Module de mise à niveau en Java
Cours 3

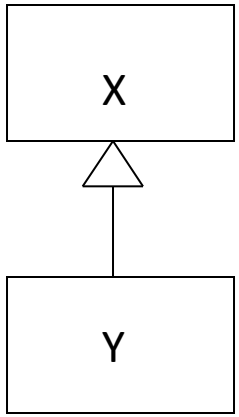
Laure KAHLEM
laure.kahlem@univ-orleans.fr

Références

- Kathy Sierra et Bert Bates. Java Tête la première. O'Reilly, 2007.
- Mickaël Kerboeuf. Fondements de la programmation orientée objet avec Java8. Ellipses, 2016
- David Flanagan. Java in a Nutshell. Manuel de référence. O'Reilly, 2002.
- Cyrille Herby. Apprenez à programmer en JAVA. Broché, 2012.
- José Paumard. « Java en ligne » [en ligne]. http://blog.paumard.org/cours/java/ (Consulté le 15/07/2015).

L'HÉRITAGE

- **Principes**
  - L'héritage permet de mettre en œuvre la notion de **réutilisation.**
  - **L'héritage est une relation entre deux classes qui permet à une classe de réutiliser les caractéristiques d'une autre classe.**



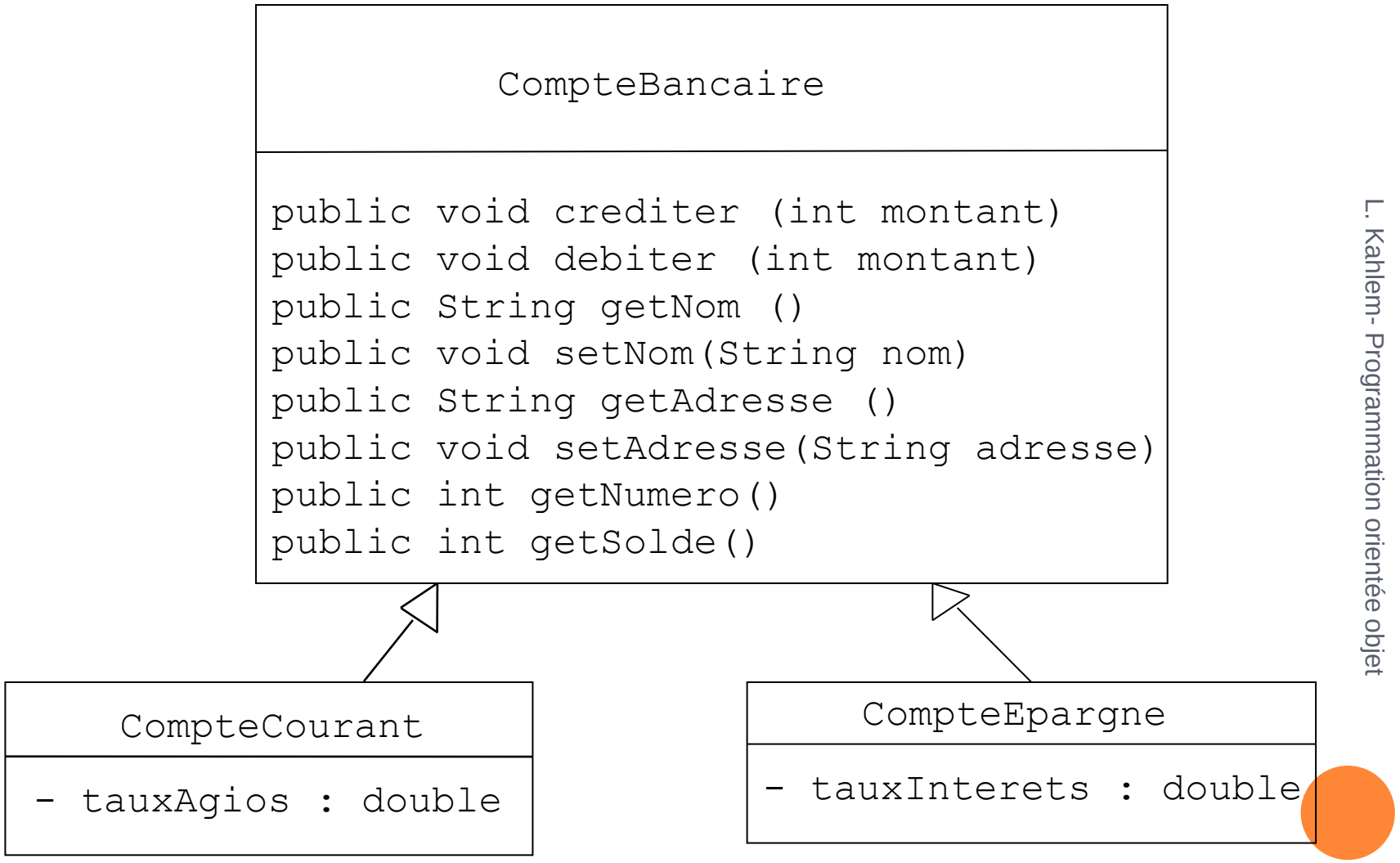
Classe parent ou superclasse

Classe enfant ou sous-classe

L'HÉRITAGE

- **Exemple**
  - Un compte bancaire peut soit être un **compte courant** soit **un compte d'épargne.**
    - **Un compte courant** permet un solde négatif mais des agios sont déduits chaque jour si le solde est négatif.
    - **Un compte d'épargne** doit toujours avoir un solde positif mais on ajoute des intérêts calculés chaque jour.

L'HÉRITAGE


CompteBancaire
public void crediter (int montant)
public void debiter (int montant)
public String getNom ()
public void setNom(String nom)
public String getAdresse ()
public void setAdresse(String adresse)
public int getNumero()
public int getSolde()
CompteCourant
- tauxAgios : double
CompteEpargne
- tauxInterets : double

L'HÉRITAGE

Classe `CompteCourant` :

```
public class CompteCourant extends CompteBancaire{
    private double tauxAgios; //taux quotidien des agios

    à compléter
}
```

Classe `CompteEpargne` :

```
public class CompteEpargne extends CompteBancaire{
    private double tauxInterets; //taux d'intérêts par jour

    à compléter
}
```

L'HÉRITAGE

- **Constructeur de la sous-classe**
  - La première instruction dans le constructeur de la sous-classe doit être l'appel au constructeur de la superclasse avec le mot clé **`super`**.
  - Si on ne fait pas d'appel explicite au constructeur de la superclasse, c'est le constructeur par défaut de la superclasse qui est appelé à condition qu'il existe.

L'HÉRITAGE

- **Constructeur de la sous-classe**

Exemple :

Classe CompteCourant :

```
//constructeur
public CompteCourant(String nom,String
    adresse, double tauxAgios) {
    super (nom, adresse);
    this.tauxAgios = tauxAgios;
}
```

L'HÉRITAGE

- **Constructeur de la sous-classe**

Exemple :

Classe CompteEpargne :

```
//constructeur
public CompteEpargne ( String nom,String
    adresse, double tauxInterets) {
    super ( nom, adresse );
    this.tauxInterets = tauxInterets;
}
```

L'héritage

- **La redéfinition de méthode**
  - Certaines méthodes de la superclasse ne conviennent pas pour la sous-classe.
    On peut alors **les redéfinir**.
  - Dans ce cas, on utilisera le mot clé **`super`** pour faire référence à la méthode de la superclasse.

L'HÉRITAGE

- **La redéfinition de méthode**

Exemple : Redéfinir **debiter** pour la classe CompteEpargne

```
//méthode debiter redéfinie
public void debiter ( double montant ) {
   if ( montant <= getSolde ())
         super.debiter ( montant );
   else
         System.out.println ( " Débit non
   autorisé ");
}
```

L'HÉRITAGE

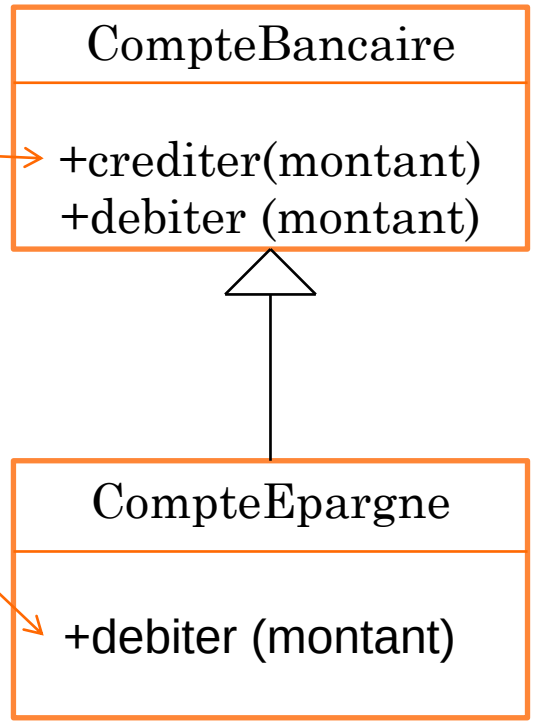
- **Polymorphisme**
  - Possibilité de manipuler des objets sans connaître leur type spécifique en utilisant des méthodes polymorphes.
  - Possibilité d'affecter un objet d'une sous-classe à une variable de la superclasse.
  - **Exemple :**

```
CompteBancaire c = new CompteEpargne(…);
```

L'HÉRITAGE

- **Quelles est la méthode appelée?**

```
CompteBancaire c =
new CompteEpargne('Dupont','4 Rue d\'Illier',
    0.01);
c.crediter(1000);
C.debiter(500);
```



C'est toujours la version la plus spécifique de la méthode qui est appelée par la JVM.

L'HÉRITAGE

- **Polymorphisme**
  - **Application : Une agence gère plusieurs comptes.**

```
CompteBancaire[] agence = new
    CompteBancaire[100];
nbComptes = 0;
CompteCourant c = new CompteCourant(…);
CompteEpargne e = new CompteEpargne(…);…
agence[0] = c; nbComptes++;
agence[1] = e; nbComptes++;…
for (int i =0;i<nbComptes; i++)
    agence[i].debiter(100);
```

L'HÉRITAGE

- **Sur chaque compte, un traitement quotidien est effectué:**
  - **Prélèvement des agios sur un compte courant**
  - **Versement des intérêts sur un compte d'épargne**

L'HÉRITAGE

- **Sur chaque compte, un traitement quotidien est effectué**
  - **Qu'ajoute-ton?**

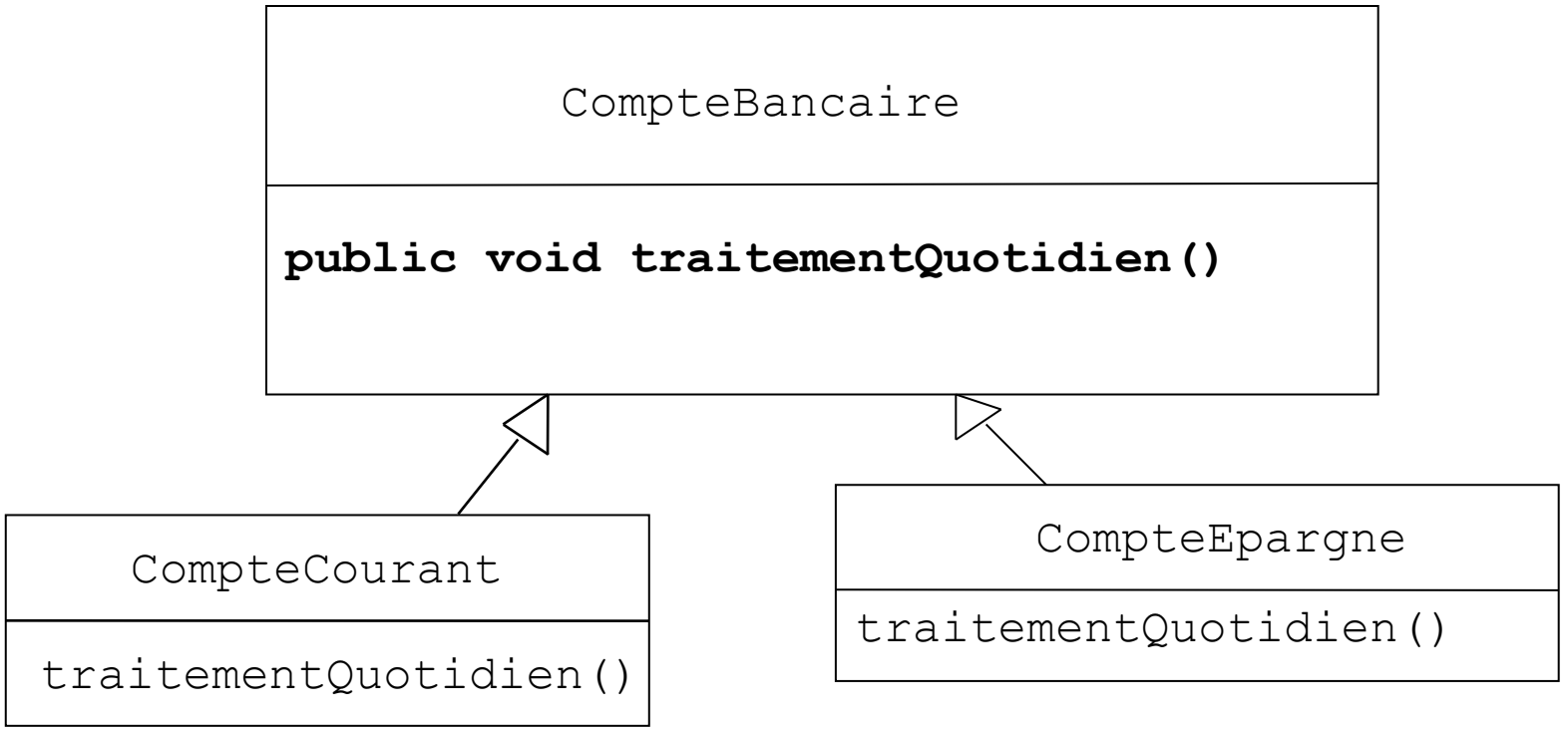
L'HÉRITAGE

- **Classes abstraites**
  - Une classe abstraite est une classe **qu'on ne pourra pas instancier.**
  - On la déclare avec le mot clé **`abstract`.**
  - Une classe abstraite pourra contenir certaines méthodes non implémentées, on parle de **méthodes abstraites.**
  - Si **on déclare une méthode abstraite** , la classe doit également être abstraite.

L'HÉRITAGE

- **Classes abstraites**
  - **Exemple :**
    - La classe `CompteBancaire` sera déclarée abstraite car la méthode `traitementQuotidien` est abstraite. Le traitement à appliquer dépend du type de compte et sera décrit dans les classes dérivées.
    - La classe `CompteBancaire` ne pourra pas être instanciée.

L'HÉRITAGE


CompteBancaire
public void traitementQuotidien()
CompteCourant
traitementQuotidien()
CompteEpargne
traitementQuotidien()

L'HÉRITAGE

Classe CompteBancaire modifiée :

```
public abstract class CompteBancaire {
    …
 public abstract void traitementQuotidien();
}
```


L'HÉRITAGE

- **Définition de la méthode `traitementQuotidien` dans chaque sous-classe :**

Classe CompteCourant :

```
public void traitementQuotidien () {
   if (getSolde () < 0) {
      debiter (-1.0*getSolde()*tauxAgios);
   }
}
```

L'HÉRITAGE

- **Définition de la méthode `traitementQuotidien` dans chaque sous-classe :**

Classe CompteEpargne:

```
public void traitementQuotidien () {
    crediter (getSolde()* tauxInterets);
}
```

L'HÉRITAGE

- **Classes abstraites et polymorphisme**
  - **Application : Une agence gère plusieurs comptes.**

```
CompteBancaire[] agence = new
    CompteBancaire[100];nbComptes = 0;
CompteBancaire c = new CompteBancaire(…);
CompteCourant c = new CompteCourant(…);
CompteEpargne e = new CompteEpargne(…);…
agence[0] = c; nbComptes++;
agence[1] = e; nbComptes++;…
for (int i =0;i<nbComptes; i++)
    agence[i].traitementQuotidien();
```

L'HÉRITAGE

- **La classe OBJECT**
  - Toutes les classes sont dérivées de la classe **`Object`.**
  - Chaque méthode de la classe `Object` est héritée par toutes les classes.
  - Méthodes importantes de la classe Object :
    - **`public boolean equals ( Object o )`**
    - **`protected Object clone()`**
    - **`public final Class getClass()`**
    - **`public int hashcode()`**
    - **`public String toString()`**

L'HÉRITAGE

- **La classe OBJECT – La méthode `equals`**
  - La méthode `equals` détermine si deux objets sont égaux ou non.
  - Son implémentation dans la classe Object vérifie que **les références d'objets sont identiques.**
  - On aura souvent besoin de redéfinir la méthode `equals` **: deux objets seront égaux quand ils auront le même état.**

L'HÉRITAGE

- **La classe OBJECT – La méthode `equals`**
  - Redéfinir la méthode `equals` pour la classe `CompteBancaire`

L'HÉRITAGE

```
public boolean equals(Object autre){
   //tester si les objets sont identiques
   if (this == autre) return true;
   //doit renvoyer false si le paramètre explicite
       vaut null
   if (autre == null) return false;
   //si les classes ne correspondent pas , elles ne
   //peuvent être égales
   if (getClass() != autre.getClass())
            return false;
     //autre est un objet de type CompteBancaire
     //non null
   CompteBancaire autreCompte
      =(CompteBancaire)autre;
     //tester si les comptes ont même numéro
   return this.numero==
      autreCompte.numero;
}    //fin de la méthode equals
```

L'HÉRITAGE

- **La classe OBJECT – La méthode `hashCode`**
  - Un code de hachage est un entier dérivé d'un objet.
  - Par défaut, le code de hachage est extrait de l'adresse mémoire de l'objet.
  - Des objets égaux ont toujours le même code de hachage.
  - Deux objets ayant le même code de hachage ne sont pas nécessairement égaux.

L'HÉRITAGE

- **La classe OBJECT – La méthode `clone`**
  - La méthode `clone` renvoie une copie de l'objet.
  - Si la sous-classe ne redéfinit pas cette méthode, elle effectue une copie champ par champ.

L'HÉRITAGE

- **La classe OBJECT – La méthode `toString`**
  - Il est conseillé de redéfinir la méthode `toString`.
  - L'appel de `System.out.print ( o )` pour un objet `o` va afficher le résultat de l'appel de `o.toString ()`.

L'HÉRITAGE

- **La classe OBJECT – La méthode `toString`**
  - **Méthode** `toString()` **de la classe** `CompteBancaire`

```
public String toString (){
return " Compte numéro " + numero + "
ouvert au nom de " + nom  + " \n Adresse
du titulaire " + adresse+ " \n Solde
actuel " + solde" ;
}
```

L'HÉRITAGE

- **La classe OBJECT – La méthode `toString`**

  - **Méthode `toString()` de la classe `CompteCourant`**

```
public String toString(){
return " Compte courant " +super.toString();
}
```

  - **Méthode `toString()` de la classe `CompteEpargne`**

```
public String toString(){
return " Compte d'épargne "+super.toString();
}
```

Les interfaces

- **Définition d'une interface**
  - Une interface peut être considérée comme une classe 100% abstraite.
  - Une interface décrit une fonctionnalité sous forme d'une liste de méthodes mais ne fournit aucune implémentation de ces méthodes.
  - Une interface ne peut contenir de variable d'instance.
  - Les seuls attributs autorisés sont les constantes déclarées `static` et `final`.
  - Une interface ne peut être instanciée.

Les interfaces

- **Définition d'une interface**
  - Toutes les méthodes d'une interface sont implicitement publiques.
  - Une interface peut étendre plusieurs interfaces.
  - **A partir de Java 8**, on peut ajouter deux éléments supplémentaires dans une interface:
    - des **méthodes statiques**
    - des **méthodes par défaut** qui sont des méthodes **concrètes**.

Les interfaces

- **Exemple d'interface définie dans l'API Java : l'interface `Comparable`**
  - C'est une interface générique paramétrée par le type `T`.
  - Cette interface permet de définir **une relation d'ordre** sur l'ensemble des objets d'un type `T`.

Les interfaces

- **Exemple d'interface définie dans l'API Java : l'interface `Comparable`**

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

L'appel de `x.compareTo(y)` doit renvoyer

- un entier négatif quand `x<y`,
- un entier positif quand `x>y`,
- `0` quand `x.equals(y)` renvoie `true` (et seulement dans ce cas).

Les interfaces

- **Implémentation des interfaces**

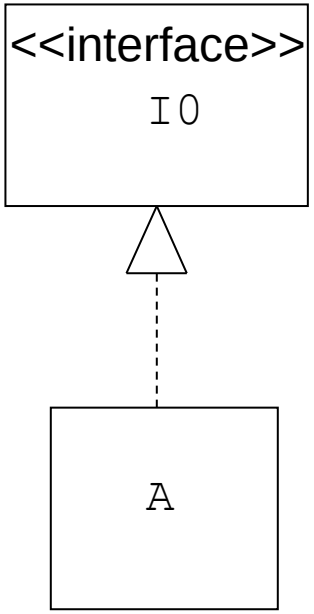
Une interface **définit un type**.

```
public interface I0{
      void m1 (…);
      void m2 ();
}

I0 varI0;
```

Une interface peut servir de type pour une variable, un attribut ou un paramètre.

Les interfaces



- **Implémentation des interfaces**

Comment obtient-on un objet possédant le type d'une interface?

1 - Définir une classe qui **implémente** l'interface

```
public class A implements I0{
        void m1 (…){/* … */}
        void m2 (){/* … */}
}
```

2 - Instancier cette classe

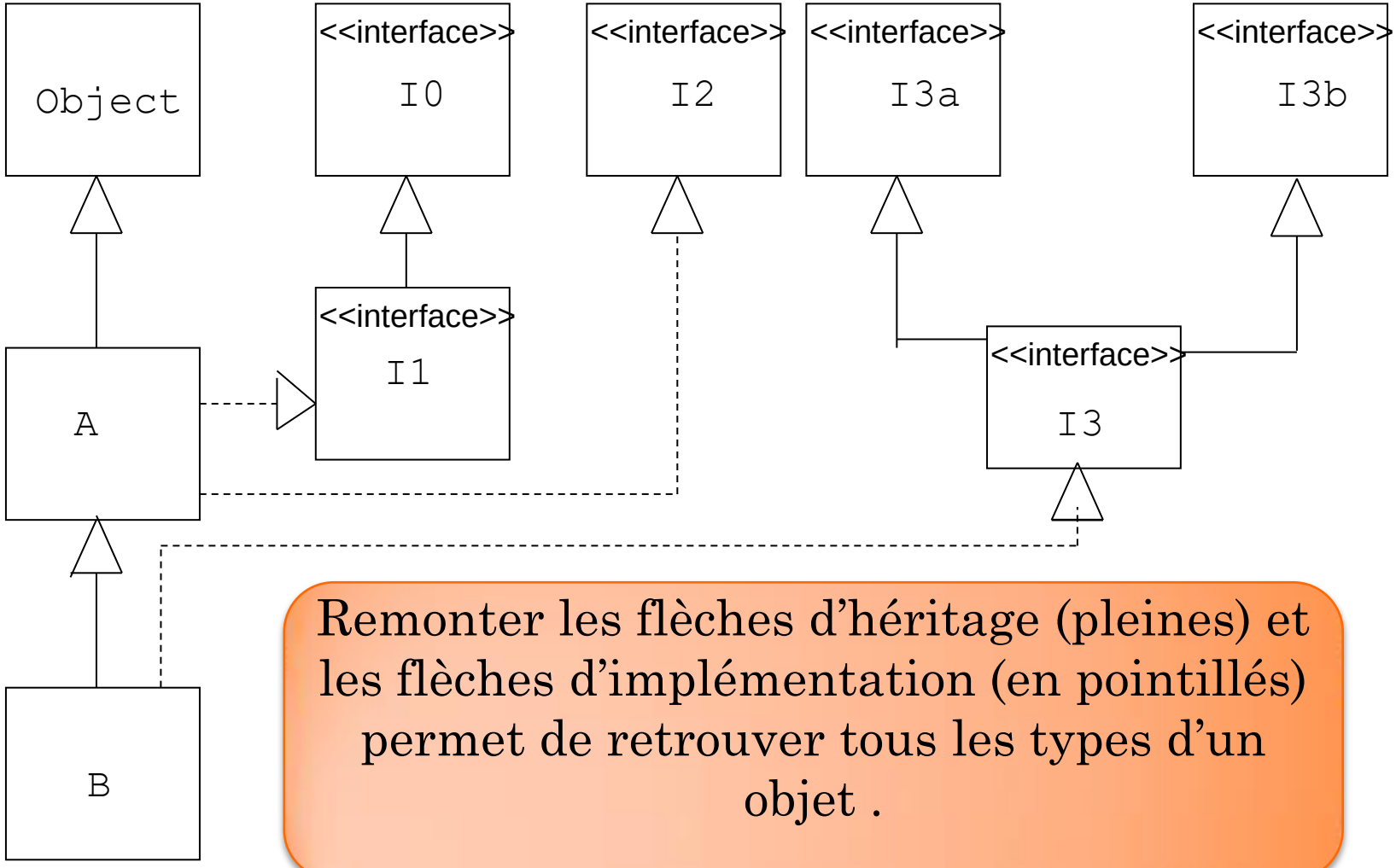
```
I0 varI0 = new A();
```

Les interfaces

- **Implémentation des interfaces**
  - On utilise le mot clé **`implements`** pour nommer la ou les interfaces qu'une classe implémente.
  - Une classe doit fournir une implémentation de toutes les méthodes de chaque interface qu'elle implémente excepté pour les méthodes par défaut. Sinon elle doit être déclarée abstraite.
  - Une méthode par défaut peut être redéfinie dans la classe qui implémente l'interface.
  - Une classe ne peut étendre qu'une seule classe mais peut implémenter plusieurs interfaces.

Les interfaces

- Polymorphisme



Remonter les flèches d'héritage (pleines) et les flèches d'implémentation (en pointillés) permet de retrouver tous les types d'un objet .

Les interfaces

- **Utilité des interfaces**
  - Les interfaces sont utilisées pour représenter des **propriétés transverses** de classes.
  - Exemple : Possibilité de définir une relation d'ordre sur des objets quelque soit leur type
    - Interface `Comparable<T>`
    - De nombreuses classes de l'API Java offrent des services pour des objets munis d'un ordre naturel.

Les interfaces

- **Utilité des interfaces**
  - Exemple : Dans la classe `java.util.Arrays` :

    `public static void sort (Object[] a)` trie les éléments du tableau passé en paramètre à condition que les éléments soient munis d'un ordre naturel, i.e. implémentent l'interface `Comparable`.

Les interfaces

- **Utilité des interfaces**

  Supposons que l'on souhaite trier un ensemble d'objets de type `CompteBancaire` suivant leur numéro.

  - Pour utiliser la méthode `public static sort (Object[] a)`, il faut que la classe `CompteBancaire` implémente l'interface `Comparable<CompteBancaire>`.

Les interfaces

- **Utilité des interfaces**
  - Exemple

```
public abstract class CompteBancaire
implements Comparable<CompteBancaire> {
    …
    public int compareTo (CompteBancaire o){
        return this.numero – o.numero;
    }
}
```

Les interfaces

- **Utilité des interfaces**
  - Exemple

```
import java.util.*;
public class TestTableauCompteBancaire {
  public static void main ( String[] args ){
     CompteBancaire c1 =
             new CompteCourant("Dupont", "a1",0.02);
     c1.crediter(5000);
     CompteBancaire c2 =
             new CompteCourant("Martin", "a2",0.02);
     c2.crediter(1000);
     CompteBancaire c3 =
             new CompteEpargne("Astruc", "a3",0.01,
  22950);
     c3.crediter(2000);
```

Les interfaces

- **Utilité des interfaces**
  - Exemple

```
CompteBancaire[] tableauDeComptes =
                                        {c3, c1, C2};
for (CompteBancaire compte : tableauDeComptes )
        System.out.println( compte);
Arrays.sort ( tableauDeComptes );
for (CompteBancaire compte : tableauDeComptes )
        System.out.println ( compte );
  }//fin du main
}//fin de la classe TestTableauCompteBancaire
```

Les interfaces

- **Utilité des interfaces**
  - Les interfaces sont utilisées **pour définir des types abstraits de données**.
  - Permet de séparer l'usage extérieur de l'objet de son implémentation.
  - Permet d'avoir plusieurs implémentations interchangeables d'une même interface.

Les interfaces

- **Utilité des interfaces**
  - **Exemple : le type de données abstrait Pile.**
    - On peut **réaliser une pile** avec un **tableau** ou une **liste chaînée**.
    - Réalisation d'une pile avec une liste chaînée:

```
class Maillon {
        Object valeur;
        Maillon suivant;

        public Maillon(Object s, Maillon m) {
          valeur = s;
          suivant = m;
       }
 }
```

Les interfaces

- **Utilité des interfaces**
  - Exemple : le type de données abstrait Pile.

```
public class Pile {
  private Maillon sommet;

  public Pile() {
  }
  public void empiler (Object s){
      sommet = new Maillon(s,
                                  sommet);
  }
public boolean estVide(){
    return (sommet == null);
  }
```

```
public Object depiler(){
     if (sommet ==null) return null;
     Object valeur =  sommet.getValeur();
     sommet = sommet.getSuivant();
     return valeur;
   }
   public Object sommet(){
      if (sommet ==nul) return null;
      return sommet.getValeur();
   }

}
```

Les interfaces

- **Utilité des interfaces**
  - Exemple : le type de données abstrait Pile.

  Pour **séparer l'usage de la pile de son implémentation**, on extrait l'interface publique de la classe :

```
public interface Pile {
   void empiler (Object s);
   boolean estVide();
    Object depiler();
    Object sommet();
}
```

Les interfaces

- **Utilité des interfaces**
  - Exemple : le type de données abstrait Pile.

  On rend privé les détails d'implémentation.

```
public class PileChainee implements Pile {
    private Maillon sommet;

    public Pile() {
    }
    public void empiler (Object s){
        sommet = new Maillon(s, sommet);
    }
    public boolean estVide(){
      return (sommet == null);
    }
```

Partout où une pile est requise, c'est par l'interface **Pile** qu'on la désignera.

Les interfaces

```
public Object depiler(){
   if (sommet ==null) return null;
    Object valeur =  sommet.getValeur();
    sommet = sommet.getSuivant();
    return valeur;
}
public Object sommet(){
    if (sommet ==nul) return null;
    return sommet.getValeur();
}
private class Maillon {
   Object valeur;
   Maillon suivant;

   public Maillon(Object s, Maillon m) {
     valeur = s;
     suivant = m;
   }
}
}// fin de la classe PileChainee
```

La classe `Maillon` est **interne** à la classe `PileChainee` et privée

Les interfaces

- **On souhaite réaliser un jeu permettant de se déplacer avec différents moyens de locomotion, voiture, moto, vélo, avion, train.**

  On sera amené à faire le plein dans une station service lorsqu'on utilisera une voiture ou une moto.

  On pourra aussi changer une roue d'un vélo, d'une moto et d'une voiture.

Les interfaces

- **Choisir les classes permettant de modéliser les différents moyens de locomotion.**
- **On souhaite maintenant ajouter la possibilité de se déplacer avec un jet ski.**