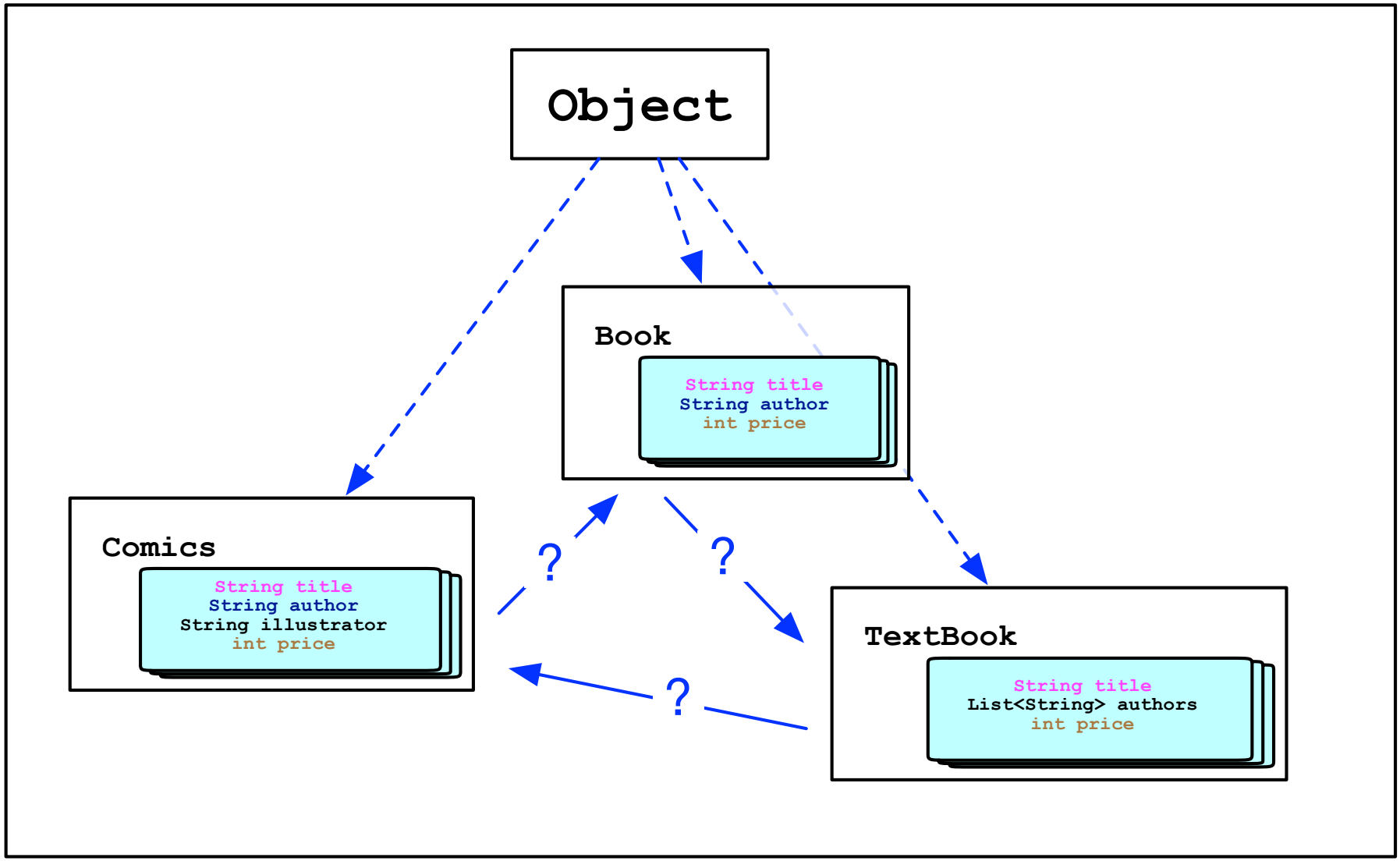
Programmation Objet. Cours 5

Marie-Pierre Béal et Carine Pivoteau
BUT 1

Héritage
Le contrat "objet"
Classe abstraite.

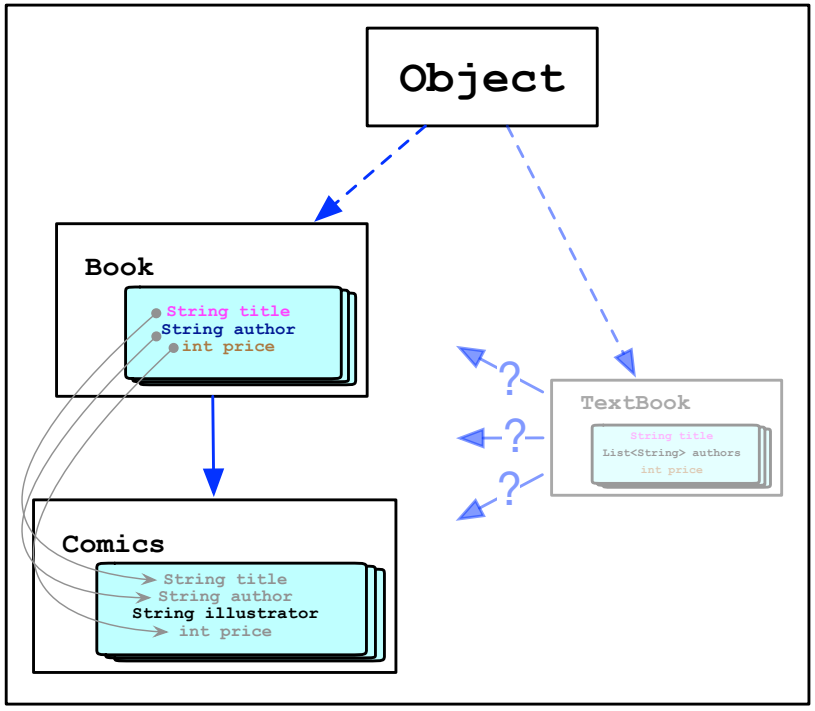
Hierarchie des types : le cas des objets qui ont beaucoup de points communs



Héritage

L'héritage consiste à faire profiter tacitement une classe dérivée C (par exemple `Comics`) des **champs** et des **méthodes** d'une super classe B (par exemple `Book`).

Cela signifie qu'un objet de la classe C peut être vu comme un objet de la classe B **avec des caractéristiques en plus**.



Autrement dit, **tout ce que peut faire un objet de B, un objet de C peut le faire aussi**, mais pas l'inverse.

Héritage et sous-typage

Si la classe C hérite de de la classe B, alors C est un **sous-type** de B.

Comment créer de l'héritage

En Java, pour signaler qu'une classe hérite d'une autre classe, on utilise le mot-clé `extends`

```
public class Heritiere extends UneSuperClasse{
  ...
}
```

- La classe dérivée possède les champs de la super classe (et peut y accéder **s'ils ne sont pas privés**).
- La classe dérivée **possède** les méthodes de la super classe (si elles ne sont pas privées).
- La classe dérivée peut déclarer de nouveaux champs et définir de nouvelles méthodes.
- Pas d'héritage avec les record : ils ne peuvent pas hériter d'une autre classe que `Record` et on ne peut pas hériter d'un record.

Exemple des points en 2D ou en 3D

```java
public class Point2D {
  private final int x = 1;
  private final int y = 2;

  @Override
  public String toString() {
    return "" + x + "," + y;
  }

  public String dim() {
    return "2D";
  }
}
```

```java
public class Point3D extends Point2D {
  private final int z;

  public Point3D() {
    z = 10;
  }

  @Override
  public String toString() {
    // return "" + x + ", " + y + "," + z;
    return "" + z;
  }
}
```

```java
  public static void main(String[] args) {
    Point2D p2 = new Point2D();
    Point3D p3 = new Point3D();

    System.out.println(p2.dim() + " : " + p2); // 2D : 1,2
    System.out.println(p3.dim() + " : " + p3); // 2D : 10
  }
```

Méthodes héritées et méthodes redéfinies

- Une classe B dérivée de A hérite de toutes les méthodes (visibles) de A. Un objet de la classe B peut donc directement utiliser ces méthodes, (*comme si elles avaient été définies dans la classe* B).
- Dans le code de la classe B, on peut faire appel aux méthodes héritées de la classe A grâce au mot-clé `super` : par exemple, dans la classe B, l'instruction `super.toString()` permet d'appeler la méthode `toString()` de A.
- La classe dérivée peut *redéfinir* (*override*, en anglais) une ou plusieurs méthodes de la super classe. Dans ce cas, la méthode redéfinie **remplace** la méthode de la super classe.
- Attention, une classe n'hérite pas des méthodes statiques !

Redéfinition et `@Override`

- Une méthode redéfinie doit avoir les mêmes nombres et types d'arguments et le même type de retour que la méthode héritée.
- L'annotation `@Override` sert à indiquer (et vérifier) qu'il s'agit bien d'une redéfinition de méthode.

```
public class UneSuperClasse{
  public void methode1(String s, int i){ ... }
  public void methode2(String s){ ... }
}
```

```
public class Heritiere extends UneSuperClasse{
  @Override
  public void methode1(String s, int i){ ... }
  // @Override : ne compile pas
  public void methode2(int s){ ... }
}
```

Remarque : sans l'annotation, ça compile, et la classe `Heritiere` possède deux `methode2` différentes. Ce n'est pas une redéfinition, c'est de la surcharge.

Amélioration de l'affichage des points en 3D

```java
public class Point2D {
  private final int x = 1;
  private final int y = 2;

  @Override
  public String toString() {
    return "" + x + "," + y;
  }

  public String dim() {
    return "2D";
  }
}
```

```java
public class Point3D extends Point2D {
  private final int z = 10;

  @Override
  public String dim() {
    return "3D";
  }

  @Override
  public String toString() {
    return super.toString() + "," + z;
  }
}
```

```java
  public static void main(String[] args) {
    Point2D p2 = new Point2D();
    Point3D p3 = new Point3D();

    System.out.println(p2.dim() + " : " + p2); // 2D : 1,2
    System.out.println(p3.dim() + " : " + p3); // 3D : 1,2,10
  }
```

Affichage des points en 3D, autre version

```java
public class Point2D {
  private final int x = 1;
  private final int y = 2;

  @Override
  public String toString() {
    return dim() + " : "
            + x + "," + y;
  }

  public String dim() {
    return "2D";
  }
}
```

```java
public class Point3D extends Point2D {
  private final int z = 10;

  @Override
  public String dim() {
    return "3D";
  }

  @Override
  public String toString() {
    return super.toString() + "," + z;
  }

}
```

```java
  public static void main(String[] args) {
    Point2D p2 = new Point2D();
    Point3D p3 = new Point3D();

    System.out.println(p2); // 2D : 1,2
    System.out.println(p3); // 3D : 1,2,10
  }
```

Constructeurs et super constructeurs

- Contrairement aux méthodes, une classe dérivée **n'hérite pas des constructeurs** de la super classe.
- Un constructeur doit toujours appeler un constructeur (visible) de la super classe, implicitement ou au moyen de `super(...)`.
  - Cette instruction doit être la première dans l'écriture du constructeur de la classe dérivée.
  - Si cette instruction est absente, c'est l'instruction `super()` qui est exécutée en premier.

```
public class UneSuperClasse{
  private final String name;
  public UneSuperClasse(String name){ this.name = name;}
}
```

```
public class Heritiere extends UneSuperClasse{
  public Heritiere(){
    super("heritière");
  }
}
```

Construction des points en 3D

```java
public class Point2D {
  private final int x;
  private final int y;

  public Point2D(int x, int y){
    this.x = x;
    this.y = y;
  }
  ...
}
```

```java
public class Point3D extends Point2D {
  private final int z;

  public Point3D(int x, int y, int z){
    super(x, y);
    this.z = z;
  }

  ...
}
```

```java
  public static void main(String[] args) {
    Point2D p2 = new Point2D(6, 6);
    Point3D p3 = new Point3D(4, 8, 16);

    System.out.println(p2); // 2D : 6,6
    System.out.println(p3); // 3D : 4,8,16
  }
```

Exemple des segments

```java
public class Segment {
   private final Point start;
   private final Point end;

   public Segment(Point start, Point end){
      this.start = Objects.requireNonNull(start);
      this.end  = Objects.requireNonNull(end);
   }

   @Override
   public String toString() {
       return "Segment [" + start + "," + end + "]";
  }
}
```

Exemple des segments colorés

```
public class ColoredSegment extends Segment{
  private final String color;

  public ColoredSegment(Point start, Point end, String color){
    super(start, end); //en premiere ligne du constructeur
    this.color = Objects.requireNonNull(color);
  }

  @Override
  public String toString() {
    return(super.toString() + ", color : " + color);
  }

  public boolean isRed(){
    return color.equals("red");
  }
}
```

Exemple des segments colorés – suite

```
public class Test{
  public static void main(String[] args) {
    Point p1 = new Point(0, 0);
    Point p2 = new Point(1, 1);

    Segment segment1 = new Segment(p1, p2);
    Segment segment2 = new ColoredSegment(p1, p2, "red");
    ColoredSegment segment3 = new ColoredSegment(p1, p2, "red");

    System.out.println(segment1);
    // Segment [(0,0), (1,1)]

    System.out.println(segment2);
    System.out.println(segment3);
    // Segment [(0,0), (1,1)], color : red
  }
}
```

Héritage - compléments

- Toute classe dérive directement ou indirectement de la classe `Object`.
- Toute record dérive directement de la classe `Record` et donc indirectement de la classe `Object`.
- L'arbre des dérivations est visible dans les fichiers créés par `javadoc` (dans les pages de documentation officielle, par exemple).
- La relation d'héritage est transitive.
- En Java, il n'y a pas d'*héritage multiple*, c'est à dire qu'une classe ne peut hériter directement que d'une seule classe. Par contre, il n'y a pas de limite au nombre de classes que l'on peut dériver à partir d'une même classe.

La classe java.lang.Object

```
protected Object          clone()
                          throws CloneNotSupportedException
public boolean            equals(Object obj)
public final Class<?>     getClass()

public int                hashCode()
public String             toString()
public final void         notify()
public final void         notifyAll()
public final void         wait()
public final void         wait(long timeout)
public final void         wait(long timeout, int nanos)
```

Le “contrat objet”

La méthode `hashCode()` de `Object` renvoie une référence au hasard et la méthode `equals()` teste seulement l'égalité des références (==).

`equals()` et `hashCode()`

Une classe/record **peut redéfinir la méthode** `equals()`. Dans ce cas, **il faut obligatoirement aussi redéfinir la méthode** `hashCode()`.

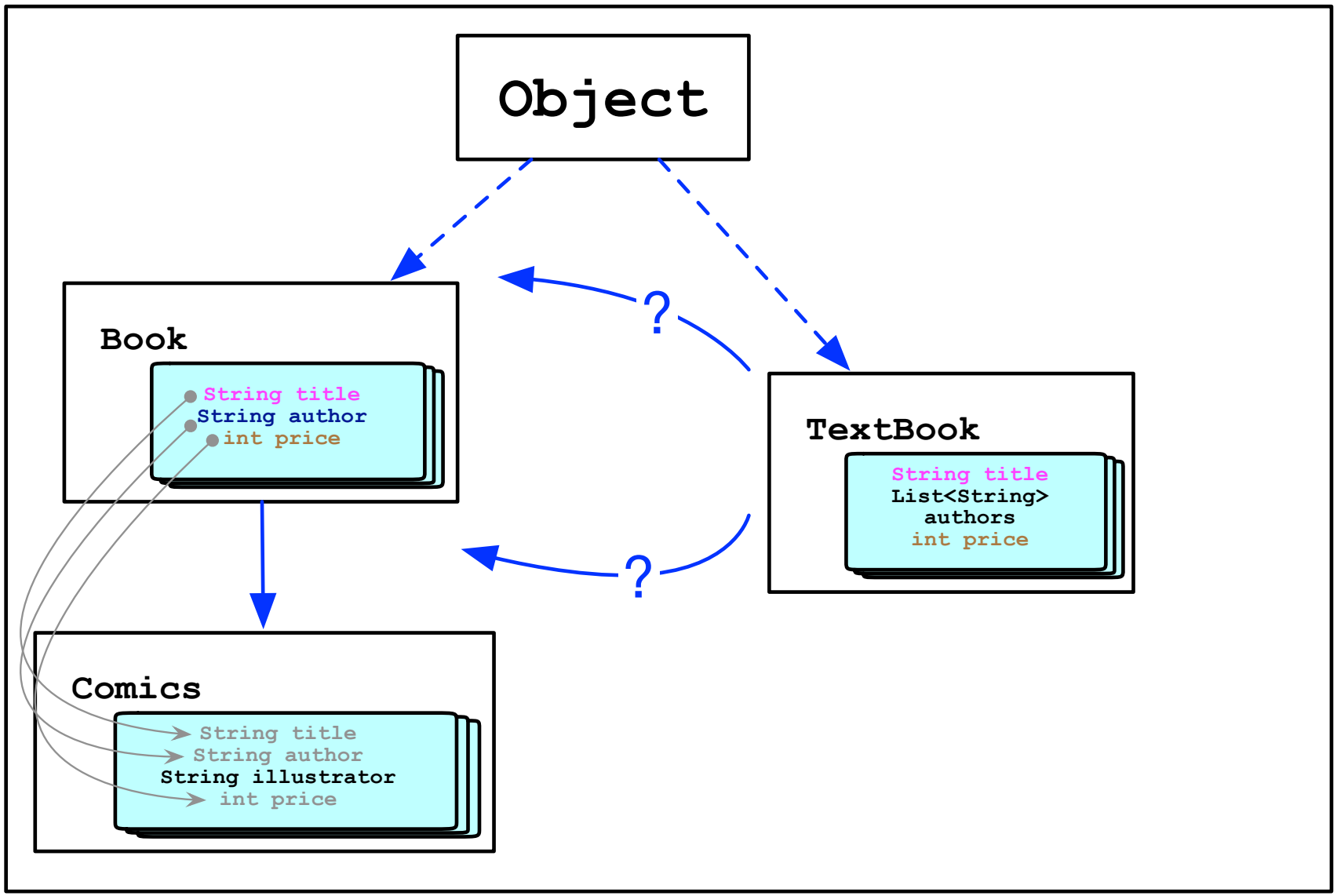
- la méthode `equals` doit définir une relation d'équivalence :
  - réflexive : x.equals(x) renvoie vrai.
  - symétrique : `x.equals(y)` est équivalent à `y.equals(x)`.
  - transitive : si `x.equals(y)` et `y.equals(z)` alors `x.equals(z)`.
- le test `x.equals(null)` doit être faux (si `x` est une référence) ;
- la méthode `hashCode` doit être consistante (plusieurs appels donnent le même résultat) ;
- Si deux objets sont égaux par `equals`, alors leur méthode `hashCode` doit donner la même valeur (mais pas la réciproque).

equals() / hashCode() et héritage

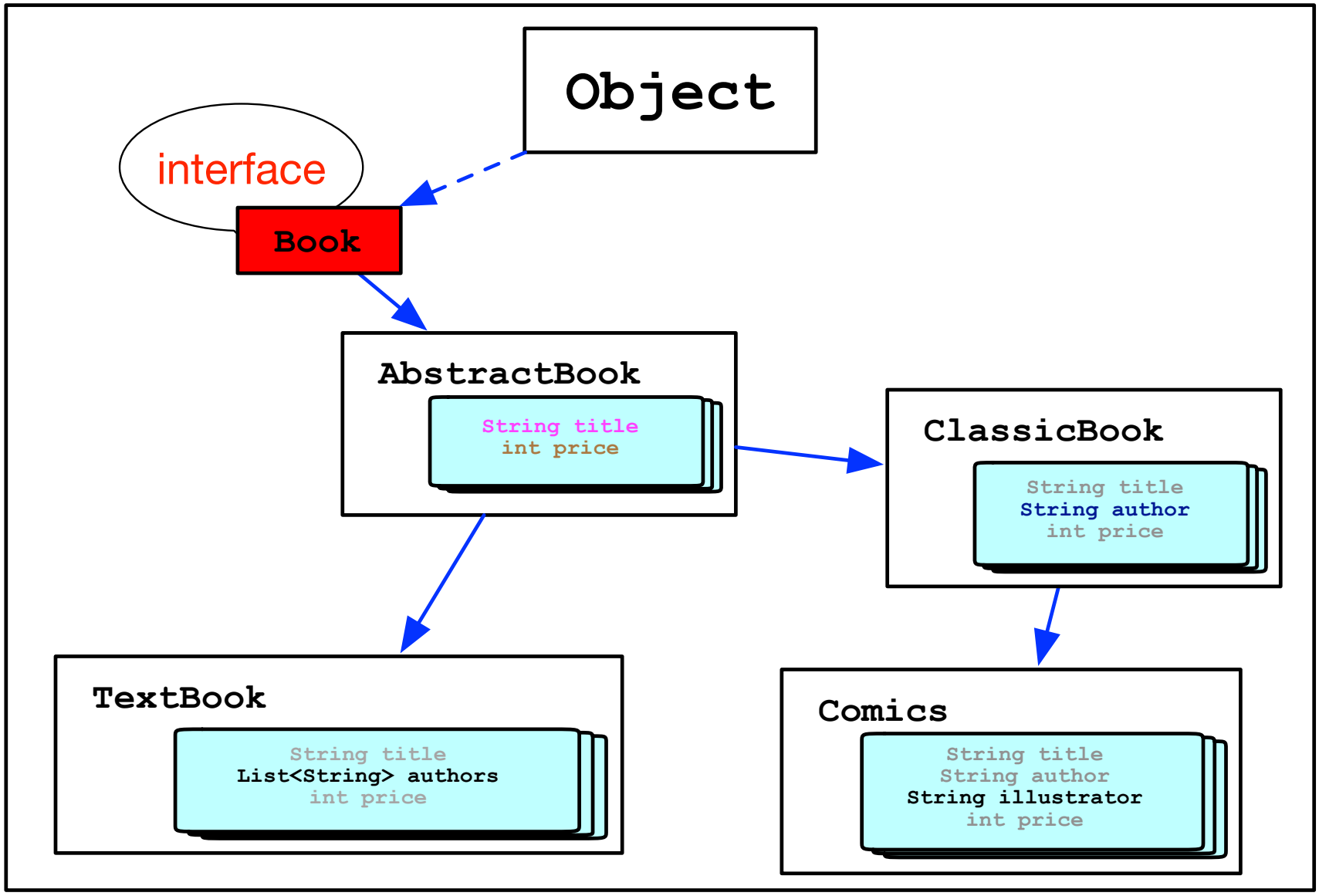
```java
public class Point{ ...
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Point)) { return false; }
    Point point = (Point) o;
    return point.x == x && point.y == y;
  }
  @Override
  public int hashCode() { return Objects.hash(x, y); }
```

```java
public class ColoredPoint extends Point{
  private final String color;
  ...
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof ColoredPoint)) { return false; }
    ColoredPoint coloredPoint = (ColoredPoint) o;
    return color.equals(coloredPoint.color) && super.equals(coloredPoint);
  }
  @Override
  public int hashCode() { return Objects.hash(xz, super.hashCode()); }
```

Hierarchie des types : beaucoup de points communs, mais pas vraiment de l'héritage, ni une interface



Hierarchie des types : les classes abstraites



Héritage : classes abstraites

En Java, pour créer une classe abstraite, on utilise le mot-clé `abstract`

```
abstract class AbstractBook {
  final String title;
  ...
}
```

Une classe abstraite sert en général à partager du code entre plusieurs classes (parties communes aux classes qui en dériveront).
C'est un choix d'implémentation, donc une classe abstraite n'a pas vocation à être publique. On lui donne une visibilité de package.

```
public class ClassicBook extends AbstractBook{
  ...
}
```

Attention, l'utilisation d'une **classe abstraite ne doit jamais se substituer à une interface !** C'est l'interface publique qui définit le type commun.

Classe abstraite - exemple

Une classe abstraite est une classe qui

- peut avoir des attributs,
- a un ou plusieurs constructeurs,
- peut avoir des méthodes concrètes ou abstraites,
- ne peut pas être instanciée (pas de `new`...).

```
abstract class AbstractShape implements Shape { // ne pas oublier l'interface !
  private final double width;
  private final double height; // bounding box (cm)

  AbstractShape(double width, double height) {
    this.width = width;
    this.height = height;
  }
  double boundingBoxArea() {
    return width * height;
  }
}
```

Classe abstraite vs. classe concrète

```java
public interface Shape{
  double area();
  default double squareInchArea(){ ... }
}
```

```java
public class Rectangle extends AbstractShape {
  public Rectangle(double width, double height) { super(width, height); }

  @Override
  public double area() {
    return boundingBoxArea();
  }
}
```

```java
public class Ellipse extends AbstractShape {
  public Ellipse(double width, double height) {  super(width, height); }

  @Override
  public double area() {
    return boundingBoxArea() * Math.PI/4;
  }
}
```

Exemple d'utilisation

```
public static void main(String[] args) {
  Shape rectangle1 = new Rectangle(6,10);
  Shape ellipse = new Ellipse(3,5);
  Rectangle rectangle2 = new Rectangle(32,2);

  System.out.println(rectangle1.area() + " cm2");
  System.out.println(ellipse.squareInchArea() + " square inches");
  System.out.println(rectangle2.squareInchArea() + " square inches");

  List<Shape> shapes = List.of(rectangle1, ellipse, rectangle2);

  for (Shape shape : shapes){
    System.out.println(shape.area()); // polymorphisme
  }
}
```

Classe abstraite = choix d'implémentation

Même interface :

```
public interface Shape{
  double area();
  default double squareInchArea(){ ... }
}
```

Implémentation différente :

```
abstract class AbstractShape implements Shape { // ne pas oublier l'interface !
  private final double boundingBoxArea;

  AbstractShape(double boundingBoxArea) {
    this.boundingBoxArea = boundingBoxArea;
  }

  double factor(); // méthode abstraite

  @Override
  public double area() {
    return boundingBoxArea * factor();
  }
}
```

Classe abstraite vs. classe concrète

```java
public class Rectangle extends AbstractShape {
  public Rectangle(double width, double height) { super(width * height); }

  @Override
  double factor() {
    return 1;
  }
}
```

```java
public class Ellipse extends AbstractShape {
  public Ellipse(double width, double height) { super(width * height); }

  @Override
  double factor() {
    return Math.PI/4;
  }
}
```

Et l'exemple d'utilisation... ne change pas ! Les choix d'implémentation ne sont pas visibles pour l'utilisateur.