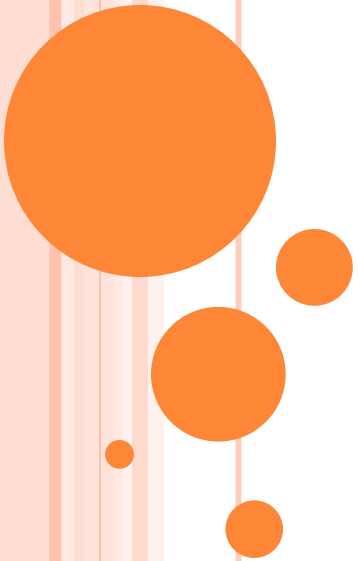


Module de mise à niveau en Java
Cours 2

Laure KAHLEM

laure.kahlem@univ-orleans.fr

Références

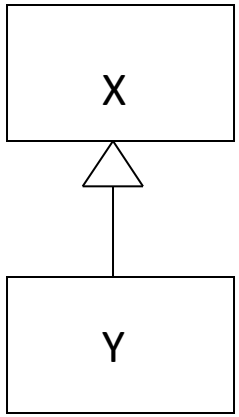
- Kathy Sierra et Bert Bates. Java Tête la première. O'Reilly, 2007.
- Mickaël Kerboeuf. Fondements de la programmation orientée objet avec Java8. Ellipses, 2016
- David Flanagan. Java in a Nutshell. Manuel de référence. O'Reilly, 2002.
- Cyrille Herby. Apprenez à programmer en JAVA. Broché, 2012.
-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

- L'héritage
- Les interfaces
- Les exceptions



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.**



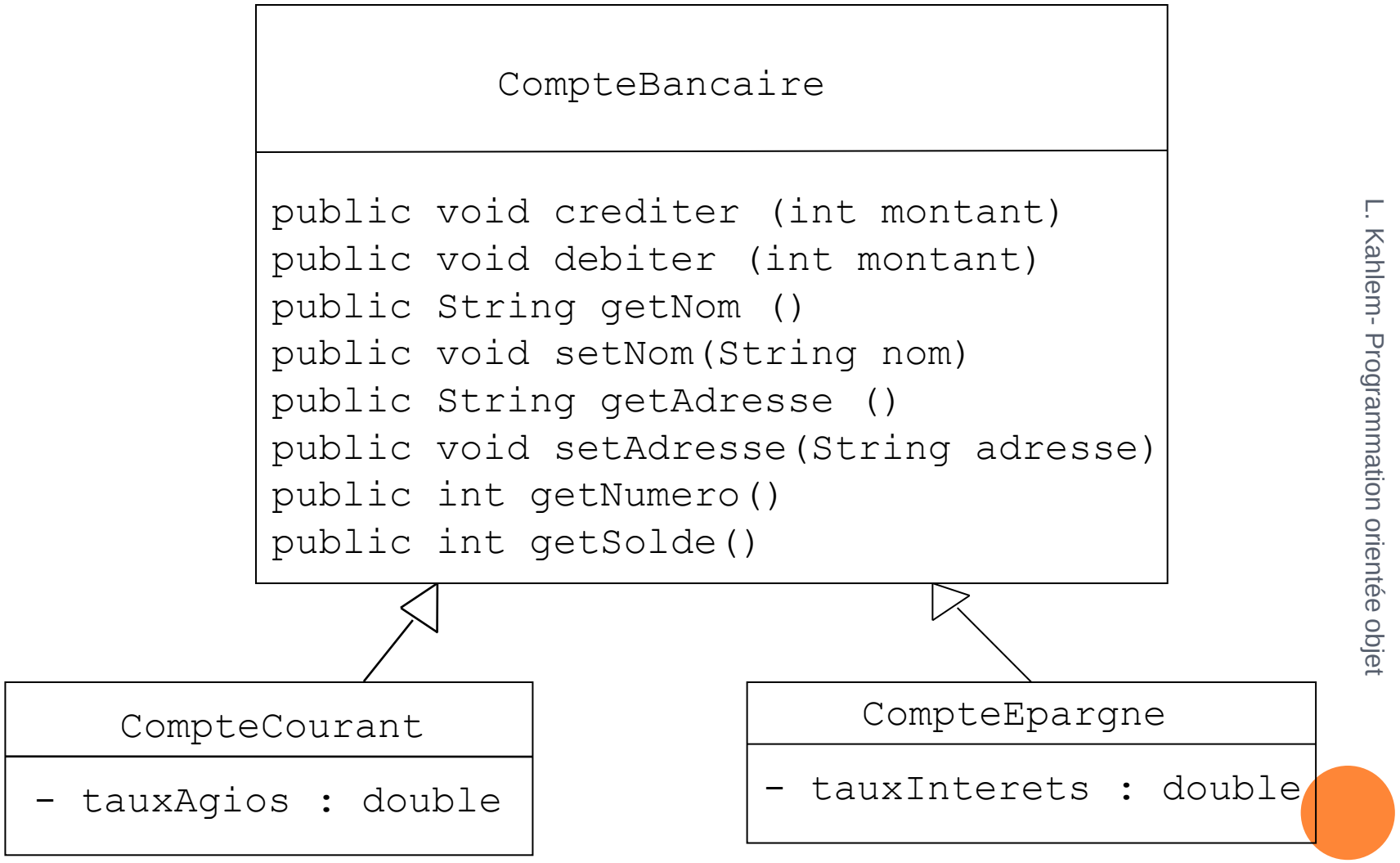
Classe parent ou superclasse

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

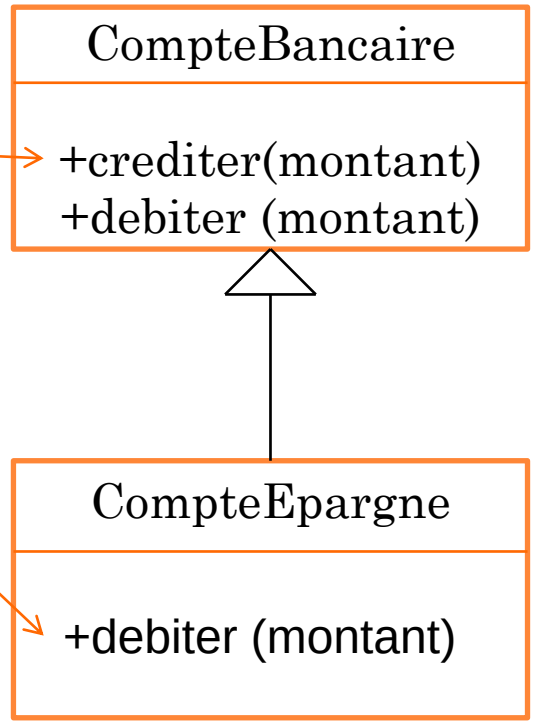
- **Polymorphisme**
  - Possibilité de manipuler des objets sans connaître leur type spécifique en utilisant des méthodes polymorphes.
  - Possibilité d'affecter un objet d'une sous-classe à une variable de la superclasse.
  - **Exemple :**

```
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.

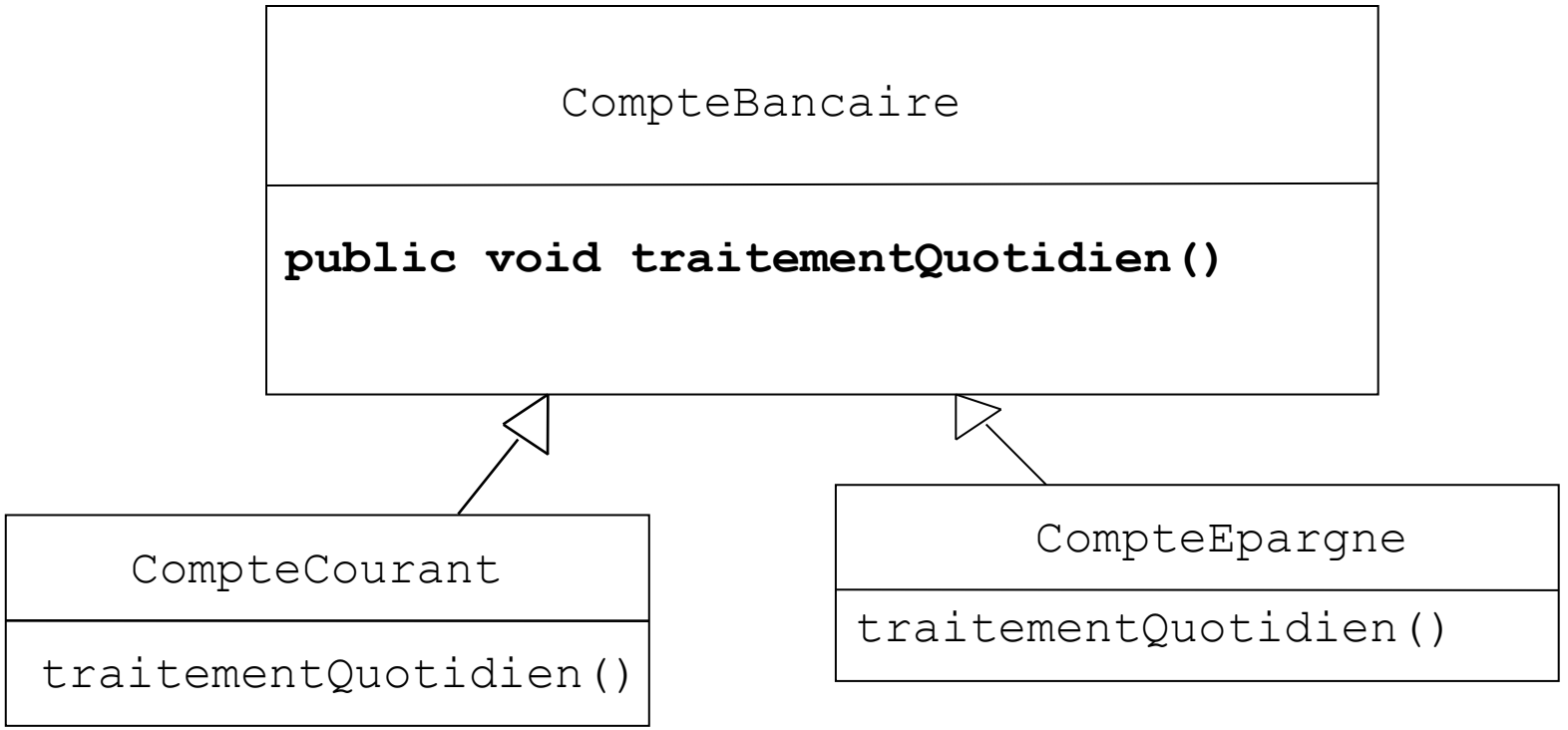


L'héritage

- **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

- Définir `Point3D` héritant de `Point` est-il une bonne idée?

```
public class Point {
 private int x;
 private int y;

 public Point(int x, int y) {
      this.x = x;
      this.y = y;
 }

 public double distance(Point
autre){
     …
   }
}
```

```
public class Point3D{
   private int x;
   private int y;
   private int Z;
        …
}
```

L'HÉRITAGE

- Définir `Point3D` héritant de `Point` est-il une bonne idée?

```
public class Point3D extends Point{
     private int Z;
…}
```




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 `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 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**

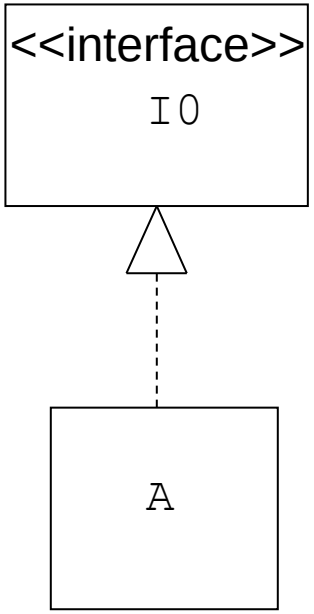
Une interface **définit un type**.

```
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

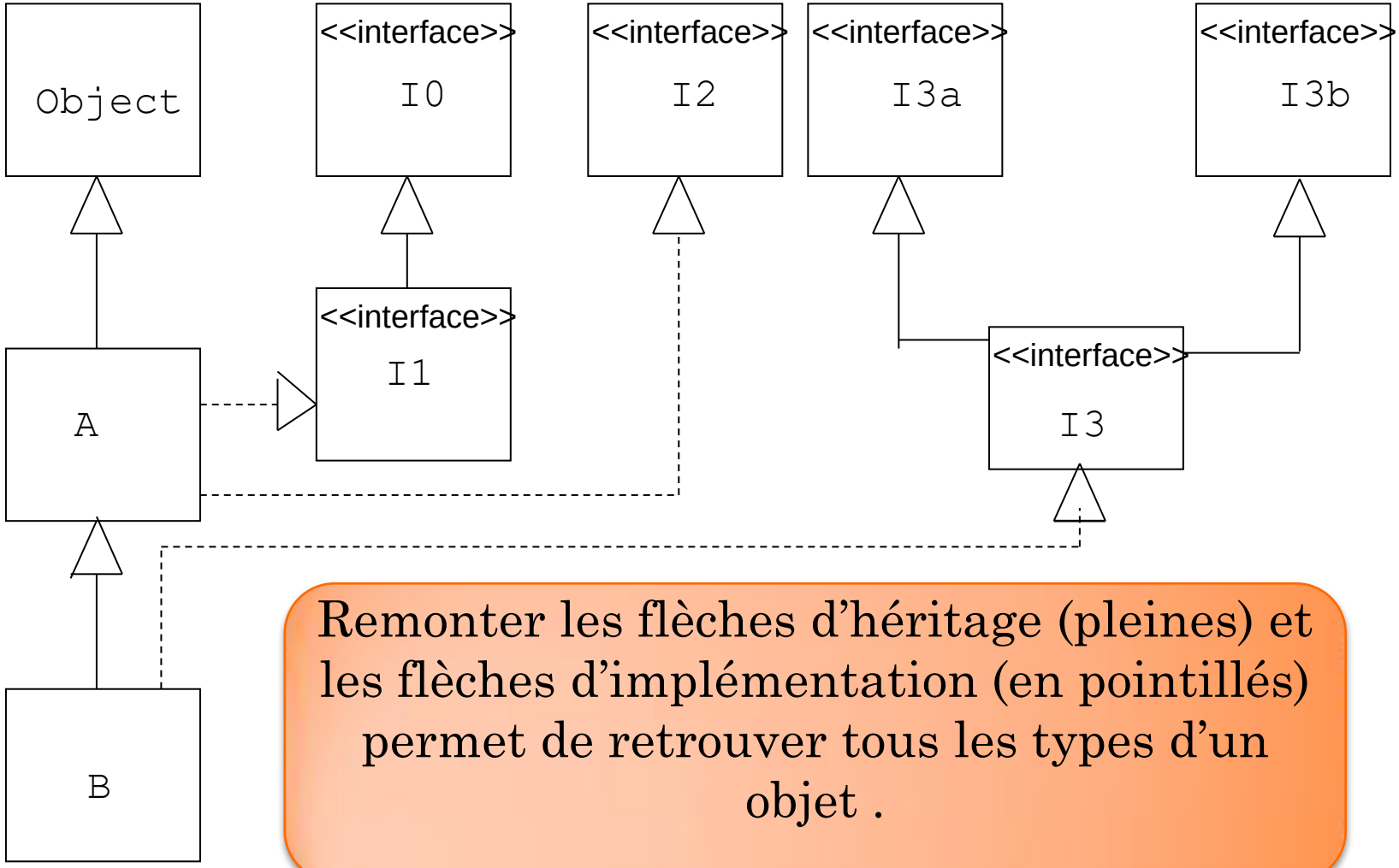
```
I0 varI0 = new A();
```

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 exceptions

- **Traitement des exceptions**
  - Une **exception** est une erreur qui survient à l'exécution d'un programme Java.
  - Java utilise un mécanisme d'interception des erreurs appelé **traitement des exceptions**.

Les exceptions

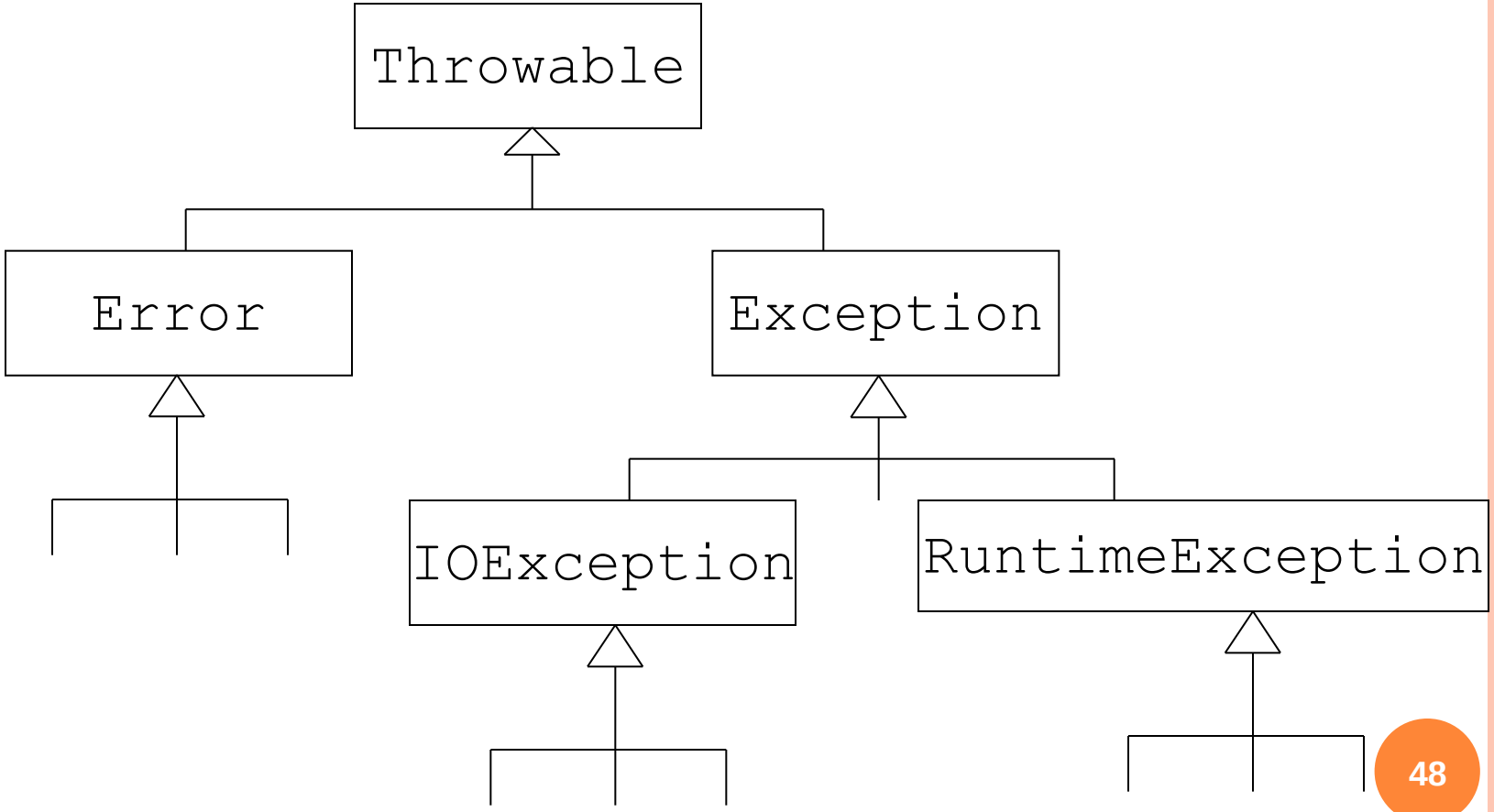
- **Traitement des exceptions**
  - Quand une erreur survient dans une méthode:
    - La méthode se termine immédiatement
      - Elle ne renvoie pas la valeur attendue
      - Elle ne renvoie aucune valeur.
    - Le contrôle est donné à des blocs de programmes spécifiques, **les gestionnaires d'exceptions** où elle est capturée et traitée.

Les exceptions

- **Exemples de causes d'erreurs**
  - **Les erreurs dues à l'utilisateur** qui ne respecte pas les règles métiers de l'application : demande de débit sur un compte d'épargne ayant un solde insuffisant.
  - **Les erreurs techniques** : connexion réseau défaillante, tentative de modification d'une clé primaire dans une base de données, imprimante déconnectée ou manquant de papier.
  - **Les contraintes physique** : disque plein.
  - **Les erreurs** de programmation : index d'un tableau invalide, emploi d'une référence à un objet non initialisé.

Les exceptions

- **Classement des exceptions**



Les exceptions

- **La hiérarchie `Exception`**

  Séparation en deux branches:

  - **Les exceptions dérivant de `RuntimeException`**

    Elles sont dues à une erreur de programmation et **devraient être évitées**.

  - **Les autres**

    Elles ont une cause externe au programme et sont potentiellement **récupérables**.

Les exceptions

- **La hiérarchie `RuntimeException`**
  - **Exemples de classes dérivant de `RuntimeException`:**
    - `ArithmeticException`
    - `IllegalArgumentException`
    - `IndexOutOfBoundsException`
    - `NullPointerException`
  - On peut toujours éviter une exception de ce type.

Les exceptions

- **Deux catégories d'exceptions**

Exceptions hors contrôle	Exceptions dérivant de la classe **`Error`** et de la classe **`RuntimeException.`**
Exceptions sous contrôle	Les autres exceptions.

Le compilateur contrôle que l'on apporte des gestionnaires d'exceptions pour toutes les exceptions sous contrôle.

Les exceptions

- Trois types d'opérations possibles

Opération	Sens	Instruction Java
Lever	Une méthode peut lever une exception pour indiquer à la méthode appelante qu'elle a demandé à réaliser une opération illégale.	`throw <instance_d'exception>`
Signaler/Relever	L'exception est interceptée par une méthode qui interrompt le flot normal de son exécution et renvoie l'exception à la méthode appelante.	`throws <nom_de_classe_d'exception>`
Gérer	L'exception est interceptée par le programme qui peut réaliser un traitement particulier puis reprendre son cours normal.	`try…catch(Exception…)`

Les exceptions

- **Signaler les exceptions avec la clause `throws`**
  - Une **méthode doit indiquer au compilateur** les valeurs qu'elle renvoie mais aussi signaler les anomalies possibles.
  - Quand on crée une méthode, on ne doit **signaler que les exceptions sous contrôle.**

Les exceptions

- **Signaler les exceptions avec la clause `throws`**

  Deux situations amènent une méthode à signaler un type d'exception dans la clause throws :

  - Soit la méthode appelle une méthode qui peut lancer une exception sous contrôle:
    - Soit la méthode **gère** l'exception : instruction **`try/catch`**
    - Soit la méthode ne gère pas l'exception et la **relève :** clause **`throws.`**
  - Soit on **lève** une exception sous contrôle dans le code avec l'instruction `throw`. Le type doit être signalé dans l'entête de la méthode avec la clause `throws`.

Les exceptions

- **Signaler les exceptions avec la clause throws**

Exemples de méthodes de l'API Java :

- Un des constructeurs de la classe `Scanner` qui prend en paramètre un ficher:

  ```
  public Scanner(File source) throws FileNotFoundException
  ```

  où `FileNotFoundException` dérive de `IOException`.

Les exceptions

- **Exemples de méthodes de l'API Java levant des exceptions de type RuntimeException**:
  - Dans la classe `Scanner` :
    - **`public int nextInt()`**
    - **Returns :** the int scanned from the input
    - **Throws :** `InputMismatchException`- if the next token does not match the *Integer* regular expression, or is out of range

      `NoSuchElementException`- if input is exhausted

      `IllegalStateException`- if this scanner is closed.

Les exceptions

- **Exemples de méthodes de l'API Java levant des exceptions de type RuntimeException**:
  - La méthode de l'interface `Comparable<T>`

    `int compareTo(T o)`

    Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

    Throws: `NullPointerException` - if the specified object is null

    `ClassCastException` - if the specified object's type prevents it from being compared to this object.

Les exceptions

- **Exemple**
  Une exception qui est levée va remonter la hiérarchie des méthodes appelantes qui la signalent avec une clause `throws` soit jusqu'à trouver une méthode qui la gère soit jusqu'à provoquer une erreur.

LES EXCEPTIONS

```
import java.lang.Exception.*;
public class ExempleNonRecup {
    public void m3() throws java.sql.SQLException{
       throw new java.sql.SQLException();
    }
    public void m2() throws java.sql.SQLException{
       this.m3();
    }
    public void m1() throws java.sql.SQLException{
       this.m2();
    }
    public static void main(String[] args) throws
    java.sql.SQLException{
       ExempleNonRecup exemple = new ExempleNonRecup();
       exemple.m1();
       System.out.println(" bonjour ");
    }
}
```

Les exceptions

- **Exécution du code**

```
utilisateur1> java ExempleNonRecup
Exception in thread "main" java.sql.SQLException
at ExempleNonRecup.m3(ExempleNonRecup.java:22)
at ExempleNonRecup.m2(ExempleNonRecup.java:25)
at ExempleNonRecup.m1(ExempleNonRecup.java:28)
at ExempleNonRecup.main(ExempleNonRecup.java:32)
```

LES EXCEPTIONS

```
import java.lang.Exception.*;
public class ExempleRecup {
    public void m3() throws java.sql.SQLException{
       throw new java.sql.SQLException();
    }
    public void m2() throws java.sql.SQLException{
       this.m3();
    }
    public void m1() throws java.sql.SQLException{
       this.m2();
    }
    public static void main(String[] args) {
       ExempleRecup exemple = new ExempleRecup();
       try {
              exemple.m1();
       }catch (SQLException e) {
    System.out.println(e.getMessage());
       }
       System.out.println(" bonjour ");
    }
}
```

Les exceptions

- **Exécution du code**

```
utilisateur1> java ExempleRecup
java.sql.SQLException
bonjour
```