Les interfaces

Les interfaces

- Une interface permet de créer un sous-type sans créer de classe de base. **Une interface n'est pas une classe.**
- Une interface est un peu comme une classe abstraite, mais en version minimaliste: elle peut contenir des constantes et des méthodes abstraites, mais ne peut pas avoir de méthodes d'instance concrètes, de constructeurs ou d'attributs.
- Une interface permet donc de définir un ensemble de méthodes à implémenter. Elle établit ainsi un contrat que certaines classes peuvent s'engager à respecter.
- La syntaxe de déclaration d'une interface est:

```
visibilité interface NomInterface {
    // définition des constantes publiques
    // Il ne peut pas y avoir de champs d'instance.
    // Il ne peut pas y avoir de constructeurs.
    // définition des méthodes abstraites
}
```

- Exemple:

```
public interface Amical {
    public final int NB_CALIN_MIN = 3;
    public abstract void faireCalin();
    public abstract int nombreCalin();
}
```

Les interfaces

- Une interface contient :
  - Éventuellement des constantes publiques et statiques.
    - Les mots-clés `public`, `static et final` sont **implicites et facultatifs**, ils peuvent être omis puisque les attributs d'instance ne sont pas autorisés.
  - Éventuellement des méthodes publiques et abstraites.
    - Les mots-clés `public` et `abstract` sont donc également **implicites et facultatifs**.
  - Depuis Java 8, des méthodes par défaut (`public default`), des méthodes statiques (`public static`) et depuis Java 9, des méthodes privées (`private`) et des méthodes privées statiques (`private static`). Dans ce cours, nous traiterons des interfaces qui contiennent des constantes et des méthodes abstraites.
  - Et rien d'autre

```
interface Amical {
    int NB_CALIN_MIN = 3;
    void faireCalin();
    int nombreCalins();
}
```

```
public interface Amical {
    public static final int NB_CALIN_MIN = 3;
    public abstract void faireCalin();
    public abstract int nombreCalin();
}
```

Implémentation d'une interface

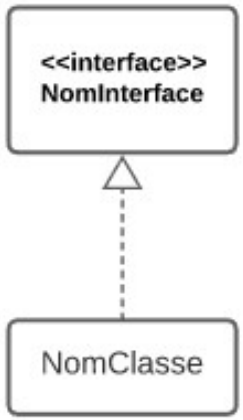
- On dit **qu'une classe implémente une interface**, si elle **définit les méthodes de l'interface**.
- En java, on déclare qu'une classe implémente une interface avec le mot clé **`implements`**.
- **Si la classe n'implémente pas toutes les méthodes abstraites de l'interface, elle devient alors une classe abstraite** (`abstract`).
- Une classe qui implémente une interface peut utiliser ses constantes directement sans avoir à spécifier le nom de l'interface.

- Syntaxe

```
public class NomClasse implements NomInterface {
    // On peut utiliser une constante de l'interface NomInterface
    // directement sans avoir à spécifier le nom de l'interface.

    // Implémentation de toutes les méthodes abstraites
    // de l'interface NomInterface.

@Override
    public typeRetour nomMethode( typePar1 nomPar1, typePar2 nomPar2, … ) {
        . . .
    }
}
```

Représentation graphique d'une interface

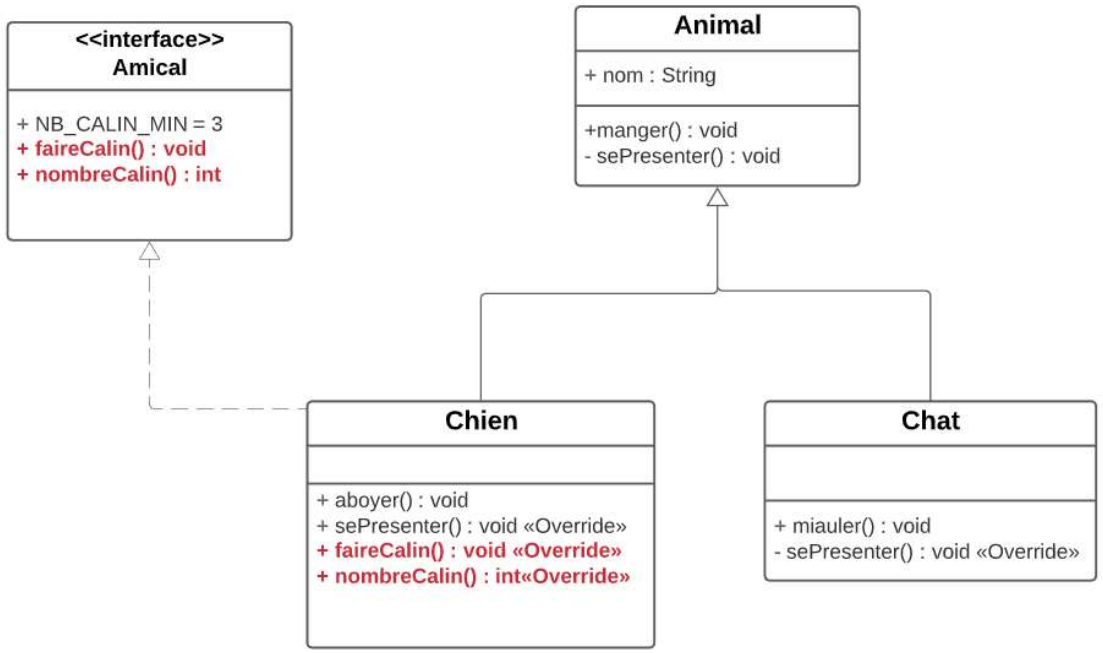
- Une flèche pointillée avec un bout vide pointe de la classe vers l'interface.



Implémentation d'une interface

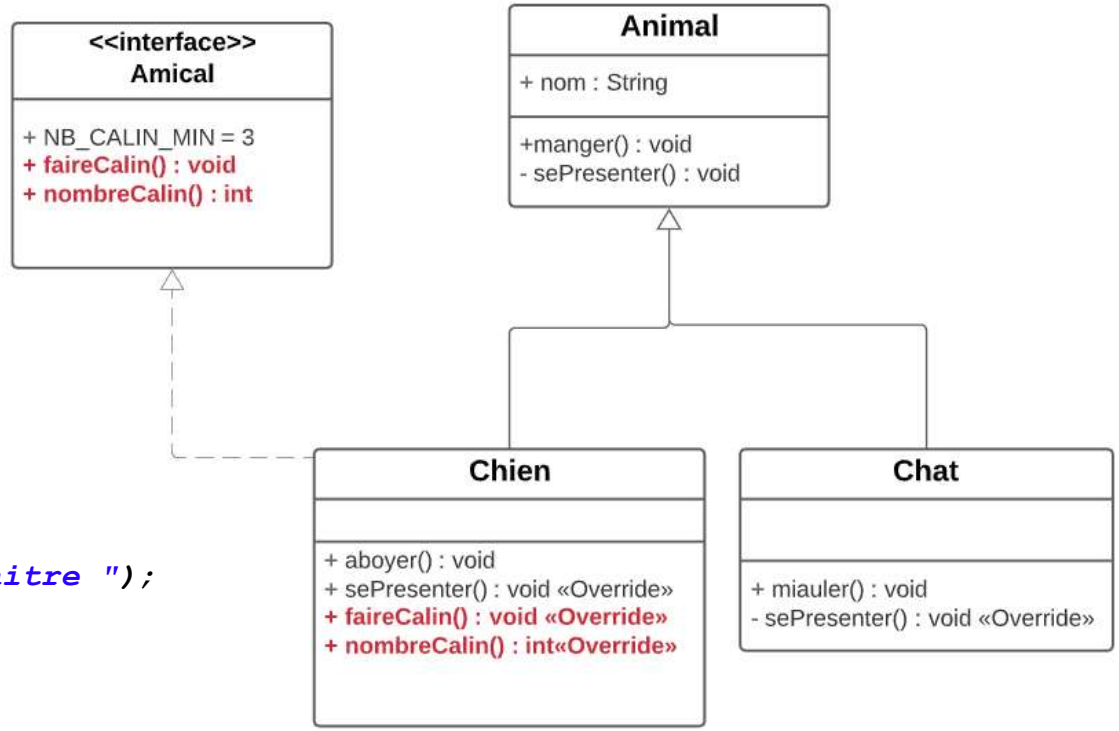
- Une classe peut hériter d'une autre classe et implémenter une interface. Dans ce cas, on doit **respecter l'ordre des déclarations**: l'expression d'héritage (`extends`) doit être avant l'expression d'implémentation (`implements`), sinon le code ne compilera pas.
- Exemple: La classe `Chien` hérite de la classe `Animal` et implémente l'interface `Amical`.
- La classe Chien doit implémenter toutes les méthodes de l'interface `Animal`. Sinon, la classe `Chien` deviendra abstraite.

```
public class Chien extends Animal implements Amical {
  //contenu de la classe Chien

   …
  // Méthode de l'interface Amical
   @Override
   public void faireCalin() {
      …
   }
   @Override
   public int nombreCalin() {
      …
   }
      …
  // Méthode de la super-classe Animal
   @Override
   public void sePresenter() {
      …
   }
}
```



Implémentation d'une interface

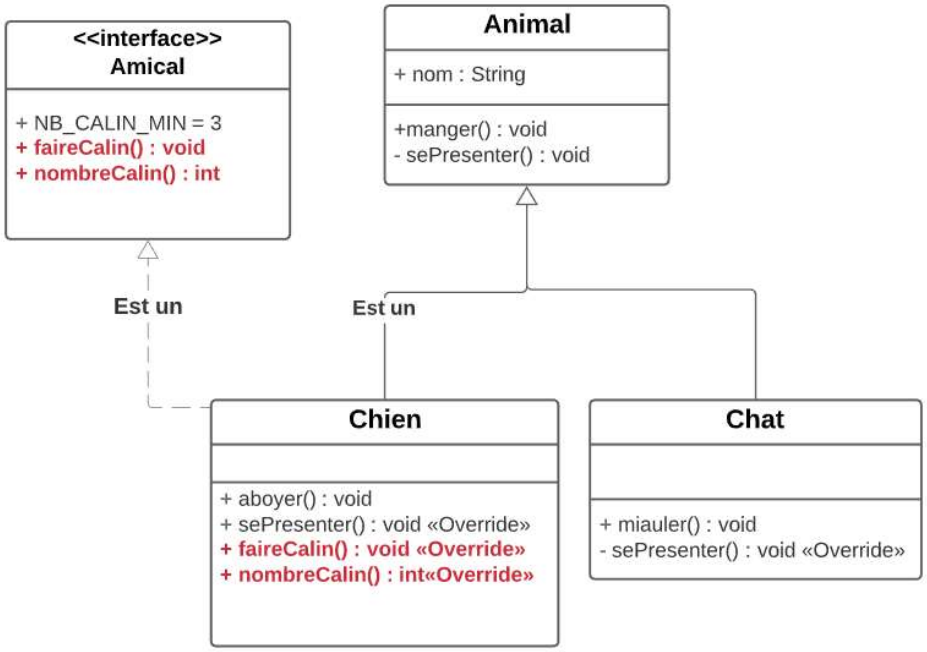
```
public class Chien extends Animal implements Amical {
    public Chien(String nom) {
        super(nom);
    }
    public void aboyer() {
        System.out.println("Wof");
    }
    @Override
    // Méthode de la superclasse Animal
    public void sePresenter() {
        System.out.println("Je suis le chien "
            + getNom());
    }
     //Méthode de l'interface Amical
    @Override
    public void faireCalin() {
        System.out.println("Je me colle à mon maitre ");
    }
     //Méthode de l'interface Amical
    @Override
    public int nombreCalin() {
        return NB_CALIN_MIN * 2;
     // Accès à la constante de l'interface directement
    }

}
```



Implémentation d'une interface

Une interface permet d'attribuer un type supplémentaire à une classe .

- C'est la solution de Java **pour imposer un contenu commun à des classes en dehors d'une relation d'héritage.**
- **Java ne permet pas l'héritage multiple**, l'utilisation des interfaces permet de pallier cette carence, dans la mesure où une classe peut implémenter plusieurs interfaces.
- **Exemple:** Un objet `Chien` est de type `Animal`, il est également de type `Amical`.



Une classe peut implémenter plusieurs interfaces

- Une classe peut implémenter plusieurs interfaces. Dans ce cas, elle doit implémenter toutes les méthodes abstraites de toutes les interfaces sinon, elle devient une classe abstraite.

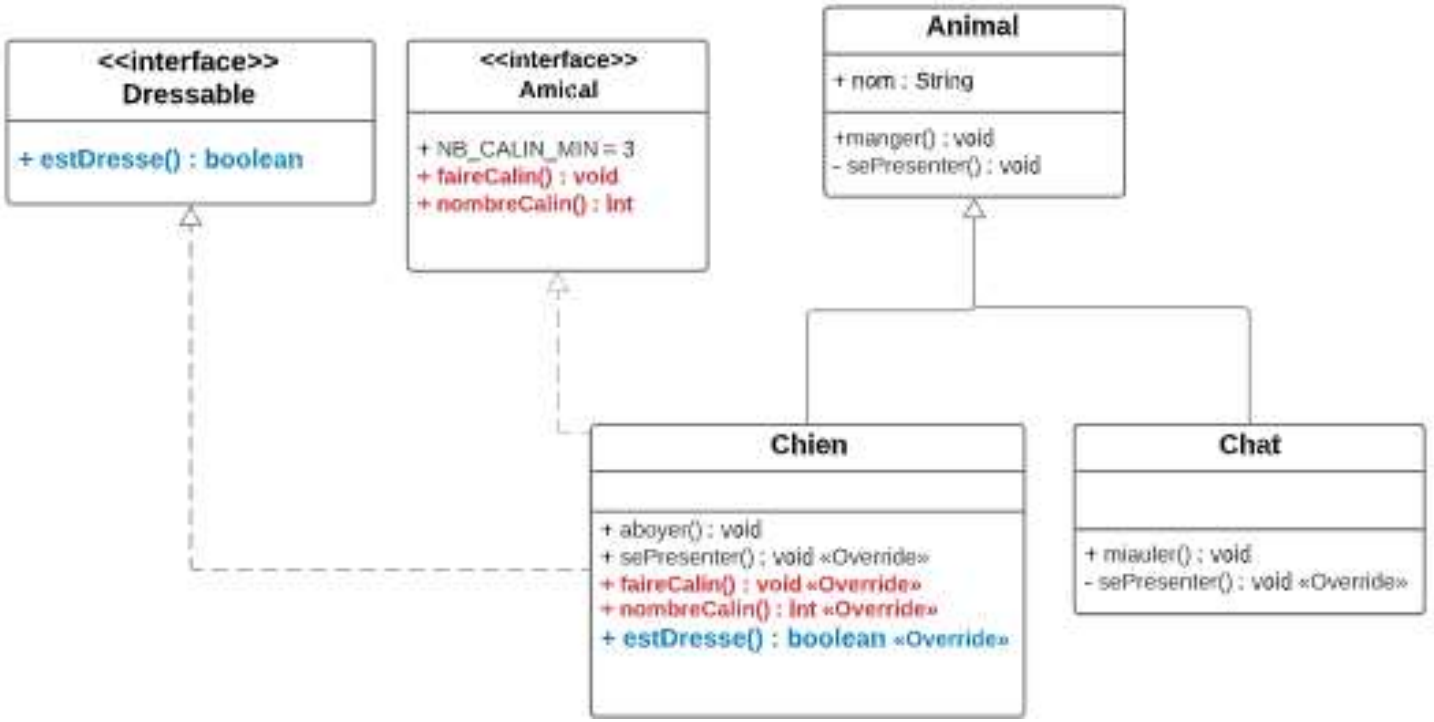
```
public class NomClasse implements NomInterface1, NomInterface2, … {
    // Implémentation de toutes les méthodes abstraites
    // des interfaces NomInterface1, NomInterface2, …

    @Override
     public typeRetour nomMethode( typePar1 nomPar1, typePar2 nomPar2, … ) {
        . . .
    }
}
```

- Une classe qui n'implémente pas toutes les méthodes des interfaces devient abstraite

```
public abstract class NomClasse implements NomInterface1, NomInterface2, … {
   // Les méthodes abstraites des interfaces
   // NomInterface1, NomInterface2, …
   // ne sont pas toutes implémentées.
   . . .
}
```

Une classe peut implémenter plusieurs interfaces

- **Exemple:**



Héritage entre les interfaces

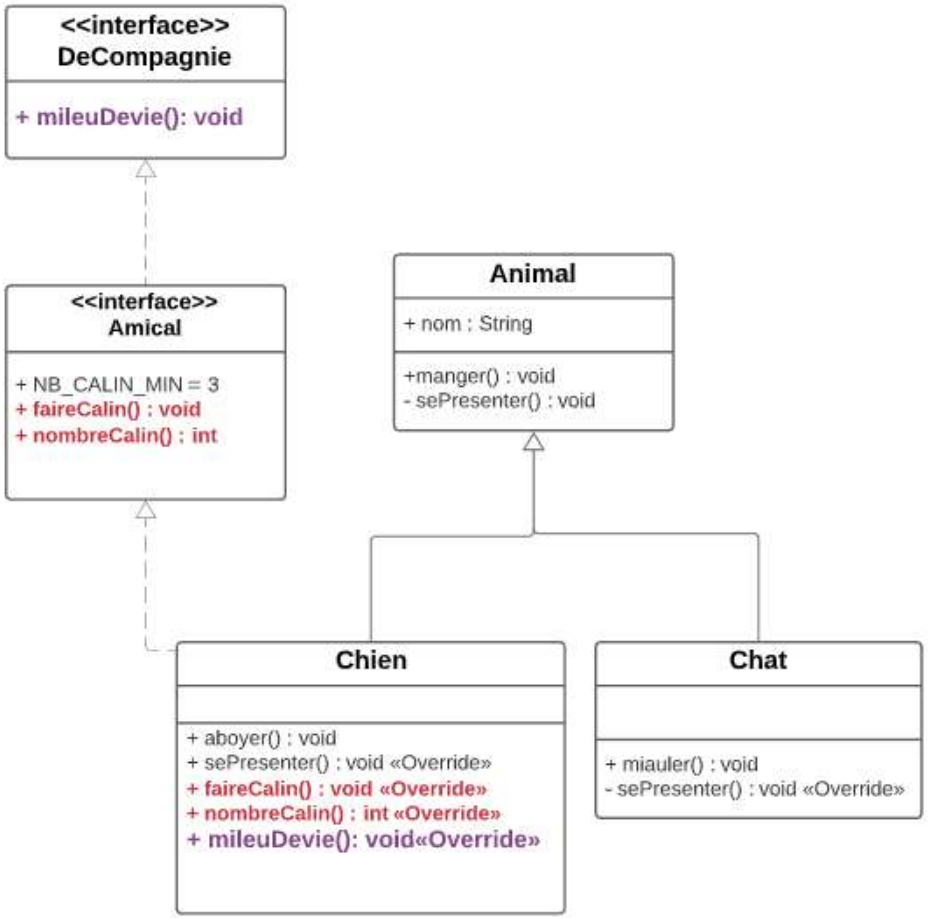
- Une **interface peut hériter d'une autre interface**.
- On a vu, dans les précédents cours, qu'une classe ne peut pas hériter de plusieurs classes. Contrairement à une classe, une interface peut hériter de plusieurs interfaces. Dans la déclaration de l'interface, le mot-clé `extends` sera suivi par la liste des noms d'interfaces séparés par des virgules.

```
public interface NomInterfaceFille extends NomInterfaceMere1,  NomInterfaceMere2, … {
    //constantes et méthodes de l'interface NomInterfaceFille
}
```

- Une classe qui implémente `NomInterfaceFille` doit implémenter toutes les méthodes abstraites des deux interfaces `NomInterfaceFille` et `NomInterfaceMere`. Sinon, elle devient une classe abstraite.
- Voir un exemple dans la diapositive suivante.

Héritage entre les interfaces

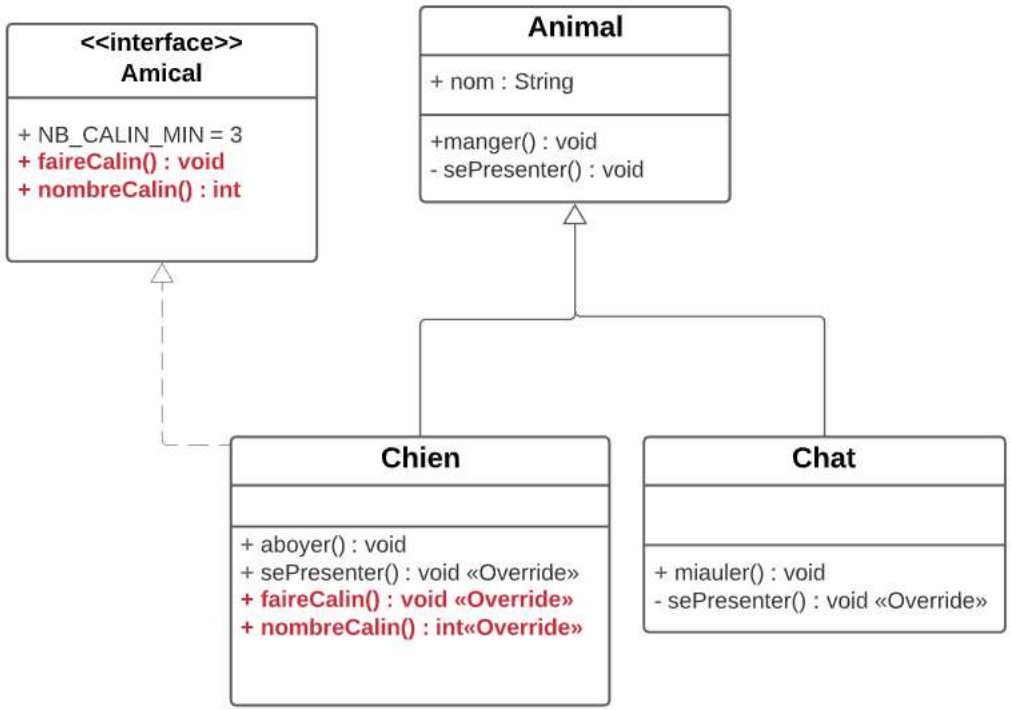
- La classe `Chien` doit implémenter toutes les méthodes des interfaces `Amical` et `DeCompagnie`. Sinon, elle sera abstraite.



Déclaration d'une variable objet d'une interface

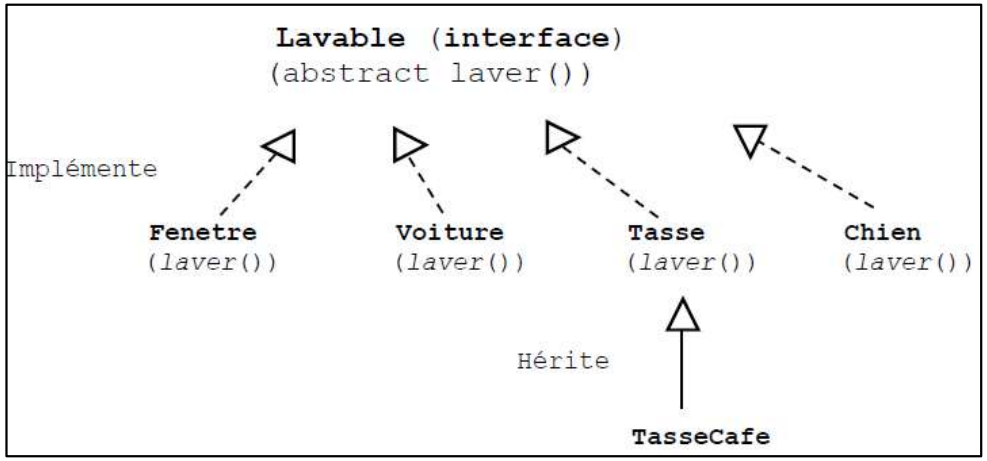
- **On ne peut pas instancier une interface avec l'opérateur `new`** puisqu'une interface n'est pas une classe.
- Par contre, on peut :
  - Déclarer une variable de type interface
  - Y affecter un objet d'une classe qui implémente l'interface
- Il est possible d'utiliser l'opérateur **`instanceof`** pour vérifier si un objet implémente une interface.
- Exemples:

```
Amical chien1 = new Amical();
// impossible puisque Amical n'a pas de
//  constructeur: c'est une interface.

Amical amical = new Chien("Pitou");
// Ok. Chien est une classe non abstraite
// qui implémente l'interface Amical

Chien chien = new Chien("Adolph");
if (chien instanceof Amical) { // vrai
   System.out.println
     ("L'objet chien implémente Amical");
}
```



Déclaration d'une variable objet d'une interface

- Exemples :




```
Lavable lavable
     = new Lavable(...); // Pas Ok.
     = new Fenetre(...); // Ok.
     = new Voiture(...); // Ok.
     = new Tasse(...); // Ok.
     = new Chien(...); // Ok.
     = new TasseCafe(...); // Ok.
lavable.éléments accessibles de Lavable
lavable.laver(); // Méthode laver() de la classe appropriée.
// Polymorphisme.
Tasse tasse = new Tasse(...);
tasse.éléments accessibles de Tasse et de Lavable
TasseCafe tasseCafe = new TasseCafe(...);
tasseCafe.éléments accessibles de TasseCafe, de Tasse et de Lavable
```

L'interface Comparable

- L'interface Comparable est utilisée pour classer les objets d'une classe définie par l'utilisateur. Cette interface contient une seule méthode nommée `compareTo (Object)`.
- Ainsi, pour comparer des objets d'une classe, on utilise la méthode `compareTo()`. La classe doit implémenter l'interface Comparable. https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

```
interface Comparable <T>
   int compareTo(T o);
}
```

- La méthode `compareTo(T o)`
  - Reçoit en paramètre un autre objet du même type que l'objet avec lequel on veut le comparer.
  - Elle retourne :
    - **0** si l'objet reçu en paramètre est **égal** à l'objet en cours (`this`).
    - un entier **négatif** (<0) si l'objet courant (`this`) est **inférieur** à l'objet spécifié en paramètre.
    - un entier **positif** (>0) si l'objet courant (`this`) est **supérieur** à l'objet spécifié en paramètre.

L'interface Comparable

- Pour vérifier si un objet1 est plus petit qu'un autre objet2, on écrit :

```
if ( obj1.compareTo(obj2) < 0 ) {
   // obj1 est plus petit que obj2.
}
```

Et non pas

```
if ( obj1 < obj2 ) {
// L'adresse obj1 est plus petite que l'adresse obj2.
}
```

- Pour vérifier si un objet1 est plus grand qu'un autre objet2, on écrit :

```
if ( obj1.compareTo(obj2) > 0 ) {
   // obj1 est plus grand que obj2.
}
```

Et non pas

```
if ( obj1 > obj2 ) {
// L'adresse obj1 est plus grand que l'adresse obj2.
}
```

- Pour vérifier si un objet1 est égal à un autre objet2, on écrit :

```
if ( obj1.compareTo(obj2) == 0 ) {
   // obj1 est égal (équivalent) à obj2.
}
```

Et non pas

```
if ( obj1 == obj2 ) {
// L'adresse obj1 est égale à l'adresse obj2.
}
```

L'interface Comparable:

Exemple: en redéfinissant la méthode `compareTo()` de la classe `Produit`, on peut établir une relation d'ordre entre les objets `Produit` basée sur l'attribut `numero`.

```
public class Produit implements Comparable<Produit> {
   private int numero;
   private String description;    private double prix;
   public Produit(int numero, String description, double prix) {
     this.numero = numero;
     this.prix = prix;
     this.description = description;
   }
   @Override
   public int compareTo(Produit autreObjet) {
     int compNumero;
     if (this.numero < autreObjet.numero) {
        compNumero = -1;
     } else if (this.numero > autreObjet.numero) {
         compNumero = 1;
     } else {
        compNumero = 0;
     }
     return compNumero;
     }
}
```

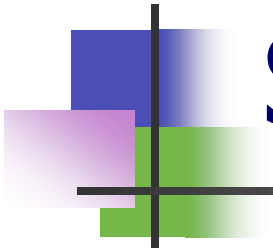
L'interface Comparable: exemple

- On peut comparer ainsi des objets de la classe Produit

```
public static void main(String[] args) {
     Produit produit1 = new Produit(10, "Chaise", 132.56);
     Produit produit2 = new Produit(20, "Table", 98.50);
     Produit produit3 = new Produit(3", "Bureau", 120.00);

    if (produit1.compareTo(produit2) < 0) {
        System.out.println("produit1 est classé avant produit2 selon la relation d'ordre ");
    } else if (produit1.compareTo(produit2) > 0) {
        System.out.println("produit1 est classé après produit2 selon la relation d'ordre ");
    } else {
        System.out.println("produit1 et produit2 ont le même classement  selon la relation
           d'ordre ");
    }
}
```

- Affichage à la console:

```
produit1 est classé avant produit2 selon la relation d'ordre
```



Sources

- RÉSUMÉ DOCUMENT 7 INTERFACES par Soti Kyprianou et Christian Mongeon