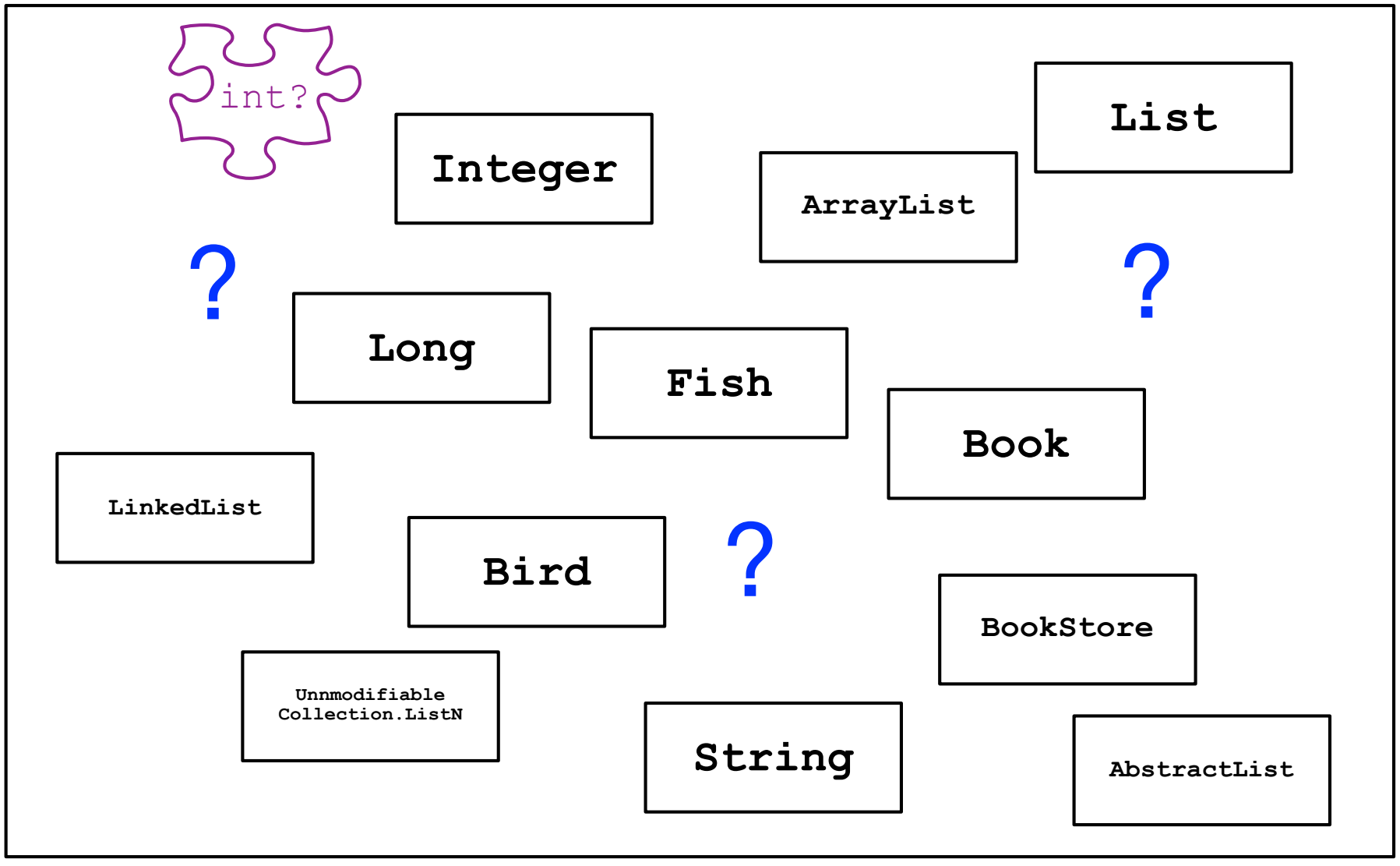
Programmation Objet. Cours 4

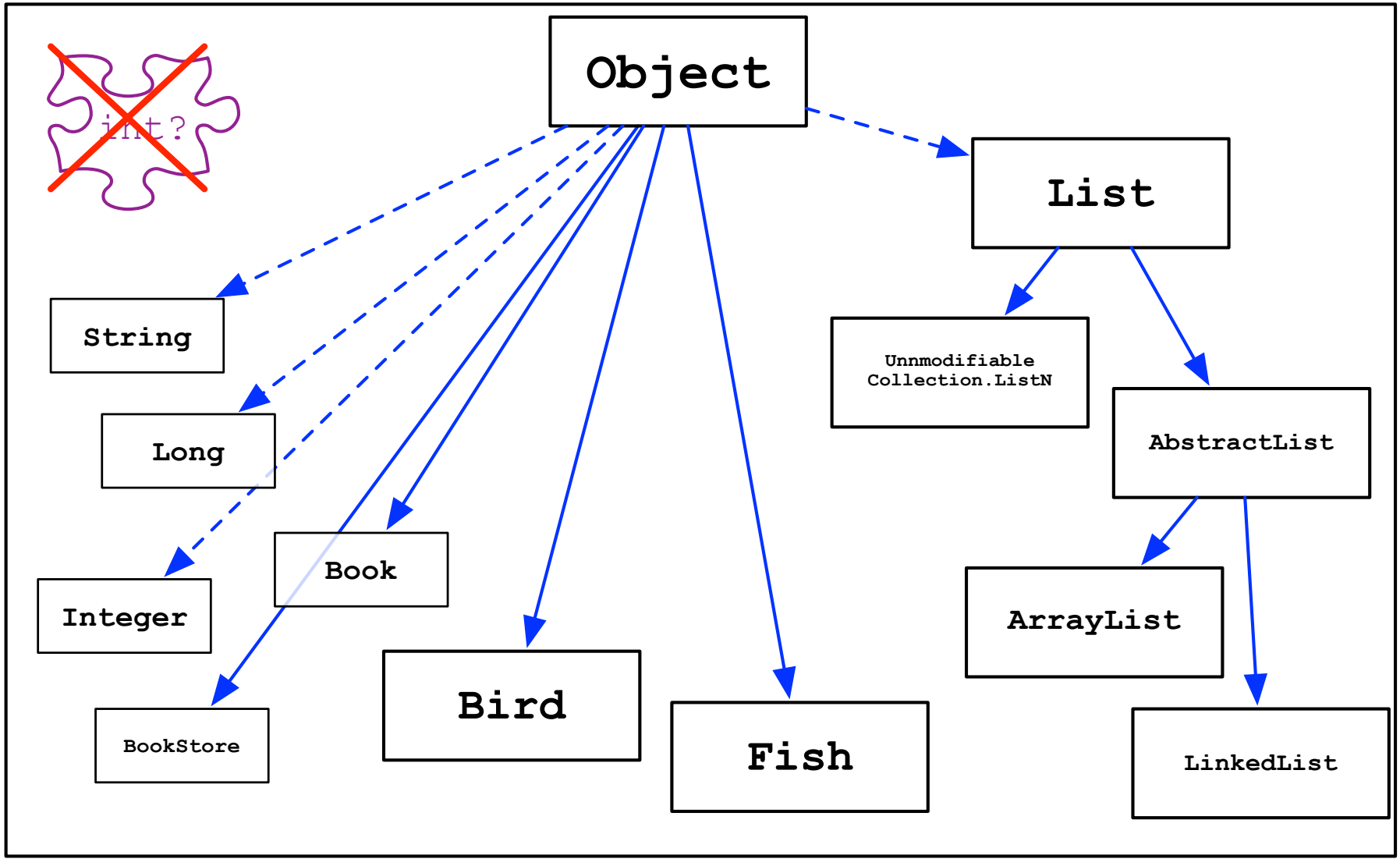
Marie-Pierre Béal et Carine Pivoteau
BUT 1

Hiérarchie des types.
Interface.
Polymorphisme.

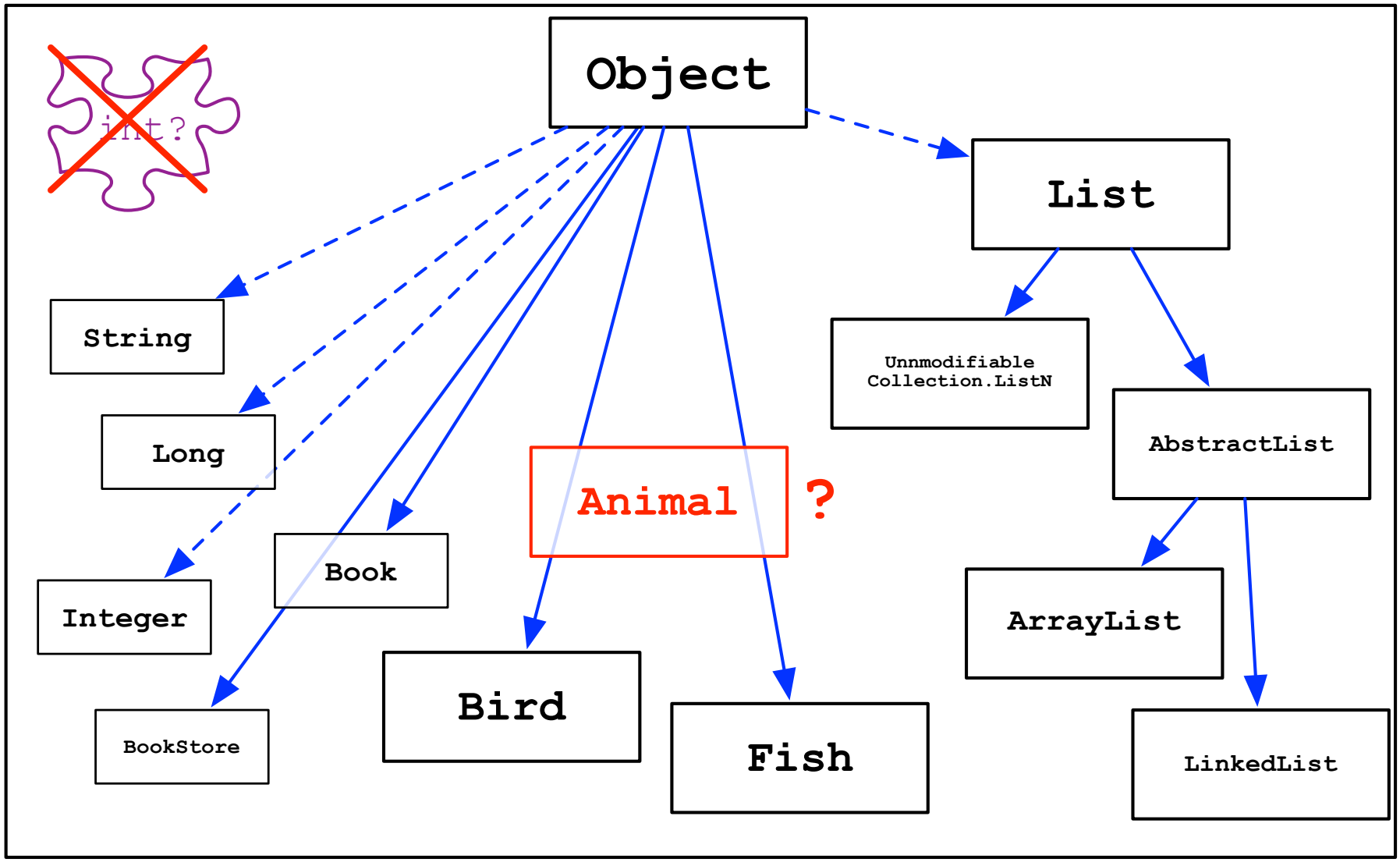
Hiérarchie des types



Au sommet de la hiérarchie : la classe `Object`



Comment hiérarchiser ?



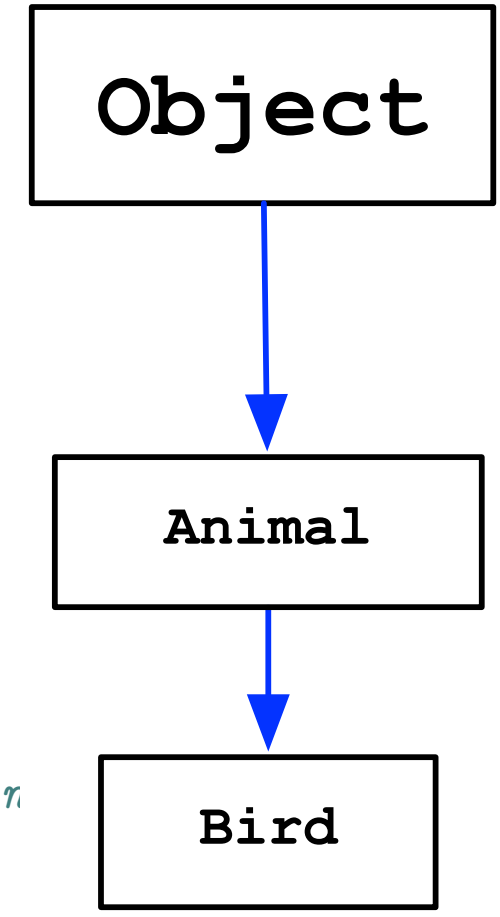
Supposons que Bird est un **sous-type** de Animal...

On parle de sous-typage en Java quand une variable de type A (par exemple Animal) contient une référence à un **objet** d'une classe moins élevée dans la hiérarchie (par exemple, Bird ou Fish).

```
public class Test{
  public static void main(String[] args) {
    Bird eagle = new Bird("Aigle", 90);

    Animal peacock = new Bird("Paon", 30);
    Object object = peacock;

    Bird bird = eagle;     // aucun problème
    Bird bird2 = peacock; // erreur de compilation
  }
}
```



Trans-typage pour les objets *(cast)*

Le **trans-typage** de référence *(cast)* est le fait forcer explicitement une référence à être d'un type donné, qui n'est pas nécessairement le type visible, ni le type réel (sa classe) de l'objet référencé.

- Cela modifie le type de la référence à un objet.
- Attention, **un *cast* ne change jamais le type réel d'un objet** ;

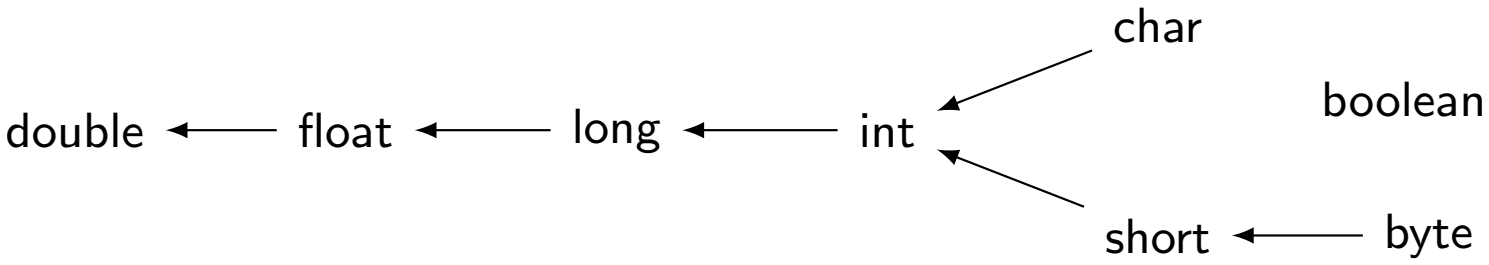
```
public static void main(String[] args) {
  Animal peacock = new Bird("Paon", 30);
  Bird bird = peacock; // erreur de compilation
  Bird bird = (Bird) peacock  // ok

  String string = "word";
  Integer number = 42;

  String s = number;           // erreur de compilation
  String s2 = (String) number; // ok à la compilation ...
                               // ...mais erreur à l'exécution
}
```

Trans-typage de primitifs

Pour les types primitifs, le **trans-typage** permet aussi la conversion d'une expression d'un certain type en une expression d'un type *plus élevé*, à condition qu'il y ait un lien de sous-typage entre eux :



Exemples :

```
int i = (int) 2.3; // cast explicite
double d = 3;      // cast implicite
int c = 'a' + 1;   // cast implicite

System.out.println('a' + 1); // 98
System.out.println( (char)('a' + 1) ); // b
```

Sous-typage : l'exemple des listes

La classe `ArrayList` est un sous-type de liste `List` (elle implémente l'*interface* `List`).

```
public static void main(String[] args) {
  List<String> dalton = List.of("Joe", "William", "Jack", "Averell");

  ArrayList<String> bronte = new ArrayList<String>();
  bronte.add("Charlotte");
  bronte.add("Emily");
  List<String> sisters = bronte; // ok

  ArrayList<String> brothers2 = dalton; // erreur de compilation
  ArrayList<String> brothers3 = (ArrayList<String>) dalton;
  // cette dernière ligne compile... mais fait une erreur à l'exécutio
}
```

Remarque : `List<Bird>` n'est pas un sous-type de `List<Animal>`.

Tester le type d'un objet avec l'opérateur `instanceof`

```
public static void main(String[] args) {
  Segment segment = new Segment(new Point(1, 1), new Point (2, 2));
  if (segment instanceof Object) {...}  // vrai
  if (segment instanceof Segment) {...} // vrai
  if (segment instanceof Point) {...}   // faux

  Animal phoenix = new Bird("phoenix", 200);
  if (phoenix instanceof Object) {...} // vrai
  if (phoenix instanceof Animal) {...} // vrai
  if (phoenix instanceof Bird) {...}   // vrai
  if (phoenix instanceof Fish) {...}  // faux

  if (null instanceof Object) {...}   // faux
}
```

L'usage de `instanceof` est restreint à des cas bien particuliers – dans ce cours, uniquement pour l'écriture de la méthode `equals()`. Il ne doit jamais se substituer au **polymorphisme** (suite du cours).

Sous-type : type des paramètres et types de retour

- Un méthode peut renvoyer un objet de n'importe quel sous-type du type de retour qu'elle déclare. Par exemple, un méthode qui doit renvoyer un `Animal` peut renvoyer un `Bird`, mais pas l'inverse.
- On peut passer en paramètre d'une méthode un objet de n'importe quel sous-type du type de paramètre qu'elle déclare. Par exemple, on peut passer un `Bird` en paramètre d'une méthode qui s'attend à recevoir un `Animal`, mais pas l'inverse.
- C'est le cas de la méthode `equals` dont l'en-tête est :

```
public boolean equals(Object o)
```

Cela permet d'obtenir le résultat suivant :

```
Animal animal = new Bird("phoenix", 200);
Bird bird = new Bird("phoenix", 200);
System.out.println(bird.equals(animal)) // true;
```

Une bonne façon d'écrire la méthode `equals`

Le paramètre de la méthode `equals` est obligatoirement un `Object` !

```
public class AnyClass
 ...
  @Override // voir cours suivant
  public boolean equals(Object o) {
    ...
```

Pour pouvoir utiliser cet `Object`, il faut s'assurer que c'est un objet de la bonne classe (ici, `AnyClass`) avec `instanceof`...

```
    if (!(o instanceof AnyClass)){
      return false;
    }
```

... puis utiliser un *cast* pour indiquer au compilateur que l'on sait que c'est un objet de la bonne classe :

```
    AnyClass t = (AnyClass) o;
```

Contrainte supplémentaire : définir hashCode

```java
public class Book {
  private final String title;
  private final String author;
  private int price;
  ...

  @Override
  public boolean equals(Object other) {
    if (!(other instanceof Book)) {
      return false;
    }
    Book book = (Book) other;
    return price == book.price && author.equals(book.author)
                               && title.equals(book.title);
  }

  @Override
  public int hashCode() { // cours suivant : "contrat objet"
    return Objects.hash(author, price, title);
  }
```

Qu'est-ce qu'un type commun ?

Dans nos exemples précédents :

- `Object` est le type commun de tous les objets.
- `List` est un type commun de `ArrayList`, `LinkedList`, ...
- `Animal` est un type commun de `Bird` et `Fish`.
- `Book` et `BookStore` n'ont pas d'autre type commun que `Object`.

Questions :

- Comment peut-on "créer" un type commun en Java ?
- À quoi ça sert ?
- Comment on décide si des objets ont un type en commun ?

Avoir un type commun, ça signifie avoir un comportement commun, c'est à dire des **méthodes en commun**.

Comment créer un type commun : avec une **interface**

En Java, pour signaler que des classes/records différents doivent avoir le même comportement, il doivent **implémenter la même interface**. Dans une interface, on trouve principalement :

- des méthodes d'instance abstraites : déclarées mais non définies,
- et des méthodes d'instance par défaut : définies et signalées mot-clé `default` (depuis java 1.8).

```
public interface NomDInterface{

  void exempleDeMethodeAbstraite(int i);

  default String exempleDeMethodeParDefaut(){
    return "exemple";
  }
}
```

Dans une interface, **toutes les méthodes sont publiques par défaut** (donc on peut omettre le mot-clé `public`).

Comment créer un type commun : avec une interface

Pour **implémenter une interface**, la classe ou le record doit :

- le signaler avec le mot-clé `implement`,
- et surtout implémenter les méthodes abstraites de l'interface.

```
public class NomDeClasse implements NomDInterface{
  ...
  @Override
  public void exempleDeMethodeAbstraite(int i){
    System.out.println(i);
  }
}
```

```
public record NomDeRecord(int n) implements NomDInterface{
  @Override
  public void exempleDeMethodeAbstraite(int i){
    System.out.println(n * i);
  }
}
```

Précisions sur les interfaces

Interface et sous-typage

Si une classe ou un record `A` implémente l'interface `I`, alors `A` est un sous-type de `I`.

Une interface est un type abstrait : **on ne peut pas faire de `new` !**

Une interface peut étendre une autre interface (mot-clé `extends`)).

Il est possible d'implémenter plusieurs interfaces différentes. Par exemple, la classe `String` implémente, entre autres, `Serializable`, `CharSequence`, `Comparable<String>`, ...

Enfin, une interface peut également contenir :

- des données constantes (`static final`),
- et des méthodes statiques,
- des méthodes d'instance privées.

Interface : l'exemple de Animal

```java
public interface Animal {
  public String move();
}
```

```java
public record Bird(String name, boolean flying) implements Animal{
  @Override
  public String move(){
    return flying ? "vole" : "se dandine";
  }
  @Override
  public String toString(){ return name; }
}
```

```java
public record Fish() implements Animal {
  @Override
  public String move(){
    return "nage";
  }
  @Override
  public String toString(){ return "poisson"; }
}
```

Quel intérêt ? Le polymorphisme !

```java
public static void main(String[] args) {
  Bird bird1 = new Bird("oiseau de paradis", true);
  Bird bird2 = new Bird("dinde", false);

  Fish nemo = new Fish()

  List<Animal> animals = List.of(bird1, bird2, nemo);

  for (Animal animal : animals){
    System.out.println(animal + " " + animal.move());
  }
  // oiseau de paradis vole
  // dinde se dandine
  // poisson nage
}
```

Et pour rajouter des insectes ? S'ils implémentent l'interface `Animal`, alors le code de la boucle `for` ne change pas.

Le polymorphisme

Le polymorphisme est une notion abstraite liée au sous-typage.

- Les objets d'un type A et de ses sous-typees (par exemple B et C) peuvent tous être vus comme des objets de type A (sous-typage).
- Les objets des types A, B et C ont au moins en commun les méthodes de A et on peut donc appeler ces méthodes sur n'importe lequel de ces objets.
- **Le polymorphisme est le mécanisme qui permet d'appeler les "bonnes méthodes" sur ces différents objets, même si on les considère tous comme étant de type A.**

Exemple géométrique

On crée une interface pour manipuler des formes géométriques 2D.

```
public interface Shape{
  static final double ONE_SQUARE_INCH = 0.155;
  void move(int dx, int dy);
  double area();
  default double squareInchArea(){ return ONE_SQUARE_INCH * area(); }
}
```

```
public class Disk implements Shape{
  private final Point center;
  private final int radius; // en cm
  ...
  @Override
  public void move(int dx, int dy){ center.move(dx, dy); }

  @Override
  public double area(){ return Math.PI * radius * radius; }
}
```

Exemple géométrique – suite

```
public class Rectangle implements Shape {
  private final Point upperLeft;
  private final Point lowerRight;
  ...
  @Override
  public void move(int dx, int dy){
    upperLeft.move(dx,dy);
    lowerRight.move(dx,dy);
  }
  @Override
  public double area() { return width() * height(); }
}
```

```
public static void main(String[] args) {
  Rectangle rectangle = new Rectangle(6,10);
  Disk disk = new Disk(new Point(0,0), 5);
  List<Shape> shapes = List.of(rectangle, disk);
  for (Shape shape : shapes){
    System.out.println(shape.area() + " cm" );
    System.out.println(shape.squareInchArea() + " sq in" );
  }
} // on peut rajouter des polygones... si on sait calculer leur aire.
```