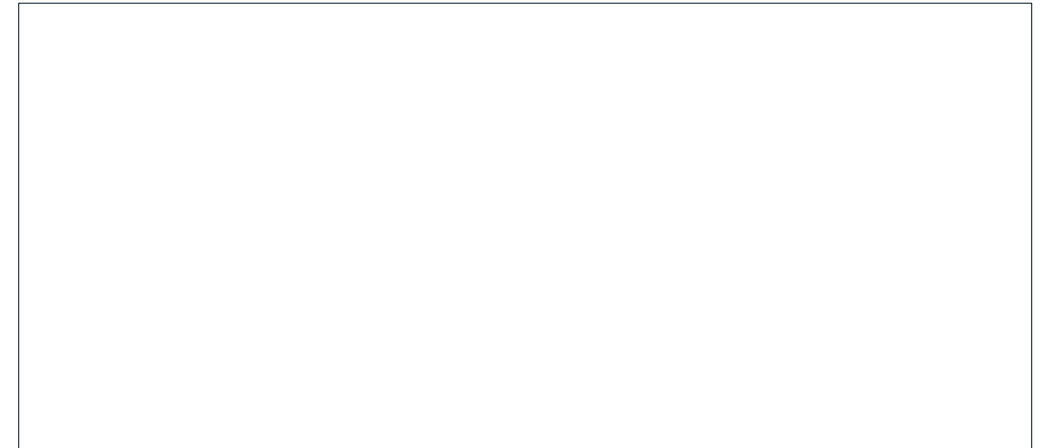
Le constructeur avec arguments de la super-classe est appelé avant celui de la sous-classe.

CONSTRUCTEURS

```
public class A
{
    int i;
    public A(int i)
    {
        System.out.println("constructeur de A");
        this.i = i;
    }
}

class B extends A {
    public static void main(String[] argv) {
        new B();
    }
}
```

Quelle est la sortie du programme ?

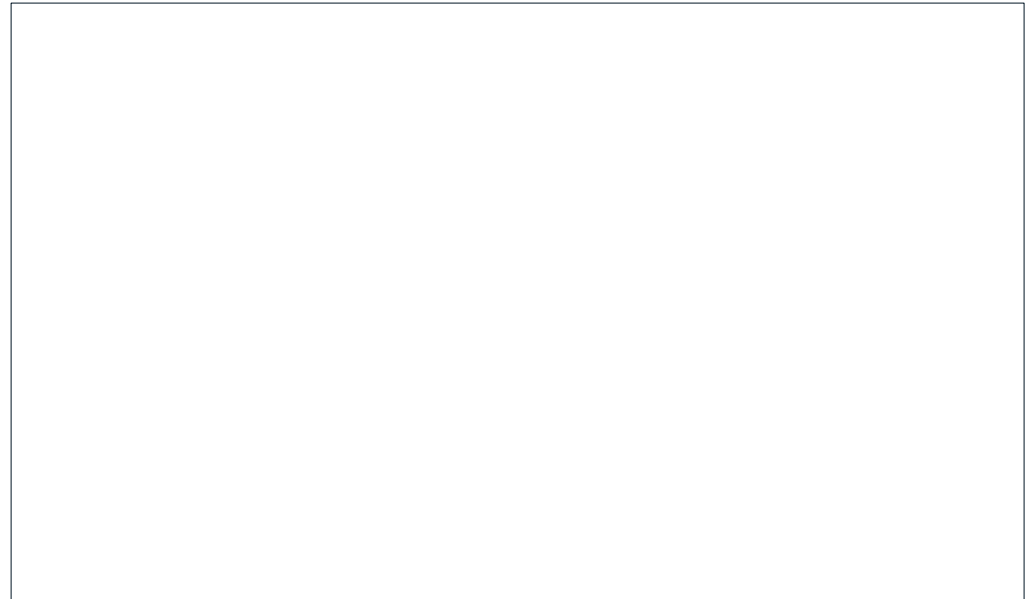


CONSTRUCTEURS

```
public class A {  
    public A() {  
        System.out.println("constructeur de A");  
    }  
}  
  
public class B extends A {  
    public B(){  
        System.out.println("constructeur de B");  
    }  
    public B(int r) {  
        this();  
        System.out.println("autre constructeur de B");  
    }  
}
```

```
class C extends B {  
    public static void main(String[] argv) {  
        new C();  
    }  
}
```

Quelle est la sortie du programme ?



CONSTRUCTEURS

```
public class A {
    public A() {
        System.out.println("constructeur de A");
    }
}
class B extends A {
    public B(){
        System.out.println("constructeur de B");
    }
    public B(int r) {
        this();
        System.out.println("autre constructeur de B");
    }
}
```

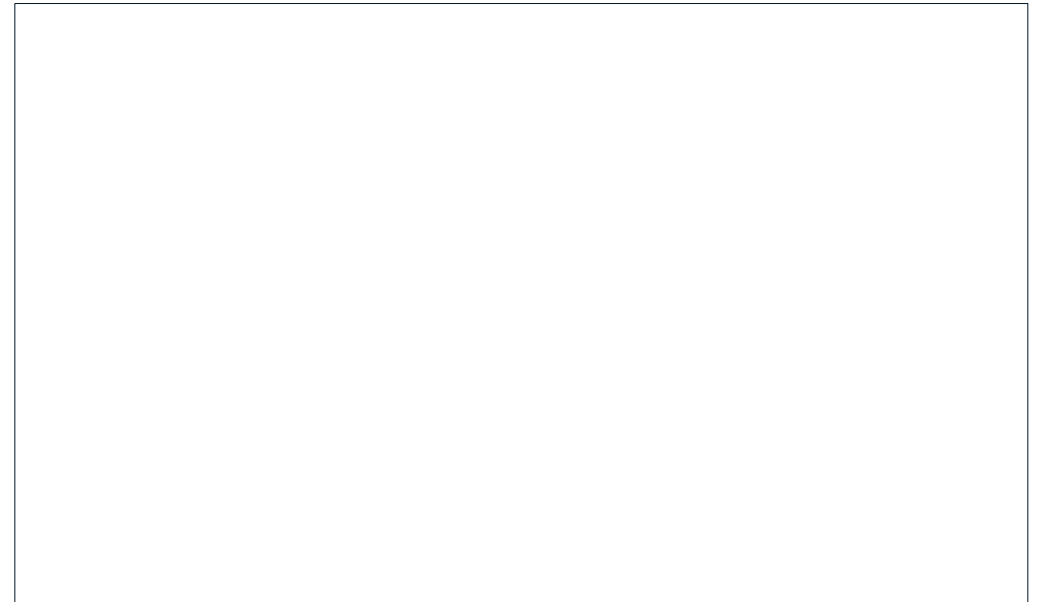
```
class C extends B {
    public C() {
        super(3);
        System.out.println("constructeur de C");
    }
    public static void main(String[] argv) {
        new C();
    }
}
```

Quelle est la sortie du programme ?

BLOC D'INITIALISATION

```
public class A
{
    public A()
    {
        System.out.println("constructeur de A");
    }
    { System.out.println(" Bloc de A"); }
}
public class B extends A {
    public B()
    {
        System.out.println("constructeur de B");
    }
    { System.out.println(" Bloc de B"); }
    public static void main(String[] argv)
    {
        new B();
    }
}
```

Quelle est la sortie du programme ?



REDÉFINITION DES MÉTHODES

- La redéfinition consiste à modifier le comportement d'une méthode de la superclasse, en fournissant une nouvelle implémentation dans la sous-classe.

```
class Chien extends Animal
{
    @override
    public void parler()
    {
        System.out.println(" Un chien aboie!");
    }
}
```

```
public class Animal
{
    public void parler()
    {
        System.out.println("Un animal parle !");
    }
}
```

- L'annotation **@Override** indique qu'une méthode d'une classe redéfinit une méthode de sa superclasse
=> **recommandé**.

REDÉFINITION DES MÉTHODES

- La méthode redéfinie dans une sous-classe doit avoir le **même nom**, le **même type de retour** et les **mêmes paramètres (même nombre, types et ordre)** que la méthode de la super-classe.
 - L'annotation **@Override** permet de vérifier que la redéfinition respecte bien ces règles.
- La redéfinition n'est pas la surcharge. Cette dernière consiste à proposer une variante de la méthode héritée avec des paramètres différents.

REDÉFINITION DES MÉTHODES

- Une méthode redéfinie dans la sous-classe masque la méthode héritée de la super-classe, lorsqu'elle est appelée par un objet de la sous-classe.

```
public class Main
{
    public static void main (String[] args)
    {
        chien Fox = new Chien();
        Fox.parler(); // Le chien aboie
    }
}
```

REDÉFINITION DE MÉTHODES

- En Java, on utilise le mot-clé **super** pour appeler une méthode de la super-classe, lorsqu'elle est redéfinie dans la sous-classe.

```
class Chien extends Animal
{
    public void parler()
    {
        super.parler()
        System.out.println(" Un chien aboie ! ");
    }
}
```

```
public class Main
{
    public static void main (String[] argv)
    {
        Chien Fox = new Chien();
        Fox.parler();
    }
}
Sortie :
// Un animal parle !
// Un chien aboie !
```

REDÉFINITION DE MÉTHODES

- Une classe peut interdire qu'une sous-classe redéfinisse une certaine méthode en utilisant le modificateur final.

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

Cette méthode ne peut pas être redéfinie par les sous-classes de A

```
class B extends A
{
    ❌ void uneMethode()
    {
        ...
    }
}
```

UP-CASTING

```
class Chien extends Animal
{
    @override
    public void parler()
    {
        System.out.println(" Le chien aboie!");
    }
}
```

```
class Chat extends Animal
{
    @override
    public void parler()
    {
        System.out.println(" Le chat miaule!");
    }
}
```

- **L'up-casting** consiste à affecter une instance d'un type enfant à une référence de type parent.

```
public static void main(String[] argv)
{
    Animal A;
    A = new Chien(); //up-casting
    A = new Chat();//up-casting
}
```

- **L'up-casting est une conversion implicite**, car un objet de type enfant est de type parent.

POLYMORPHISME

- En cas d'**up-casting**, le **polymorphisme s'applique automatiquement**, lorsque la référence appelle une **méthode redéfinie (@override)** dans la sous-classe.

```
public static void main(String[] args)
{
    Animal A ;
    A = new Chien(); //up-casting
    A.parler();
    A = new Chat(); //up-casting
    A.parler();
}
```

Même si **A** est de type parent, **la JVM exécute la version redéfinie correspondante au type enfant référencé par A => liaison dynamique**

- **Polymorphisme** => *Des objets référencés par une même variable peuvent avoir des comportements différents.*

POLYMORPHISME

```
public class A
{
    void faire()
    {
        System.out.println("niveau a");
    }
}

class B extends A
{
    @override
    void faire()
    {
        System.out.println("niveau b");
    }
}
```

```
class C extends B
{
    public static void main(String[] argv)
    {
        A a = new A();
        a.faire();
        a = new B();
        a.faire();
        a = new C();
        a.faire();
    }
}
```

Qu'affiche le programme ci-dessus ?

POLYMORPHISME

```
public static void main(String[] argv)
{
    Animal[] tab = { new Chien(), new Chat() };
    for (Animal A : tab)
    {
        System.out.println(A.getClass()); // type dynamique
        A.parler();
    }
}
```

Utilisez la méthode **getClass()** pour obtenir le type réel de l'objet à l'exécution.

Qu'affiche le programme ci-dessus ?

DOWN-CASTING

- Le down-casting consiste à affecter *un type référence parent* à *un type référence enfant*.
- Le down-casting nécessite une conversion explicite.
- Le down-casting peut provoquer l'exception **ClassCastException** si l'objet référencé n'est pas du bon type.

```
public static void main(String[] args)
{
    Animal unA = new Animal();
    try{
        Chien unC = (Chien) unA ; // down-casting qui génère l'exception ClassCastException
    }
    catch (ClassCastException exc)
    {
        System.out.println(exc +", un Animal n'est pas forcément un Chien");
    }
}
```

**java.lang.ClassCastException: Animal, un
Animal n'est pas forcément un Chien**

DOWN-CASTING

- Dans l'exemple ci-dessous A **référence bien** un chien. C'est pour cela que le down-casting a été accepté.

```
public static void main(String[] args)
{
    Animal A = new Chien(); //up-casting
    Chien  autreC = (Chien) A ; // down-casting accepté
}
```

DOWN-CASTING

- Avant un downcasting, il est fortement recommandé d'utiliser **instanceof** pour vérifier le type d'objet afin d'éviter l'exception **ClassCastException**.

```
public static void main(String[] args)
{
    Animal C = new Animal();
    if (C instanceof Chien) { // cas où C est réellement un chien
        Chien E = (Chien) C;
        System.out.println("Conversion réussie");
    }
    else {
        System.out.println("Impossible de convertir en Chien");
    }
}
```

Impossible de convertir en Chien

DOWN-CASTING

```
class Chien extends Animal
{
    @override
    public void parler() {...}
    public void garderMaison() { ... }
}
```

```
class Chat extends Animal
{
    @override
    public void parler() {...}
    public void jouerAvecBalle() {...}
}
```

```
Animal A = new Chien();
A.parler(); ✓
A.garderMaison(); ✗
A = new Chat();
A.jouerAvecBalle(); ✗
```

DOWN-CASTING

- En vérifiant le type de l'objet avec **instanceof**, vous pouvez appeler des méthodes spécifiques au sous-type

```
public static void lister(Animal[] tab )
{
    for (Animal A : tab)
    {
        A.parler(); // liaison dynamique
        if (A instanceof Chien) {
            ((Chien)A).garderMaison(); //down-casting
        }
        if (A instanceof Chat) {
            ((Chat)A).jouerAvecBalle(); //down-casting
        }
    }
}
```



CLASSE OBJECT

- La classe **Object** est la super-classe de toute classe en Java.
 - Si une classe ne précise pas de superclasse, elle hérite automatiquement de la classe **Object**.
 - Toutes les classes Java font partie d'une hiérarchie descendant de la classe **java.lang.Object**
 - Toutes les classes héritent des méthodes définies dans **Object** et peuvent donc les utiliser directement ou les redéfinir.

CLASSE OBJECT

- Parmi les méthodes héritées **de la classe object**, on trouve :

- **toString** : crée une valeur String pour l'objet.

```
public String toString();
```

- **equals** : compare une paire d'objets.

```
public boolean equals (Object obj);
```

- **hashCode** : génère une valeur de hachage de type int pour l'objet.

```
public int hashCode();
```

- **clone** : produit une réplique de l'objet.

```
protected Object clone();
```

LA MÉTHODE toString

- Il est recommandé de redéfinir la méthode `toString()`. Cette méthode permet de *retourner sous forme d'une chaîne de caractères les valeurs des attributs* de l'objet.

```
class Animal
{
    private String nom;
    public Animal(String nom) {
        this.nom = nom;
    }
    @override
    public String toString()
    {
        return " Nom : " + Nom ;
    }
}
```

```
class Essai
{
    public static void main(String[] argv)
    {
        Animal A = new Animal("Fox");
        System.out.println(A); //appel implicite de toString()
    }
}
```

Nom : Fox

LA MÉTHODE TOSTRING

- Redéfinissons la méthode **toString** dans la classe dérivée **Chien**

```
class Chien extends Animal
{
    private String couleur;
    public Chien (String nom, String couleur)
    {
        super(nom);
        this.couleur = couleur;
    }

    @override
    public String toString()
    {
        return super.toString() + " \n Couleur :" + Couleur ;
    }
}
```

```
class Essai
{
    public static void main(String[] argv)
    {
        Animal A = new Chien("Fox", "Brun");
        System.out.println(A);
    }
}
```

Nom : Fox
Couleur: Brun

Appel de toString() de Chien

LA MÉTHODE EQUALS

- Pour comparer deux objets, on redéfinit la méthode **equals** dans la classe.

```
@Override
public boolean equals(Object obj)
{   if (this==obj) return true;
    if ( obj == null || this.getClass() != obj.getClass())
    {
        return false;
    }
    Animal A = (Animal)obj;
    return Objects.equals(nom, A.nom);
}
```