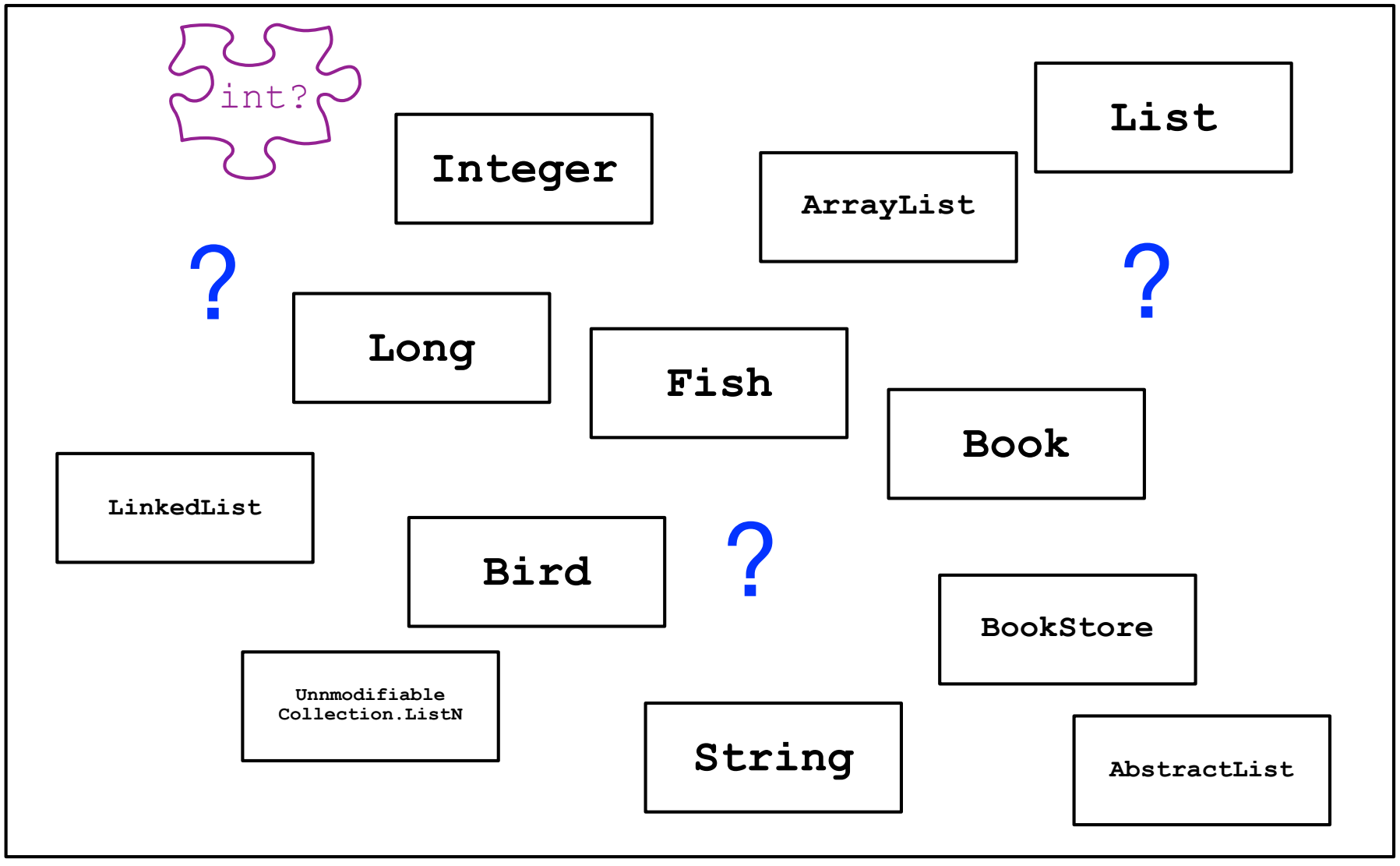
Programmation orientée objet. Cours 4

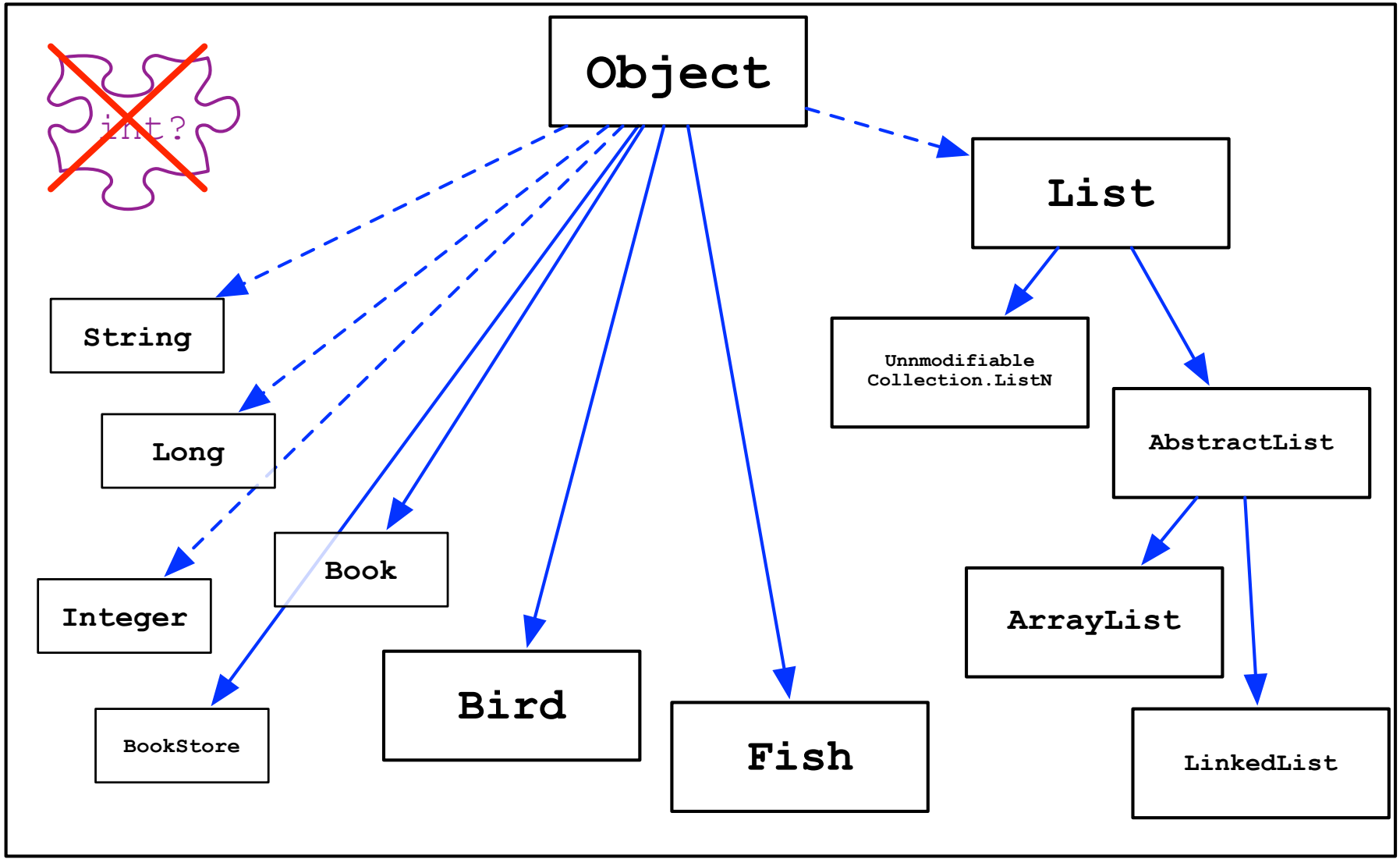
Marie-Pierre Béal et Carine Pivoteau
BUT 1

Hiérarchie des types.
Interface.
Polymorphisme.

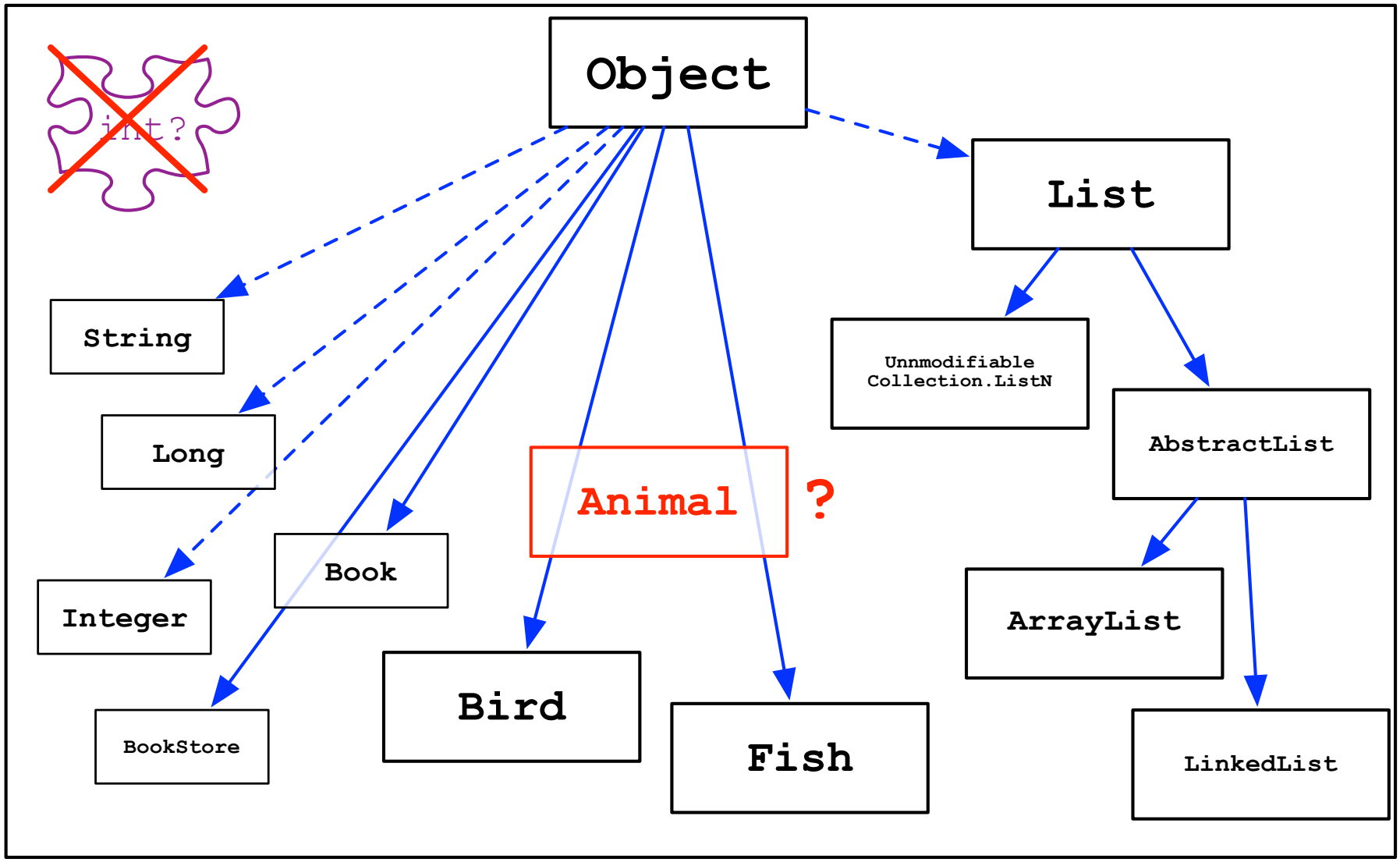
Hiérarchie des types



Au sommet de la hiérarchie : la classe `Object`



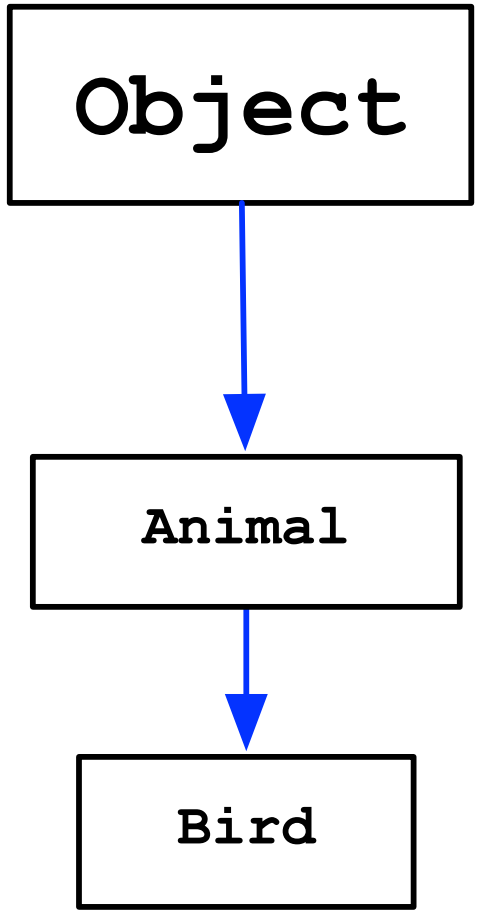
Comment hiérarchiser ?



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 = peacock; // erreur de compilation
  }
}
```



Trans-typage pour les objets (cast)

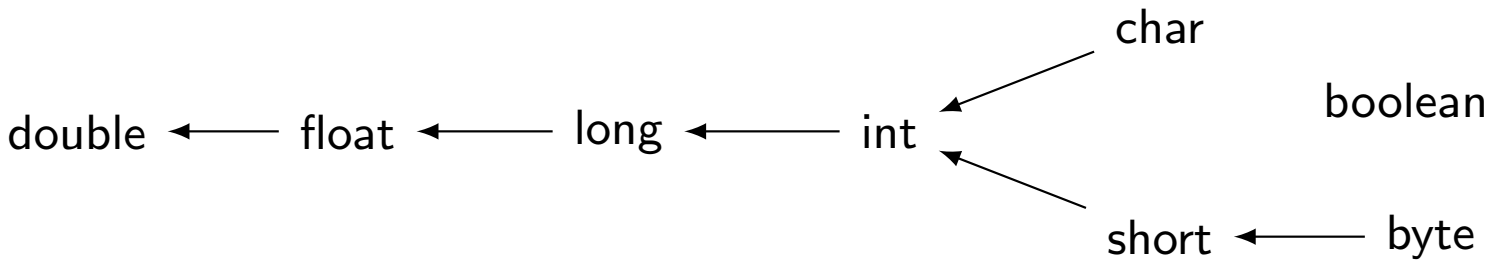
Le trans-typage avec cast est le fait forcer explicitement une référence à être d'un type donné, qui n'est pas nécessairement le type visible, ni le type réel (sa classe) de l'objet référencé.

- 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 `List` (elle implémente l'interface `List`).

```
public static void main(String[] args) {
  var dalton = List.of("Joe", "William", "Jack", "Averell");
  var bronte = new ArrayList<String>();
  bronte.add("Charlotte");
  bronte.add("Emily");
  List<String> sisters = bronte; // ok
  ArrayList<String> brothers1 = dalton; // erreur de compilation
  ArrayList<String> brothers2 = (ArrayList<String>) dalton;
    // cette dernière ligne compile
    // mais fait une erreur à  l'exécution
  }
```

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

- L'usage de `instanceof` est restreint à des cas bien particuliers.
- Dans ce cours, `instanceof` est interdit sauf 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

- Une 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 :

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

Écriture de equals dans une classe

Le paramètre de la méthode equals est obligatoirement un Object

```
public class AnyClass {

  @Override
  public boolean equals(Object o) {
    return o instanceof AnyClass anyClass
           && ... // tests sur anyClass
           ;
  }
  // il faut aussi écrire hashCode
}
```

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

Contrainte supplémentaire : définir hashCode

```
// ici on prendrait plutôt un record
// mais on prend une classe pour montrer comment
// écrire equals et hashCode
public class Book {
  private final String title;
  private final String author;
  private final int price;

  public Book(String title, String author, int price){
    this.title = Objects.requireNonNull(title);
    this.title = Objects.requireNonNull(author);
  }
....
```

Contrainte supplémentaire : définir hashCode

```
....
  @Override
  public boolean equals(Object other) {
    return other instanceof Book book
           && 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 MyInterface{

  void abstractMethod(int i);
  default String defaultMethod(){
    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 `implements`,
- et surtout implémenter les méthodes abstraites de l'interface.

```
public class MyClass implements MyInterface {
  ...
  @Override
  public void abstractMethod(int i){
    System.out.println(i);
  }
}
```

```
public record MyRecord(int n) implements MyInterface {
  @Override
  public void abstractMethod(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) {
  var bird1 = new Bird("oiseau de paradis", true);
  var bird2 = new Bird("dinde", false);
  var nemo = new Fish();
  var animals = List.of(bird1, bird2, nemo); // List<Animal>
  for (var 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-types (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;
  double area();
  default double squareInchArea(){ return ONE_SQUARE_INCH * area(); }
}
```

```
public record Disk(Point center, int radius) implements Shape{
  ...
  @Override
  public double area(){
    return Math.PI * radius * radius;
  }
}
```

Exemple géométrique – suite

```
public record Rectangle(Point upperLeft,  Point lowerRight)
                                                    implements Shape {
  ...
  @Override
  public double area() {
    return width() * height();
  }
}
```

Exemple géométrique – suite

```
public static void main(String[] args) {
  var corner1 = new Point(1, 2);
  var corner2 = new Point(3, 4);
  var rectangle = new Rectangle(corner1, corner2);
  var disk = new Disk(new Point(0,0), 5);
  var shapes = List.of(rectangle, disk);
  for (var 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.
```