Chapitre 9

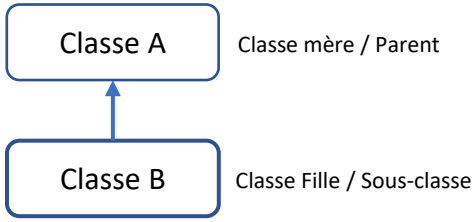
L’héritage

1. Intérêt et définitions
2. Droits d’accès aux propriétés
3. Construction d’objets hérités
4. Redéfinition et surcharge
5. Hiérarchie des classes et Héritage successif
6. Héritage de classes abstraites et interfaces

1 Intérêt et définitions (1/3)

- L'héritage est un puissant mécanisme Java qui répond à la problématique suivante :
  - Comment peut-on faire pour qu'une classe **B** réutilise et/ou étende le comportement d'une classe **A** existante ?
  - En d'autres termes : comment fait-on pour que B **hérite** des fonctionnalités que fournie déjà une classe A avec la possibilité de pouvoir les enrichir/modifier ?

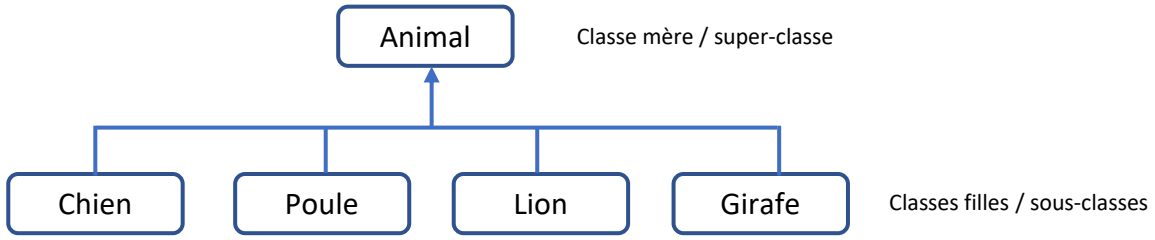
- Techniquement, la notion d'héritage est une relation dans laquelle on définit une nouvelle classe B en se basant sur une classe A existante
  - La classe A est appelée **classe mère** ou **parent** ou **super-classe** de B
  - La classe B est appelée **sous-classe** de A ou **classe fille** de A



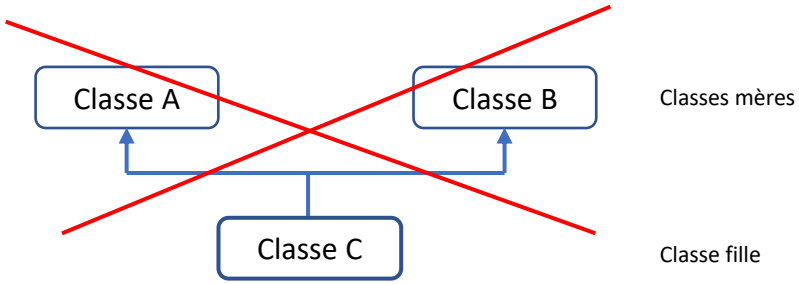
- L'héritage apporte trois dimensions importantes à une classe fille :
  - L'acquisition directe des attributs et méthodes définies dans sa classe mère, moyennant quelques conditions
  - La possibilité de redéfinir ou surcharger le fonctionnement des méthodes héritées
  - La possibilité de définir ses propres attributs et méthodes, comme toute classe ordinaire

1 Intérêt et définitions (2/3)

- Java permet qu'une classe puisse avoir plusieurs sous-classes/classes filles :



- Mais Java n'autorise pas l'héritage multiple. Une classe fille ne peut pas avoir plusieurs classes parents directs



1 Intérêt et définitions (3/3)

- Pour définir une relation d'héritage entre deux classes, on utilise le mot clé extends

```
public class A {

  //attributs
  //constructeurs
  //méthodes
}
```

```
public class B extends A {

  //attributs
  //constructeurs
  //méthodes
}
```

La classe B est déclarée en spécifiant qu'elle est une sous-classe de la classe A

- On peut définir une classe en la rendant non héritable. Elle ne pourra donc pas avoir de classe fille :
  - ➢ Pour ce faire on utilise le mot clé final au moment de sa déclaration

```
public final class A {

  //attributs
  //constructeurs
  //méthodes
}
```

```
public class B extends A {

  //attributs
  //constructeurs
  //méthodes
}
```

La classe B ne compilera pas, car A a été marquée comme final et donc non héritable

1 Intérêt et définitions

2 Droits d’accès aux propriétés

3 Construction d’objets hérités

4 Redéfinition et surcharge

5 Hiérarchie des classes et Héritage successif

6 Héritage de classes abstraites et interfaces

2 Droits d'accès aux propriétés (1/2)

❑ Précédemment, nous avons dit qu'une classe fille acquière directement les attributs et méthodes définies dans sa classe mère, **moyennant quelques conditions**

➔ Les conditions fixant l'accès à ces propriétés hérités sont les suivantes :

- Tout attribut ou méthode défini(e) dans une classe mère avec le modificateur private n'est pas directement accessible dans sa classe fille. On ne peut donc le manipuler directement
- Tout attribut ou méthode défini(e) dans une classe mère avec le modificateur default package (sans modificateur donc), n'est accessible dans la classe fille que si cette dernière est située dans le même package que sa classe mère
- Les attributs ou méthodes déclarés public ou protected sont accessibles directement dans toute classe fille
- L'accès à un attribut/méthode d'une classe mère dans sa classe fille peut se faire
  - > en utilisant la notation pointée préfixée du mot clé **super**
  - > en utilisant la notation pointée ordinaire avec un objet de la classe fille

2 Droits d'accès aux propriétés (2/2)

❑ Exemple :

```java
package p.classemere;

import java.util.List;

public class ClasseMere {

   public int abc;
   protected String str;

   private double xyz;
   List<Short> ls;


   public void methode1() {…}


   private float methode2() {…}

}
```

```java
package p.classefille;

import p.classemere.*;

public class ClasseFille extends ClasseMere {

   private void test() {
      super.methode1(); ✅

      super.methode2(); ❌
   }

   public static void main(String[] args) {
      ClasseFille cf = new ClasseFille();
      System.out.println(cf.str); ✅
      System.out.println(cf.abc); ✅

      System.out.println(cf.ls); ❌
      System.out.println(cf.xyz); ❌

      cf.methode2(); ❌
    }
}
```

Soit la classe **ClasseMere**, déclarée dans le package *p.classemere*, ayant des attributs et des méthodes définis avec divers modificateurs comme ci-dessus

Soit la classe **ClasseFille**, déclarée dans un autre package *p.classefille*, héritant de la classe **ClasseMere**, et dans laquelle nous avons deux méthodes *test()* et *main()* dont les instructions essaient d'accéder aux attributs et méthodes définies dans **ClasseMere**

En appliquant les règles d'accès étudiées, nous marquons avec :

❌ les instructions qui tomberont en erreur, car l'attribut/méthode n'est pas accessible

✅ les instructions corrects qui compilent, car l'attribut/méthode est accessible

3 Construction d'objets hérités (1/2)

❑ Java oblige une sous-classe à prendre en charge l'intégralité de la construction de sa classe mère

➔ C'est-à-dire que si une classe B hérite d'une classe A et que cette dernière dispose :

- d'attributs,
- d'un ou plusieurs constructeurs initialisant ces attributs,

➢ alors, tout constructeur de B doit invoquer un constructeur de A en lui passant les valeurs effectives attendues de façon à ce que lorsque B se construit, que son parent A se construise aussi en amont. Une classe fille ne peut exister si son parent n'existe pas

- Pour ce faire, Java fourni une méthode spéciale appelée super(…) qui doit être la **première instruction** du constructeur de B et qui prend en paramètres et dans l'ordre ceux déclarés dans le constructeur correspondant de la classe mère A
- Si le constructeur de B n'invoque pas explicitement la méthode *super(..)* dans son/ses constructeurs, alors Java l'insérera implicitement et appellera le constructeur par défaut de sa classe mère A

3 Construction d'objets hérités (2/2)

❑ Exemple :

```java
package p.classemere;

public class Animal {

  private int nbrePattes;
  private String espece;

  public Animal(int nbrePattes, String espece) {
    this.nbrePattes = nbrePattes;
    this.espece = espece;
  }

  public int getNbrePattes() {
    return nbrePattes;
  }

  public String getEspece() {
    return espece;
  }

}
```

```java
package p.classefille;

import p.classemere.Animal;

public class Chien extends Animal {

    String nom;

    public Chien(String nom, int nbrePattes, String espece) {
       super(nbrePattes, espece);
       this.nom = nom;
    }

    public static void main(String[] args) {
       Chien ch = new Chien("Jack", 4, "Berger Allemand");

       System.out.println("Cet Animal s'appelle " +
                          ch.nom + " et c'est un " +
                          ch.getClass().getSimpleName() +
                          " de type " + ch.getEspece() +
                          " à " + ch.getNbrePattes() + " pattes");

    }

}
```

Soit la classe **Animal**, déclarée dans le package *p.classemere*, ayant deux attributs private *nbrePattes* et *espece*, un constructeur qui initialise ces attributs et deux méthodes public *getter* pour récupérer leurs valeurs.

Soit la classe **Chien**, déclarée dans un autre package *p.classefille*, héritant de la classe **Animal**. Pour que la classe Chien au moment de son instanciation dans la méthode main(), valorise aussi les attributs *nbrePattes* et *espece* de sa classe mère, on ajoute dans son constructeur la méthode *super(nbrePattes, espece)* qui invoquera le constructeur `public Animal(int nbrePattes, String espece)` de la classe Animal.
L'exécution de ce programme affichera : **Cet Animal s'appelle Jack et c'est un Chien de type Berger Allemand à 4 pattes**

4 Redéfinition

- Java permet à une sous-classe de modifier le corps d'une méthode qu'elle a héritée de sa classe mère
  - > Ce processus s'appelle la **redéfinition**
  - > La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques) et fournir dans la sous-classe une nouvelle implémentation de la méthode. Cette nouvelle implémentation masque alors complètement celle de la super-classe

    (i.e, nous verrons la notion d'exception et sa propagation au chapitre 11, pour l'instant il faut retenir les deux premiers critères)

- Exemple :

```java
package p.classemere;

public class Animal {

  public int nbrePattes;
  public String espece;

  public Animal(int nbrePattes, String espece) {
    this.nbrePattes = nbrePattes;
    this.espece = espece;
  }

  public String retournerInfoAnimal() {
    return nbrePattes + " " + espece;
  }
}
```

```java
package p.classefille;

import p.classemere.Animal;

public class Chien extends Animal {

    String nom;

    public Chien(String nom, int nbrePattes, String espece) {
       super(nbrePattes, espece);
       this.nom = nom;
    }

    public String retournerInfoAnimal() {
       return nom + " " + espece + " " + nbrePattes ;
    }

}
```

La classe Chien hérite de Animal et redéfinit sa méthode retournerInfoAnimal(). Elle ne change ni le nom de cette dernière, ni le type retour, ni le modificateur, ni les paramètres. Elle modifie juste le corps de la méthode

Chien ch = new Chien("Jack", 4, "Berger Allemand");

Lorsqu'on invoquera donc cette méthode au travers de l'instance ch, ch.retournerInfoAnimal(), c'est celle redéfinie dans la classe Chien qui sera toujours appelée et non plus celle de Animal

4 Surcharge

- Java permet à une sous-classe de définir d'autres méthodes qui **portent le même nom** qu'une méthode héritée de sa classe mère
  - > Ce processus s'appelle la **surcharge**
  - > La surcharge d'une méthode héritée implique l'utilisation du même nom, mais n'oblige pas de conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques). On peut donc dans la sous-classe avoir plusieurs méthodes de même nom et avoir accès à la fois à la/les méthode(s) héritée(s), mais aussi à la/les méthode(s) surchargée(s)

- Exemple :

```java
package p.classemere;

public class Animal {

  private int nbrePattes;
  private String espece;

  public Animal() {
  }

  public String retournerInfoAnimal(int nbrePattes, String espece) {
    return nbrePattes + " " + espece;
  }
}
```

```java
package p.classefille;
import p.classemere.Animal;

public class Chien extends Animal {

   String nom;

   public Chien() {
      super();
   }

   public String retournerInfoAnimal(String nom, int nbrePattes, String espece) {
      return nom + " " + espece + " " + nbrePattes;
   }

}
```

La classe Chien hérite de Animal et surcharge sa méthode retournerInfoAnimal()

Chien ch = new Chien();

ch.retournerInfoAnimal(4, "Berger Allemand") appelle la méthode définie dans Animal

Alors que

ch.retournerInfoAnimal("Jack", 4, "Berger Allemand") appelle la méthode de la classe Chien

1 Intérêt et définitions

2 Droits d’accès aux propriétés

3 Construction d’objets hérités

4 Redéfinition et surcharge

5 Hiérarchie des classes et Héritage successif

6 Héritage de classes abstraites et interfaces

5 Hiérarchie des classes et Héritage successif (1/3)

❑ Toute classe Java hérite **par défaut** d'une classe de base prédéfinie dans le package *java.lang*.
Cette classe de base s'appelle **Object**

➔Ainsi, lorsqu'on déclare une classe sans utiliser le mot clé extends, Java par défaut le lui ajoute implicitement un héritage vers la classe Object. En d'autres termes, toute classe est une sous-classe de la classe Object

- Exemple :

```
public class A {

  //attributs
  //constructeurs
  //méthodes
}
```



```
public class A extends Object {

  //attributs
  //constructeurs
  //méthodes
}
```

- La classe Object est une classe qui fournie les fonctionnalités de base dont hérite toute classe et qui peuvent lui être utile

5 Hiérarchie des classes et Héritage successif (2/3)

- Contenu de la classe java.lang.Object fourni par Java :

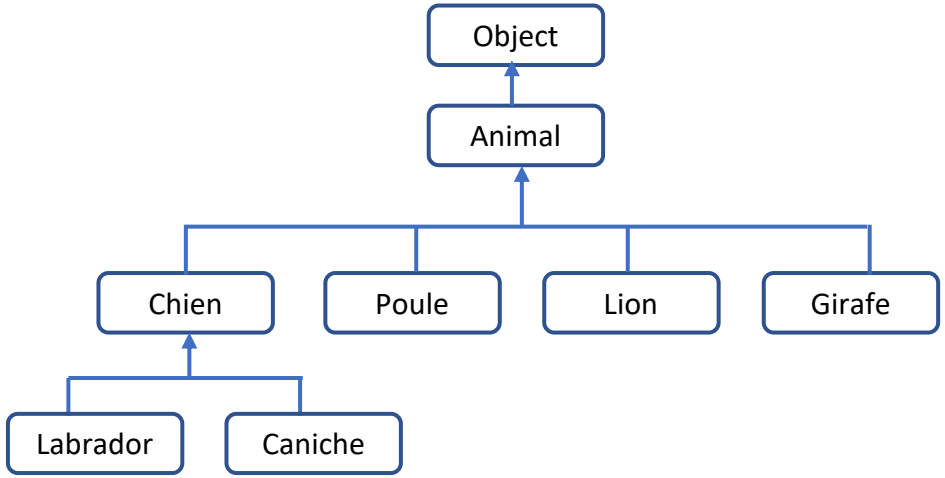
- Documentation sur les différents membres ici

```
package java.lang;

public class Object {

    private static native void registerNatives();
    static {
        registerNatives();
    }


    public final native Class<?> getClass();

    public native int hashCode();

    public boolean equals(Object obj) {
        return (this == obj);
    }

    protected native Object clone() throws CloneNotSupportedException;

    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }

    public final native void notify();

    public final native void notifyAll();

    public final native void wait(long timeout) throws InterruptedException;

    public final void wait(long timeout, int nanos) throws InterruptedException {
        if (timeout < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }

        if (nanos < 0 || nanos > 999999) {
            throw new IllegalArgumentException(
                                "nanosecond timeout value out of range");
        }

        if (nanos > 0) {
            timeout++;
        }
        wait(timeout);
    }

    public final void wait() throws InterruptedException {
        wait(0);
    }

    protected void finalize() throws Throwable { }
}
```

5 Hiérarchie des classes et Héritage successif (3/3)

- Java permet l'héritage successif entres classes. Dans cette hiérarchie, le seul et unique ancêtre de toutes est **Object**

- Par transition, une sous-classe héritera de toutes les fonctionnalités disponibles chez son parent, puis grands-parents, arrière grands-parents, etc :



Sur le schéma ci-contre, on observe une succession d'héritage dont l'étude d'une branche hiérarchiquement et chronologiquement est définie comme suit :

1. public class Animal extends Object {…}
2. public class Chien extends Animal {…}
3. public class Labrador extends Chien {…}
4. public class Caniche extends Chien {…}

Ainsi, la classe Labrador a pour parent Chien, elle-même ayant pour parent Animal, et enfin elle-même ayant pour parent Object (idem pour Caniche).

La classe Labrador (resp. Caniche) héritera de tous les attributs/méthodes accessibles dans Chien, Animal et Object

1 Intérêt et définitions

2 Droits d’accès aux propriétés

3 Construction d’objets hérités

4 Redéfinition et surcharge

5 Hiérarchie des classes et Héritage successif

6 Héritage de classes abstraites et interfaces

6 Héritage de classes abstraites

- Au chapitre 4, nous avions défini la notion de classe abstraite
  - ➔ Pour rappel, une **classe abstraite** est une classe particulière Java, portant le mot clé *abstract,* non instanciable, qui a deux fonctions :
    - définir des méthodes abstraites qui seront implémentées plus tard dans ses sous-classes. Exemple de la méthode *isDomestique()*
    - définir des méthodes déjà codés et qui peuvent directement être réutilisées. Exemple de la méthode *getNom()*

```
public abstract class Animal {

    private String nom;

    public abstract boolean isDomestique();

    public String getNom(){ return this.nom; }

}
```

- Toute classe héritant d'une classe abstraite, et si elle-même n'est pas une classe abstraite, a **l'obligation d'implémenter (donner un corps) à toutes les méthodes abstraites héritées**, mais aussi éventuellement de réaliser toutes les autres opérations dont permet l'héritage (redéfinition, surcharge, construction, etc) :

```
public class Chien extends Animal {

    private String type;

    public boolean isDomestique() { return true };

    public String getInfo(){ return (this.getNom() + type ); }

}
```

6 Héritage d'interface

- Au chapitre 4, nous avions défini la notion d'interface

  ➔ Pour rappel, une **interface** est une classe particulière Java, disposant de méthodes abstraites, statiques et par défaut (default). Son intérêt est de fournir un ensemble de fonctionnalités réutilisables par plusieurs classes qui l'héritent
  - Java permet à une classe d'hériter d'une interface au travers du mot clé implements
  - Java permet à une **classe d'hériter de plusieurs interfaces** différentes (ce qui est interdit en ce qui concerne l'héritage de classe)

```
public interface CompteBancaire {

    void deposerArgent(double montant) ;
    void retirerArgent(double montant);
    double getBalance();

    default void transferer(CompteBancaire destination) {
        double montantATransferer = this.getBalance();
        destination.deposerArgent(montantATransferer);
    }

    static double getBalanceTotale(CompteBancaire... comptes) { //… }
}
```

```
public interface PlatfondTransaction {

    double PLAFOND_DEPOT = 3000.0;

    double PLAFOND_RETRAIT = 2000.0;

}
```

- Toute classe héritant d'une interface a **l'obligation d'implémenter (donner un corps) à toutes les méthodes abstraites héritées**, mais aussi éventuellement de réaliser toutes les autres opérations dont permet l'héritage (redéfinition, surcharge, construction, etc) :

```
public class CompteCourant implements CompteBancaire, PlatfondTransaction {

    private double solde;

    public void deposerArgent(double montant) { if(montant < PLAFOND_DEPOT ) { this.solde += montant; } };
    public void retirerArgent(double montant) { if(montant < PLAFOND_RETRAIT ) { this.solde -= montant; } };
    public double getBalance(){ return this.solde; }

}
```


SUCCESS