Héritage, redéfinition & type abstrait

Le sous-typage

- L'idée du sous-typage est que:
  - Le comportement (méthode) dépend de l'objet réellement contenu dans la variable
    - L'affichage d'un objet est différent de l'affichage d'un Pixel, lui-même différent de celui d'une Personne
    - Mais tous peuvent s'afficher...
    - Ils disposent tous de la « méthode » toString()
    - Même exemple avec equals(), avec hashcode()

Le sous-typage

- Plus généralement, on voudrait avoir des **types**
  - Sur lesquels un **ensemble de méthodes est disponible (fonctionnalités)**
  - Mais dont la **définition exacte (comportement) dépend du sous-type**
  - La méthode finalement exécutée sera la plus précise possible
  - Exemple: toute figure a une surface, mais la surface d'un carré ne se calcule pas comme la surface d'un cercle...

Comment définir des sous-types

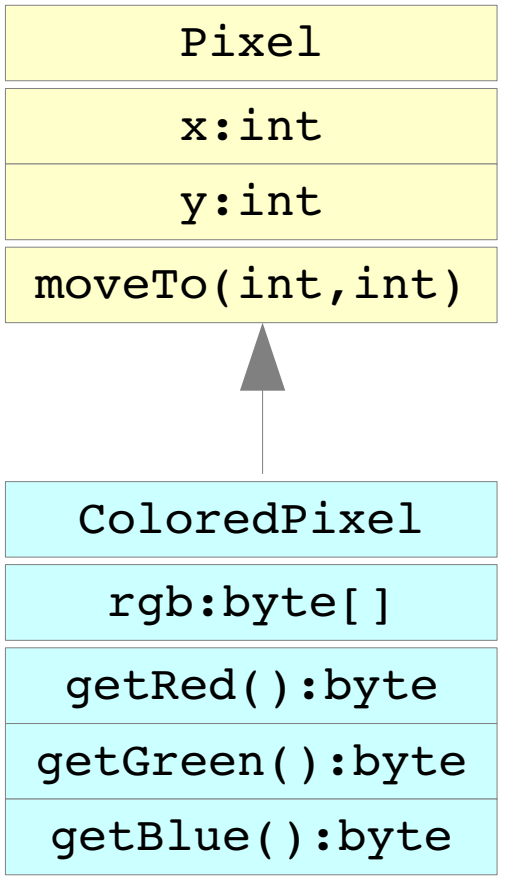
- On a vu les relations de conversions qui sont autorisées entre **types primitifs**
  - `byte < short < int < long < float < double`
    `char < int`
    - Ça n'est pas vraiment du sous-typage…
- On a vu que toute classe A hérite implicitement de la classe Object, et définit ainsi un type A qui est **sous-type** du type Object
  - Ça, oui, c'est du sous-typage

Comment définir des sous-types

- L'**héritage** définit des sous-types:
  - Soit explicitement:
    ```
    class Student extends Person { ... }
    ```
  - Soit implicitement :
    `Pixel` ou **`int[]`** héritent de **`Object`**
- L'**implémentation d'interface** définit aussi des sous-types
  - Une interface **déclare les méthodes applicables** par les objets des classes qui l'implémentent
  - Une **classe** implémente l'interface en **définissant ses méthodes**:
    ```
    class Carre implements Mesurable { ... }
    ```

L'héritage

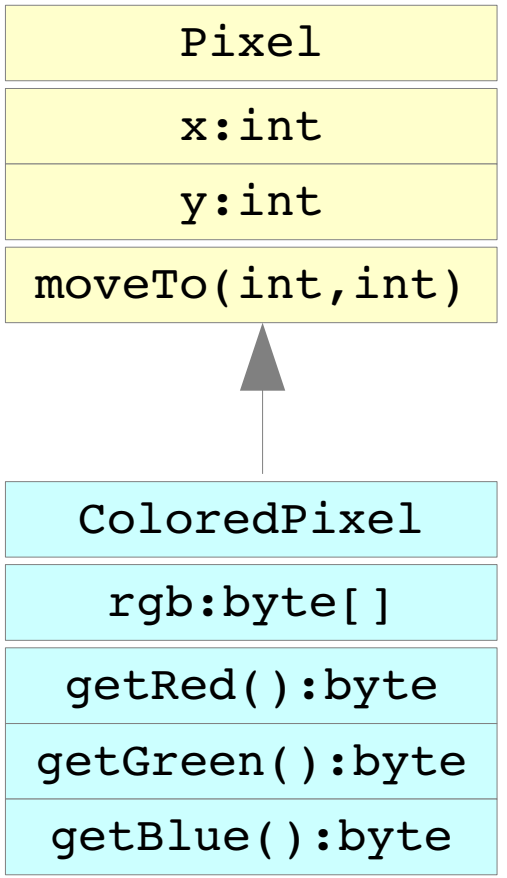
- Consiste à définir une classe,
  dite classe dérivée ou classe fille,
  à partir d'une autre classe,
  dite classe de base ou classe mère,
  en récupérant automatiquement
  dans la classe dérivée
  tous les membres
  de la classe de base,
  et en lui en ajoutant
  éventuellement
  de nouveaux membres



L'héritage

```java
public class Pixel {
    private int x;
    private int y;
    public void moveTo(int newX, int newY) {
        this.x = newX;
        this.y = newY;
    }
}
```

```java
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    public byte getRed() { return rgb[0]; }
    public byte getGreen() { return rgb[1]; }
    public byte getBlue() { return rgb[2]; }
}
```



Que sont les objets de la classe dérivée?

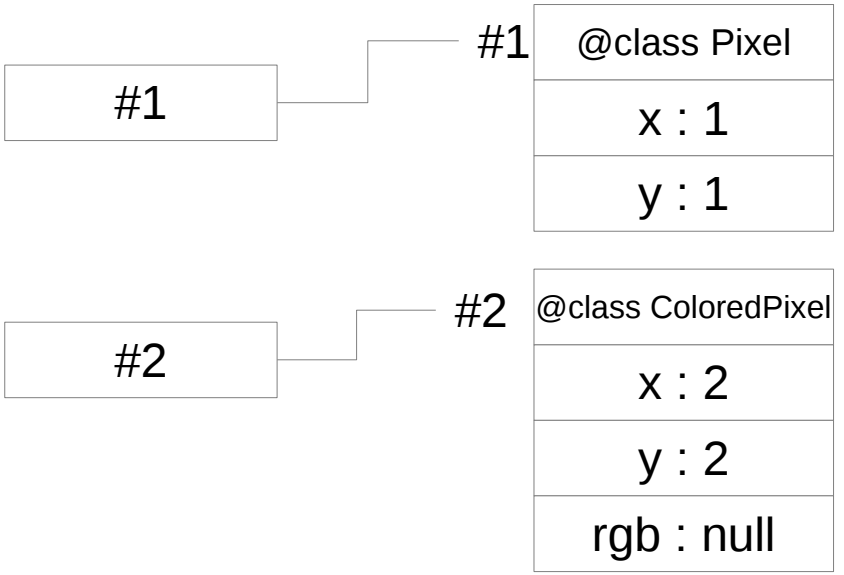
- Tout objet d'une classe dérivée est considéré comme étant avant tout un objet de la classe de base
  - Un pixel coloré « est » avant tout un pixel
- Tout objet d'une classe dérivée « cumule » les champs dérivés depuis la classe de base avec ceux définis dans sa propre classe
  - Il y a un int x et un int y dans un objet de la classe ColoredPixel

Que sont les objets de la classe dérivée?

```
public static void main(String[] args) {
    Pixel p = new Pixel();
    p.moveTo(1, 1);

    ColoredPixel cp = new ColoredPixel();
    cp.moveTo(2, 2);

    cp.getRed();  // compile, mais lève NullPointerException
                  // parce que le tableau n'a pas été alloué
}
```



Tous les champs sont hérités

- Ils peuvent être manipulés si leur accessibilité le permet
  - Si x n'est pas private dans Pixel, on peut dire this.x dans ColoredPixel
  - Il faut en général éviter d'avoir des champs non private

```
public class Pixel {
    int x;
    private int y;
    // ...
}
```

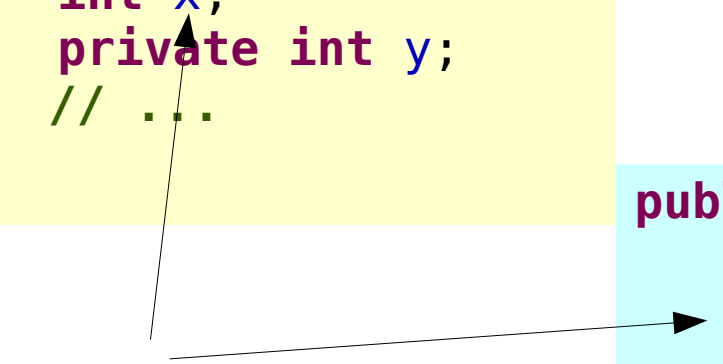
```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    void test() {
        System.out.println(this.x);  // 0
        System.out.println(this.y);  // compile pas !
                                     // field Pixel.y is not visible
    }
}
```

Tous les champs sont hérités

- Ils peuvent être **masqués** par la définition de champs qui ont le même nom dans la classe dérivée
  - Attention : c'est du « masquage » et pas de la « redéfinition » comme pour les méthodes...

```
public class Pixel {
    int x;
    private int y;
    // ...
}
```

```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    private String x;
    void test() {
        System.out.println(this.x);   // ??
    }
}
```



Tous les champs sont hérités

- Si String x est déclaré dans ColoredPixel, c'est celui qui sera considéré dans cette classe quand on parle de this.x
- Il est possible de manipuler celui qui est masqué (s'il est accessible) par la notation **super.x**
- super a la même valeur que this à l'exécution mais est typé comme la super classe (ici Pixel)

```
public class Pixel {
    int x;
    private int y;
    // ...
}
```

```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    private String x;
    void test() {
        System.out.println(this.x);   // null
        System.out.println(super.x);  // 0
    }
}
```

Résolution du champ à accéder

- La **détermination du champ** qui doit être accédé s'appelle « **la résolution** »
    - savoir où on ira chercher la valeur à l'exécution
- La résolution des **champs** est effectuée par le compilateur, en **fonction du type déclaré de la variable** qui contient la référence

```
public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel();
    // le type déclaré de cp est ColoredPixel
    System.out.println(cp.x);   // null

    Pixel p = cp;
    // le type déclaré de p est Pixel, même si la référence
    // contenue dans p est celle d'un ColoredPixel
    System.out.println(p.x);    // 0
}
```

Le masquage des champs

- Avoir un champ qui a le même nom qu'un champ d'une superclasse est en général une ***mauvaise idée***
  - **super**, c'est **this** vu avec le **type de la super-classe**
  - **super.super.x** n'existe pas...
  - Pas plus que ref.super ni ref.super.x...
- En revanche, le **transtypage** (***cast***) permet d'accéder en changeant le type déclaré de la référence ref

```
class A {
  int x = 1;
}

class B extends A {
  String x = "zz";
}

class C extends B {
  boolean x = true;
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);       // true
    System.out.println(((B)c).x); // zz
    System.out.println(((A)c).x); // 1
  }
}
```

Constructeurs et héritage

- La construction (initialisation) de toute instance d'une classe débute par la construction (initialisation) d'une instance d'Object
  - En pratique, tout constructeur débute par un appel au constructeur de sa super-classe: **`super()`**

```
public class Pixel {
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }
   // ...
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  public ColoredPixel(int x, int y) {
    super(x, y); // notez que x et y sont private!
    rgb = new byte[3];
  }
}
```

Constructeurs et héritage

- **`super()`**
  - Doit obligatoirement être la première instruction du constructeur
  - Le constructeur implicite (ajouté par le compilateur) fait appel au constructeur sans argument de la super-classe
- On n'hérite pas des constructeurs

```
public class Pixel {
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }
   // ...
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  public ColoredPixel() { // Compile pas !
    // super(); // Constructeur Pixel() is undefined
  }
}
```

Constructeurs et initialisations

- L'appel au constructeur est une étape dans l'initialisation des objets de cette classe:
  - Commence par l'initialisation des champs de l'objet « en tant qu'instance de la super-classe »: c'est l'appel à super(...)
  - Ensuite il se charge d'initialiser les objets qui lui sont propres (en tant qu'instance de la classe dérivée)
  - L'appel à super ne peut pas utiliser des champs dont l'existence ou la valeur dépendrait de l'instance de la classe dérivée...

```
public class ColoredPixel extends Pixel {
  private int v = 0;
  private static int s = 0;
  public ColoredPixel() {
    // super(v,v);
    // error: cannot reference v before supertype constructor has been called
    super(s,s); // OK
  }
}
```

L'héritage des méthodes

- En plus des champs, en tant que « membres », la **classe dérivée hérite des méthodes** de la classe de base
- Seuls **les constructeurs ne sont pas hérités**
  - Ils restent propres à leur classe
- **Attention**: le **code** (sémantique) d'une méthode de la super classe **peut ne plus être correct** dans la classe dérivée
  - Pixel::moveTo() est correcte ColoredPixel mais pas Pixel::equals() ni Pixel::toString()
- Dans certains cas, il faut **donner une nouvelle définition** de la même méthode à utiliser pour la classe dérivée

Héritage => sous-typage

- Partout où on attend un Pixel, on peut fournir un ColoredPixel
- Quel sens (sémantique) doivent avoir les méthodes ?

```
public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel(1,2);
    cp.setRed((byte) 100);

    Pixel p = cp; // héritage => sous-typage

    System.out.println(p);  // ?

    System.out.println(p.equals(new ColoredPixel(1,2))); // ?
}
```

Redéfinition de méthode

- Fournir une nouvelle définition de la même méthode:
- Même nom, mêmes arguments, code différent
- L'annotation **`@Override`** demande au compilateur de vérifier qu'on redéfinit bien quelque chose

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  // ...
  @Override
  public String toString() {
    return super.toString()+"["+rgb[0]+":"+rgb[1]+":"+rgb[2]+"]";
  }
  public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel(2,2);
    System.out.println(cp);  // (2,2)[0:0:0]
    Pixel p = new Pixel(5,5);
    System.out.println(p);   // (5,5)
    Object o =  new ColoredPixel(2,2);
    System.out.println(o);   // (2,2)[0:0:0]
  }
}
```

L'héritage c'est ...

3 choses (indissociables)

- On **veut** récupérer **l'ensemble des membres** (champs, méthodes) de la super-classe (même privés)
- On **doit redéfinir toutes les méthodes** qui n'ont pas la bonne sémantique dans la sous-classe
- La sous-classe **est sous-type de** la super-classe

si on ne veut pas une de ces 3 choses alors il ne faut pas faire d'héritage

Héritage et Object

- En Java, toutes les classes héritent de Object
  - soit directement

    le compilateur ajoute **`extends java.lang.Object`**

  - soit indirectement

    par ex, ColoredPixel hérite de Pixel qui hérite de Object

  => toutes les classes sont sous-type de Object
- Il faut redéfinir equals / hashCode / toString si c'est nécessaire !

Redéfinition (méthodes) *versus* masquage (champs)

- Les **champs** définis dans les classes dérivées sont tous présents dans l'objet instance de la classe dérivée
  - Même s'ils ont même nom et même type
  - On peut accéder à celui immédiatement supérieur par super.x
  - La **résolution** dépend du **type déclaré** du paramètre
  - Ça permet d'accéder à chacun d'entre eux par transtypage
- Pour la **méthode**, une seule est conservée !
  - On peut accéder à celle immédiatement supérieure par super.m()
  - La **résolution** est faite en deux temps
    - **Compile-time**: on vérifie que c'est possible sur le type déclaré
    - **Runtime**: on cherche la plus précise étant donnée le type « réel »
  - Les autres ne sont plus accessibles

Redéfinition versus surcharge

- Si la signature de la méthode qu'on définit dans la classe dérivée n'est pas la même que celle de la classe de base, il s'agit de **surcharge**:
  - Dans ce cas les deux méthodes cohabitent dans la classe dérivée

```
class A {
  void m1() { ... }
  void m2() { ... }
  Pixel m3() { ... }
  void m4(Pixel p) { ... }
}
class B extends A {
  @Override void m1() { ... }                  // redefinition
           void m2(int a) { ... }              // surcharge
  @Override ColoredPixel m3() { ... }          // redefinition
  @Override void m4(Pixel p) { ... }           // redefinition
           void m4(ColoredPixel p) { ... } // surcharge
}
```

Les principes de la redéfinition

- Quand on redéfinit une méthode m() dans B alors qu'elle était définie dans A, où B est un sous-type de A
  - L'objectif est de lui donner une définition plus précise *(mieux adaptée à B qu'à A)*, de sorte qu'elle soit appelée à run-time, y compris si à compile-time le compilateur n'avait vu que celle qui est définie dans A

Les principes de la redéfinition

- Le compilateur est sensé éviter les mauvaises surprises (i.e. découvrir un problème à run-time!) c'est ce qui gouverne les règles
  - Une méthode d'instance ne peut pas redéfinir une méthode static
  - L'accessibilité de la méthode redéfinie ne peut pas être plus restrictive
  - Le type de retour ne peut pas être d'un super-type (références)
  - Les exceptions propagées ne peuvent être que d'un sous-type

La méthode equals()

- De la même manière qu'il existe une méthode toString() dans la classe Object, que toute sous-classe peut redéfinir
- Il existe dans Object une méthode equals(Object obj) dont le « contrat » est clairement établi par la documentation
  - Par défaut, elle teste l'**égalité primitive des références**
  - Il faut la redéfinir

```
public class Pixel {
  private int x, y;
  // ...
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Pixel))
      return false;
    Pixel p = (Pixel) obj;
    return (x==p.x) && (y==p.y);
  }
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof ColoredPixel))
      return false;
    ColoredPixel cp = (ColoredPixel) obj;
    return super.equals(obj) &&
         rgb[0]==cp.rgb[0] &&
         rgb[1]==cp.rgb[1] &&
         rgb[2]==cp.rgb[2];
  }
}
```

Le contrat de la méthode equals()

- Définit une relation d'équivalence sur les références non-nulles
  - **Reflexive**
    - Pour toute référence x non nulle, x.equals(x) vaut true
  - **Symétrique**
    - Pour toutes références x et y non nulles, x.equals(y) ssi y.equals(x)
  - **Transitive**
    - Pour toutes références x, yet z non nulles,
      si x.equals(y) et y.equals(z) alors x.equals(z)
  - **Cohérente**
    - Tant qu'on ne modifie pas les valeurs utilisées pour tester l'égalité, la valeur de x.equals(y) retourne toujours la même valeur
  - Pour toute référence x non nulle, x.equals(null) vaut false
- Des objets égaux au sens de equals doivent avoir le même **hashcode**
  - Redéfinition de equals() implique en général redéfinition de hashCode()

La symétrie peut se discuter...

- Demandez à un Pixel en (2,2) s'il est égal à un ColoredPixel en (2,2), il dira que oui!
  - Il teste uniquement les coordonnées...
- Demandez à un **ColoredPixel magenta** en (2,2) s'il est égal à un Pixel en (2,2), il dira que non!
  - Il est sensé tester la couleur que le Pixel n'a même pas...
- On peut trouver que ce code est acceptable... ou pas

```
public class ColoredPixel extends Pixel {
  // ...
  public static void main(String[] args) {
    Object o1 = new Pixel(2,2);
    Object o2 = new ColoredPixel(2,2);
    System.out.println(o1.equals(o2)); // true
    System.out.println(o2.equals(o1)); // false
  }
}
```

Pour être plus strict...

- Il faut considérer que deux objets qui ne sont pas de la même classe ne peuvent pas être égaux
  - instanceof ne suffit plus
  - Il faut connaître la classe « exacte » de l'objet (à runtime)
  - Méthode Class getClass() de la classe Object

Dans Pixel :

```
@Override
public boolean equals(Object obj) {
  if(obj == null) return false;
  if(obj.getClass() != getClass())
    return false;
   Pixel p = (Pixel) obj;
   return (x==p.x) && (y==p.y);
}
```

```
@Override
public boolean equals(Object obj) {
  if(obj == null) return false;
  if(obj.getClass() != getClass())
    return false;
  ColoredPixel cp = (ColoredPixel) obj;
  return super.equals(obj) &&
    Arrays.equals(this.rgb, cp.rgb);
}
```

Dans ColoredPixel :

```
public static void main(String[] args) {
  Object o1 = new Pixel(2,2);
  Object o2 = new ColoredPixel(2,2);
  System.out.println(o1.equals(o2)); // false
  System.out.println(o2.equals(o1)); // false
}
```

Attention : deux objets de deux classes différentes héritant de Pixel ne pourront plus jamais être égaux… sauf à redéfinir complètement equals sans passer par super.equals

La méthode hashCode()

- Cette méthode est utlisée lorsqu'on stocke des objets dans une table de hachage (exemple java.util.HashMap)
- Elle établit également un « **contrat** » (de pair avec equals())
  - `public int hashCode()`
  - Étant donnée une exécution de la JVM, différents appels à la méthode hashCode() doivent retourner la même valeur tant qu'on ne modifie pas les valeurs utilisées pour tester l'égalité (equals())
  - Si deux objets sont égaux au sens de equals(), la méthode hashCode() appelée sur les deux doit produire la même valeur
  - Deux objets distincts au sens de equals() peuvent avoir des hashCode() identiques (c'est une « collision »), mais fournir des hashCode() distincts pour des objets distincts au sens de equals() améliore la performance des tables de hachage.

Utilisation de hashCode() et equals()

- Les ensembles, les tables de hachage, etc.
- Si equals est redéfinie, mais pas hashCode, voilà ce qui arrive

```java
import java.util.HashSet;

public class Pixel {
  // ...
  public static void main(String[] args) {
    Pixel zero = new Pixel(0,0);
    Pixel def = new Pixel();
    HashSet<Pixel> set = new HashSet<>();
    set.add(zero);
    System.out.println(zero.equals(def));  // true
    System.out.println(set.contains(def)); // false
    System.out.println(zero.hashCode());   // 1808253012
    System.out.println(def.hashCode());    //  589431969
  }

}
```

Incohérence entre equals() et hashCode()

Exemple de hashCode() pour nos pixels

```java
public class Pixel {
  // ...
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Pixel))
      return false;
    Pixel p = (Pixel) obj;
    return (x==p.x) && (y==p.y);
  }
  @Override
  public int hashCode() {
    return Integer.rotateLeft(x,16) ^ y;
  }
```

```java
public static void main(String[] a){
  Pixel zero = new Pixel(0,0);
  Pixel def = new Pixel();
  HashSet set = new HashSet();
  set.add(zero);
  set.contains(def); // true
  zero.hashCode();   // 0
  def.hashCode();    // 0
  zero.equals(def);  // true
}
```

```java
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  // ...
  @Override
  public int hashCode() {
    // return super.hashCode() ^ Integer.rotateLeft(rgb[0],16)
    //        ^ Integer.rotateLeft(rgb[1],8) ^ rgb[0];
    return super.hashCode() ^ Arrays.hashCode(rgb);
  }
}
```

Les classes et méthodes « final »

- Le mot-clé **final** existe pour les méthodes:
  - Il signifie que la méthode ne pourra pas être redéfinie dans une sous-classe
  - Peut être utile pour garantir qu'aucune autre définition ne pourra être donnée pour cette méthode (sécurité)
- Le mot-clé **final** existe pour les classes:
  - Il devient alors impossible d'hériter de cette classe
  - Les méthodes se comportent comme si elles étaient final

Les interfaces

- Une **classe** définit:
  - Un type
  - Une structure de données pour les instances (les champs)
  - Des méthodes avec leur code (leur définition)
- Une **interface** définit:
  - Un type
  - Des méthodes sans leur code (méthodes abstraites) – sauf depuis Java 8 : `default`
- => pas de champ, pas d'objet, pas d'état

Les interfaces

- Une interface **ne peut pas être instanciée**
- Elle est destinée à être « **implémentée** » par des classes
  - À qui elle donnera son type
  - Qui fourniront des définitions pour les méthodes déclarées (code)
- L'idée, c'est celle d’une « promesse » :
  - quand on a une variable déclarée du type de l'interface, et qu'on appelle une méthode dessus
  - on est sûr (compilo-garanti) que la référence contenue accède à une instance d'une classe qui « sait » implémenter la méthode

Intérêt des interfaces

- Donner un type commun à des classes différentes pour en faire un même usage
  - Ex: Manipuler des tableaux de « trucs » qui ont chacun une surface
  - Faire la somme des surfaces des trucs qui sont dans le tableau

```
public interface Surfaceable {
  public double surface();
}
```

```
public class AlgoOnTrucs {
  public static double totalSurface(Surfaceable[] array) {
    double total = 0.0;
    for(Surfaceable truc : array)
      total += truc.surface();
    return total;
  }
}
```

Utilisation d'interface

- 2 principaux avantages:
  - L'algorithme de la méthode totalSurface(Surfaceable[] array) fonctionne indépendamment de la classe réelle des objets qui sont stockés dans array:
    c'est le **sous-typage**
  - La méthode surface() effectivement appelée sur les objets contenus dans le tableau sera la plus précise possible, en fonction du type réel de chaque objet:
    c'est le **polymorphisme**

Utilisation d'interface

```
public class AlgoOnTrucs {

  public static double totalSurface(Surfaceable[] array) {
    ...
  }

  public static void main(String[] args) {
    Rectangle rectangle = new Rectangle(2,5);
    Square square = new Square(10);
    Circle circle = new Circle(1);
    Surfaceable[] t = {rectangle, square, circle};
    System.out.println(totalSurface(t));
                          // 113.1415926535898
  }
}
```

Implémentation d'interface

```java
public class Square implements Surfaceable {
  private final double side;
  public Square(double side) {
    this.side = side;
  }
  @Override
  public double surface() {
    return side * side;
  }
}
```

```java
public class Rectangle implements Surfaceable {
  private final double height;
  private final double width;
  public Rectangle(double height, double width) {
    this.height = height;
    this.width = width;
  }
  @Override
  public double surface() {
    return height * width;
  }
}
```

```java
public class Circle implements Surfaceable {
  private final double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  @Override
  public double surface() {
    return Math.PI * radius * radius;
  }
}
```

Les membres des interfaces

- Déclarations de méthode publiques
  - Toutes les méthodes sont **`abstract public`**
    - même si non spécifié, sauf `default` (voir + loin)

```
public interface Surfaceable {
  double surface(); // equivaut à
  public abstract double surface();
}
```

Les membres des interfaces

- Champs publiques constants
  - Tous les champs sont **`public final static`**
    - Le compilateur ajoute les mot-clés

```
public interface I {
   int field = 10; // equivaut à
   public final static int field = 10;
}
```

Les membres des interfaces

- Il n'est **pas possible d'instancier** une interface, autrement dit de créer un objet
  - On ne peut que déclarer des variables avec leur type
  - Ces variables pourront recevoir des références à des objets qui sont des instances d'une classe qui implémente l'interface

Implémentation d'interface et sous-typage

- Une classe peut **implémenter** une interface
  - Mot clé **`implements`**

```
public class Rectangle implements Surfaceable {
   ...
}
```

  - La classe `Rectangle` définit un **sous-type** de `Surfaceable`

```
Surfaceable s = null;

s = new Rectangle(2,5);
```

Implémentation d'interface et sous-typage

- Une interface ne peut pas implémenter une autre interface
  - On ne saurait pas comment implémenter les méthodes
- Mais une interface peut **hériter** d'une interface
  - Mot clé **`extends`**

```
public interface Paintable extends Surfaceable {
  double paint(byte[] color, int layers);
}
```

  - `Paintable` est un sous-type de `Surfaceable`

```
Surfaceable[] array = new Surfaceable[3]; // tableaux!
Paintable p = null;
array[0] = p; // OK: Paintable < Surfaceable
p = array[1]; // Cannot convert from Surfaceable to Paintable
```

Sous-typage entre interfaces

- Une interface peut **hériter** de **plusieurs** autres interfaces
  - Séparer les super-types avec des virgules

```
public interface SurfaceableAndMoveable
                 extends Surfaceable, Moveable {
   ...
}
```

  - Le type SurfaceableAndMoveable définit un **sous-type** des deux types **Surfaceable** et de **Moveable** (**sous-typage multiple**)
    - SurfaceableAndMoveable < Surfaceable et SurfaceableAndMoveable < Moveable
    - Ces deux dernières n'ont aucune relation entre-elles

Héritage de classe et implémentation d'interfaces

- Une classe peut à la fois
- **hériter** d'une classe
- et **implémenter** plusieurs interfaces
  - Sous-typage multiple

```
public class SolidCircle extends Circle implements Paintable, Moveable {
  private final Point center;
  public SolidCircle(Point center, double radius) {
    super(radius);
    this.center = center;
  }
  @Override  // Pour pouvoir implémenter Paintable
  public double paint(byte[] color, int layers) {
    // doThePaintingJob(color,layers);
    return layers * surface(); // SolidCircle < Circle < Surfaceable
  }
  @Override  // Pour pouvoir implémenter Moveable
  public void moveTo(int x, int y) {
    center.moveTo(x,y);
  }
  public static void main(String[] args) {
    SolidCircle sc = new SolidCircle(new Point(0,0), 3);
    Circle c = sc;      double d = c.surface();      // SolidCircle < Circle
    Paintable p = sc;   p.paint(new byte[]{0,0,0},2);// SolidCircle < Paintable
    Moveable m = sc;    m.moveTo(1, 1);              // SolidCircle < Moveable
  }
}
```

Vérifications du compilateur

- **Toutes les méthodes déclarées** (abstract) dans l'ensemble des interfaces dont on revendique l'implémentation **doivent être implantées**
  - Définies avec leur code
- Le modificateur de visibilité **ne peut pas être autre chose que `public`**
  - Même si on a mis la visibilité par défaut dans l'interface, le compilateur y ajoute **`public abstract`**

Vérifications du compilateur

- Que se passe t il si plusieurs méthodes de même nom et même signature de différentes interfaces doivent être implémentées dans la même classe?
  - Ce sont des « promesses » (fonctionnalités), pas des implémentations…
    - Il est préférable qu'elles soient cohérentes !

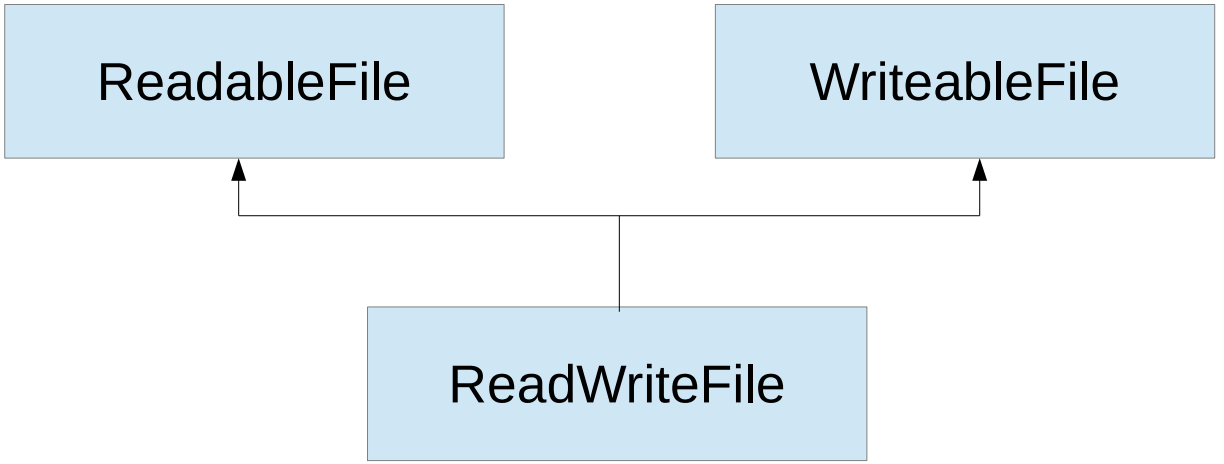
Héritage d'interface : autre exemple

Une interface qui hérite d'autres interfaces, promet l'union des méthodes

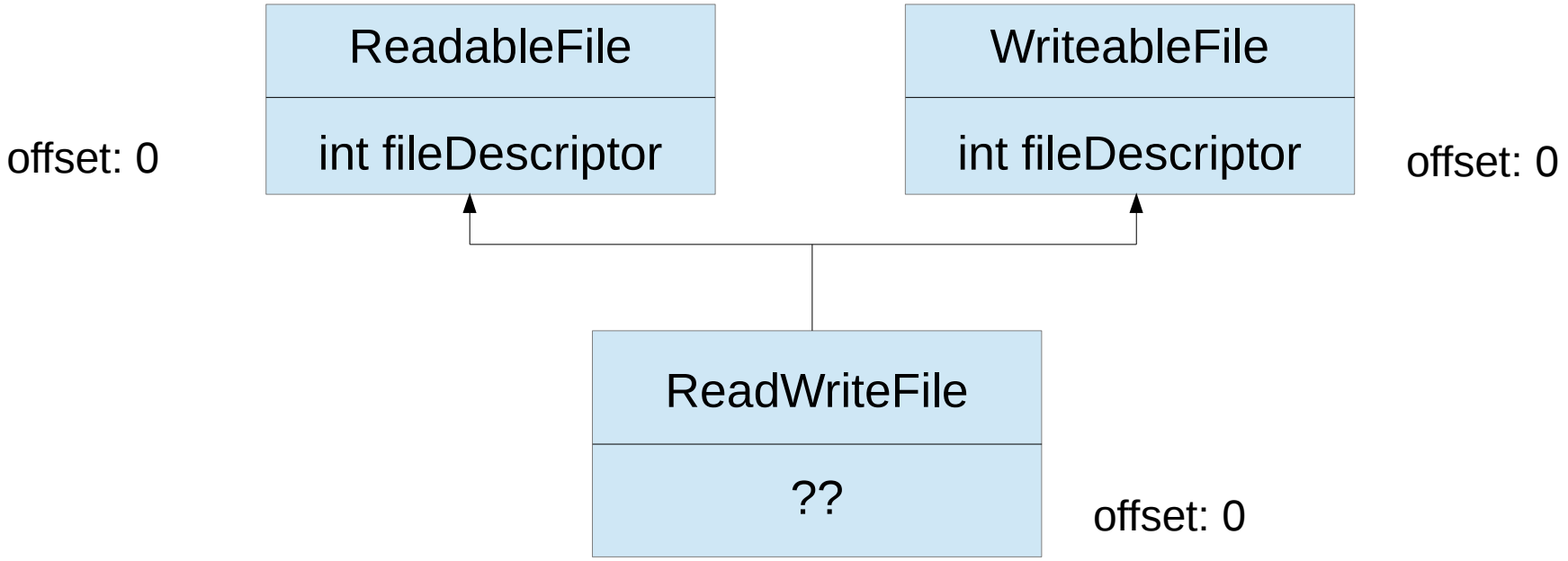
```
public interface ReadableIO {
  int length();
  int read(Buffer buffer);
}
public interface WritableIO {
  int length();
  int write(Buffer buffer);
}
public interface IO extends ReadableIO, WritableIO {
  // 3 méthodes, read, write et length
}
```

Héritage simple

- En Java (ou C#), contrairement à C++, il n'est possible d'hériter que d'une seule classe
  - il n'y a pas d'héritage multiple de classe
- Et comment on fait, si on veut des fichiers dans lesquels on peut lire et écrire ?



Pourquoi pas d'héritage multiple ?



Les champs sont accédés par un index,
si il y a de l'héritage multiple, il va y avoir un conflit d'index !

En C++, lors du sous-typage, on décale l'adresse de base, beurk !

Problème de l'héritage multiple

Le problème est qu'il n'est pas possible

- d'avoir de l'héritage multiple
- d'avoir un seul header pour un objet
  (ce qui est important pour le GC)

Il n'y a pas de problème si il n'y a pas de champ !

Solution

les interfaces et le sous-typage multiple

En résumé : interface

Une interface est un ensemble de méthodes abstraites (ou non depuis Java 8) sans champs

Une interface est un type abstrait qui permet de manipuler plusieurs classes avec un code commun

Interface et contrat

- Une interface spécifie un **contrat** que les classes qui implantent l'interface doivent respecter
  - la classe doit implanter toutes les méthodes abstraites
- Une interface permet le **sous-typage** et le **polymorphisme** sans l'héritage des champs et des méthodes
  - peut être vu comme une forme simplifiée d'héritage

Interface & instanciation

Une interface ne peut pas être instanciée
Surfaceable s = new Surfaceable();
ne compile pas

Une interface représente une classe,
c'est un **type abstrait**
Surfaceable s = new Rectangle(...);
Surfaceable s = new Circle(...);

Interface & classe

Une classe qui implante une interface doit donc implanter toutes les méthodes de l'interface

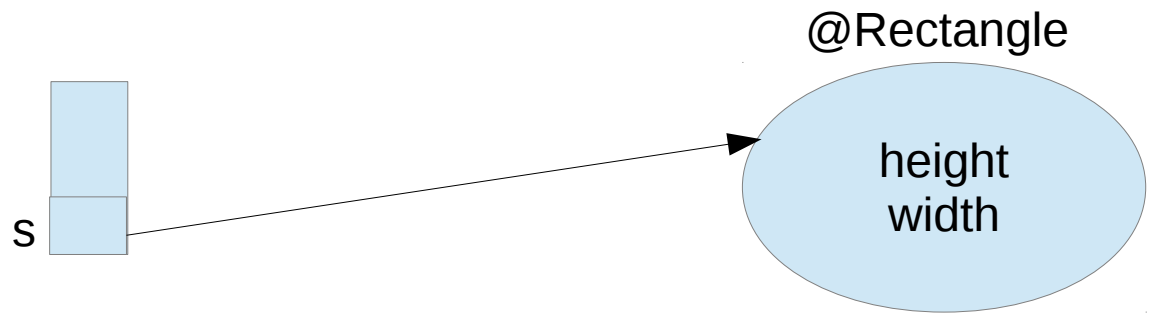
sinon on pourrait appeler une méthode qui n'a pas de code à travers l'interface !

```
public class StupidSurfaceable
                    implements Surfaceable {
  // ne compile pas, il manque la méthode surface()
}
```

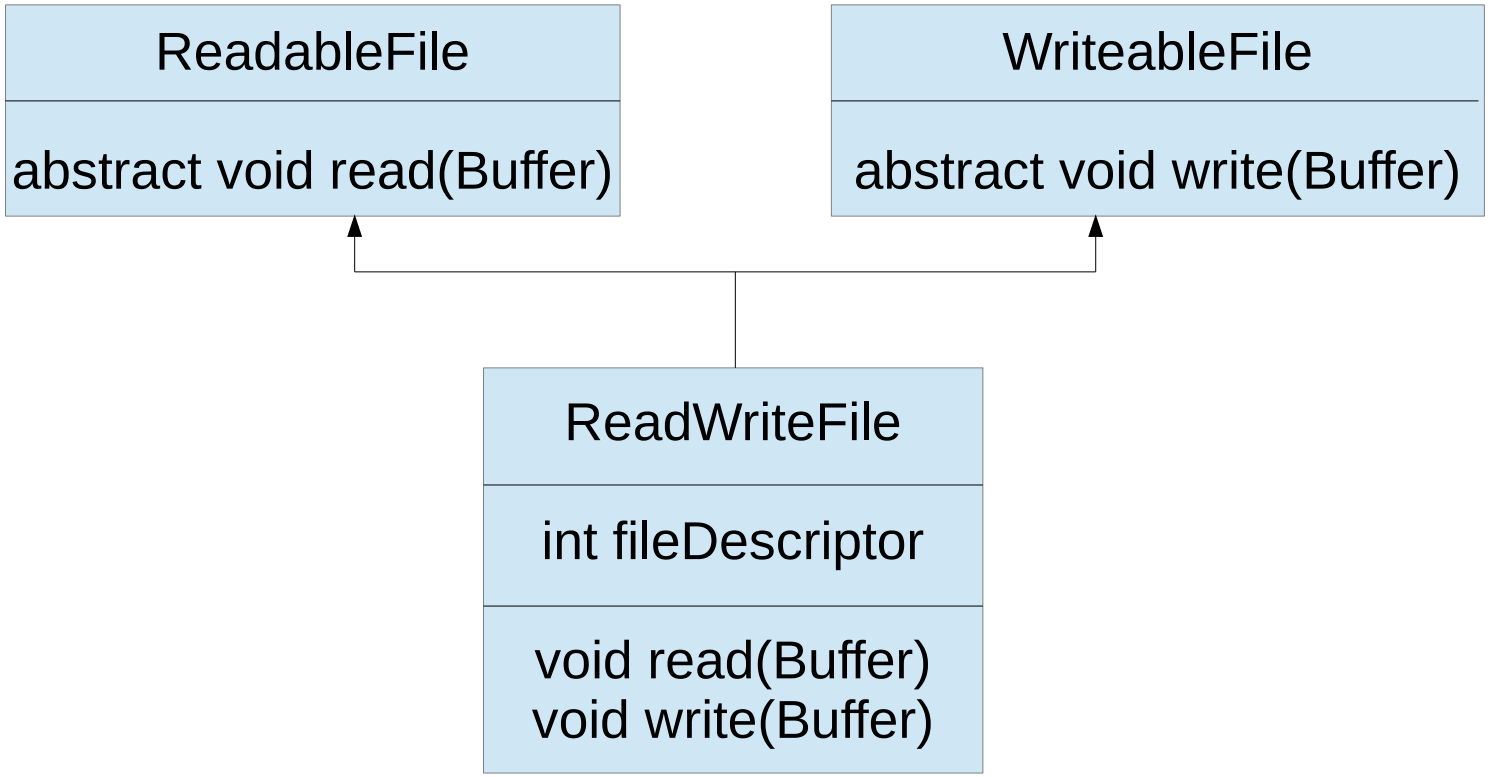
Interface & type abstrait

Une interface sert à manipuler des classes différentes en utilisant le même type

A l'exécution, une référence typée avec l'interface est forcément une référence à une classe qui implante l'interface
Surfaceable s = ...



Sous-typage multiple



Une classe peut alors implanter plusieurs interfaces

Méthode par défaut

Une méthode par défaut est une méthode non-abstraite dont le code sera utilisé si aucun code n'est fournit

```
public interface Bag {
  public abstract int size();
  public default boolean isEmpty() {
    return size() == 0;
  }
}
```

Méthode par défaut

La méthode par défaut est utilisée si aucune implantation n'est fournie

```
public class HashBag implements Bag {
  private int size;
  ...
  public int size() {
    return size;
  }
  // isEmpty de Bag est utilisé
}
```

Méthode par défaut et toString, equals et hashCode

Comme java.lang.Object fournit toujours les méthodes toString, equals et hashCode,
il est inutile d'écrire une méthode par défaut toString, equals ou hashCode dans une interface

La version de java.lang.Object sera toujours préférée à celle de l'interface

Méthode par défaut et conflit

Si deux méthodes par défaut sont disponibles, celle du sous-type est choisie si il existe un sous-type, sinon le compilateur grogne

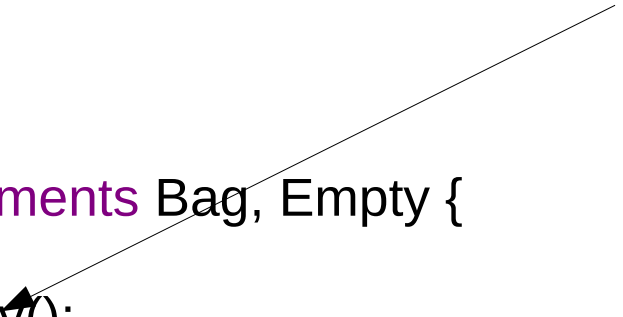
```
public interface Empty {
  public default boolean isEmpty() {
    return true;
  }
}
public class EmptyBag implements Bag, Empty {
  // problème, 2 méthodes isEmpty() par défaut
}
```

Résoudre le conflit

Il faut fournir une implantation pour résoudre le conflit

```
public interface Empty {
  public default boolean isEmpty() {
    return true;
  }
}
public interface Bag {
  ...
  public default boolean isEmpty() {
    ...
  }
}
public class EmptyBag implements Bag, Empty {
  public boolean isEmpty() {
    return Empty.super.isEmpty();
  }
}
```

SuperInterface.super permet d'accéder à l'implantation par défaut dans une interface

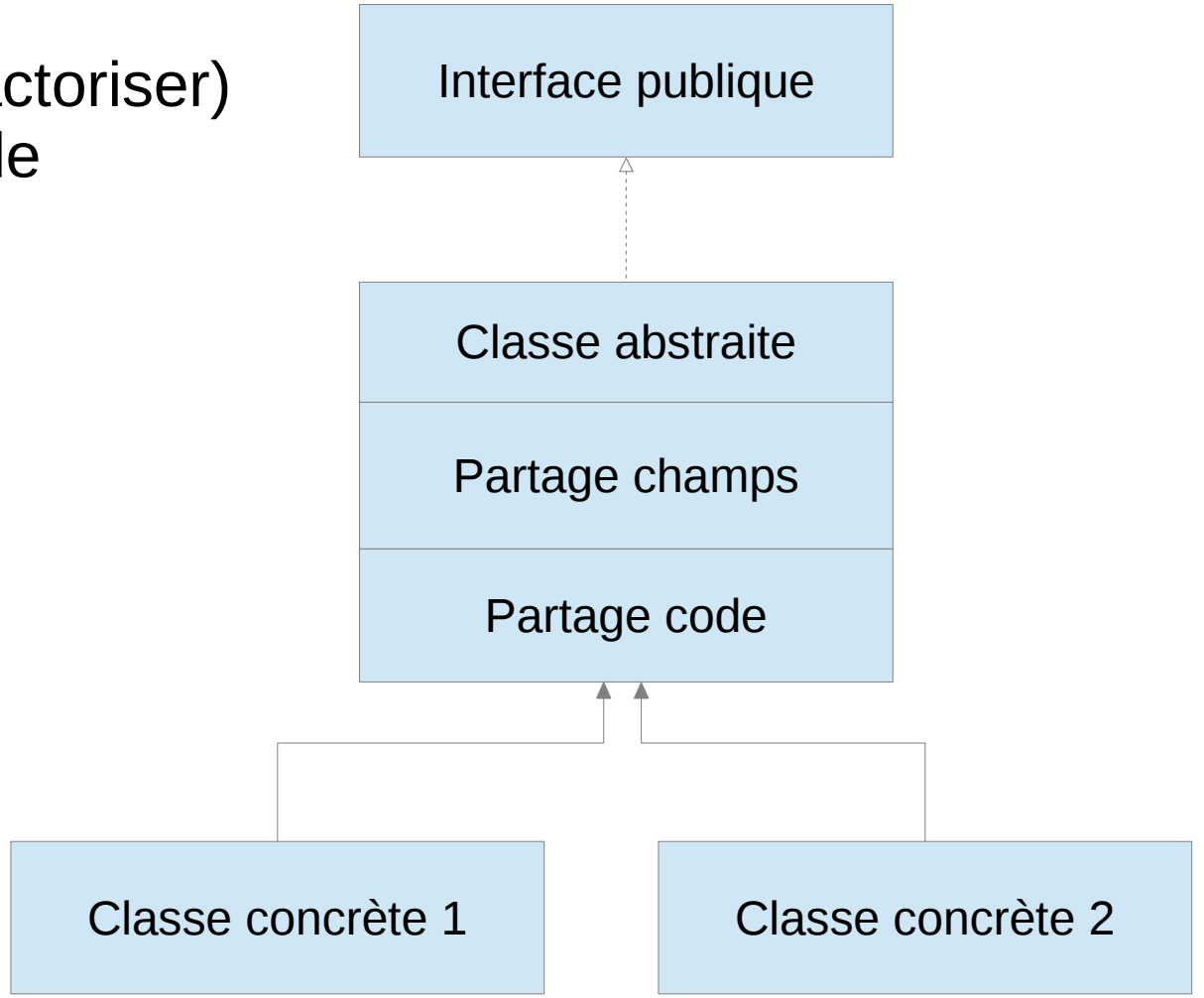


Design: interface ou héritage

- On **hérite d'une classe**
  - pour créer un nouveau type qui est « **une sorte particulière** » de classe de base
- On définit une interface et on l'implémente
  - Pour une fonctionnalité transverse
    - Comparable, Closeable, *Mesurable, Déplacable...*
  - Pour regrouper un ensemble de fonctionnalités qui pourront être implémentées par des instances qui en implantent déjà d'autres (ou qui héritent d'une autre classe)

Classe abstraite

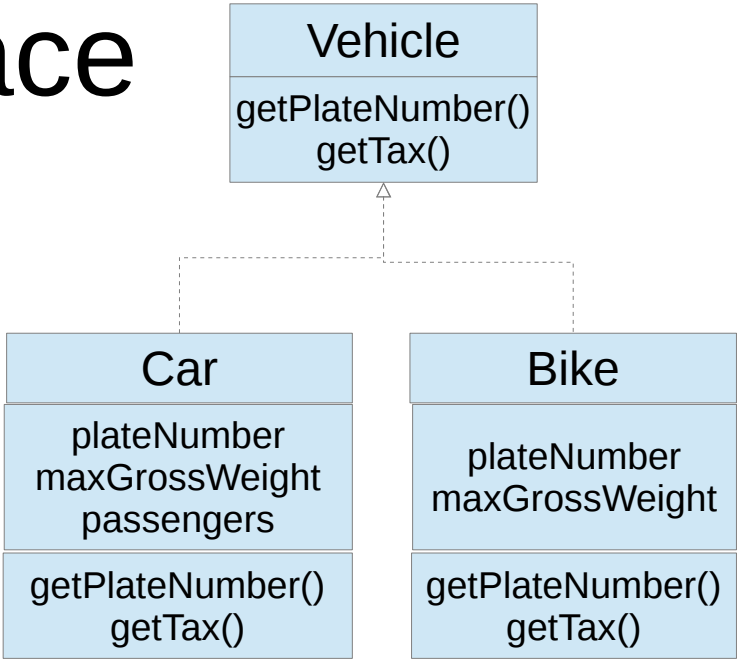
- Il est possible de définir une classe dont certaines méthodes sont abstraites
  - Utile pour partager (factoriser) des champs et du code
- Schéma classique :



```java
public class Car {
  private final String plateNumber;
  private final int maxGrossWeight;
  private final int passengers;
  public Car(String plateNumber, int maxGrossWeight, int passengers) {
    this.plateNumber = plateNumber;
    this.maxGrossWeight = maxGrossWeight;
    this.passengers = passengers;
  }
  public String getPlateNumber() {
    return plateNumber;
  }
  public int getTax() {
    return passengers * maxGrossWeight / 10;
 }
}
```

```java
public class Bike {
  private final String plateNumber;
  private final int maxGrossWeight;
  public Bike(String plateNumber, int maxGrossWeight) {
    this.plateNumber = plateNumber;
    this.maxGrossWeight = maxGrossWeight;
  }
  public String getPlateNumber() {
    return plateNumber;
  }
  public int getTax() {
    return maxGrossWeight / 2;
  }
}
```

Type commun => interface




```java
public interface Vehicle {
  String getPlateNumber();
  int getTax();
}
```

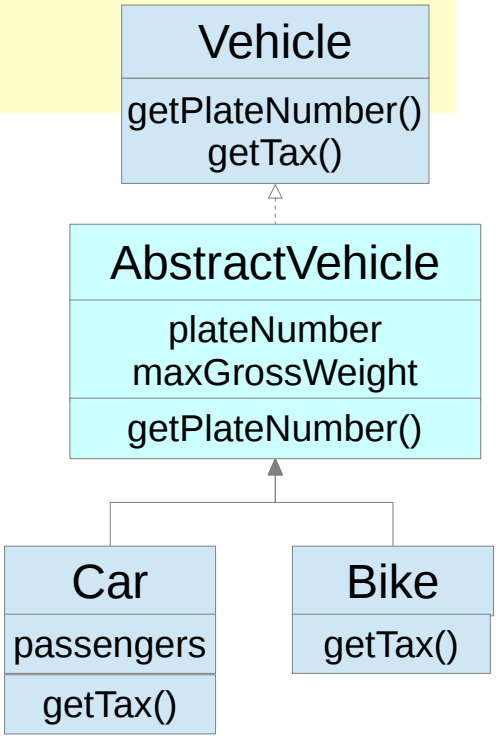
```java
public class Car implements Vehicle {
  ...
}
```

```java
public class Bike implements Vehicle {
  ...
}
```

```java
public static void main(String[] args) {
  Vehicle[] array = {
    new Car("AA-111-AA", 1850, 5),
    new Bike("BB-222-CC", 450),
    ... };
  for(Vehicle v : array) {
    System.out.println(v.getPlateNumber() + " must pay " + v.getTax());
  }
}
```

Factorisation de champs / code => classe abstraite

```java
public interface Vehicle {
  String getPlateNumber();
  int getTax();
}
```




```java
abstract class AbstractVehicle implements Vehicle {
  private final String plateNumber;
  private final int maxGrossWeight;
  public AbstractVehicle(String plateNumber, int maxGrossWeight) {
    this.plateNumber = plateNumber;
    this.maxGrossWeight = maxGrossWeight;
  }
  public String getPlateNumber() {
    return plateNumber;
  }
  protected int getMaxGrossWeight() {
    return maxGrossWeight;
  }
}
```

```java
public class Car extends AbstractVehicle {
  private final int passengers;
  public Car(String plateNumber,
             int maxGrossWeight,
             int passengers) {
    super(plateNumber, maxGrossWeight);
    this.passengers = passengers;
  }
  public int getTax() {
    return passengers * getMaxGrossWeight() / 10;
  }
}
```

```java
public class Bike extends AbstractVehicle {
  public Bike(String plateNumber,
              int maxGrossWeight) {
    super(plateNumber, maxGrossWeight);
  }
  public int getTax() {
    return getMaxGrossWeight() / 2;
  }
}
```

Classe abstraite et instantiation

Comme une interface, une classe abstraite ne peut pas être instanciée
AbstractVehicle v = new AbstractVehicle();
ne compile pas

Une classe abstraite peut être déclarée abstract sans méthode abstract

Si une des méthodes de la classe est abstract, alors la classe doit être déclarée abstract

Méthode abstraite et ...

Avoir une méthode abstract et static ne veut rien dire

- abstract: on doit la redéfinir
- static: pas de redéfinition possible

Avoir une méthode abstract et private ne veut rien dire

- abstract: on doit la redéfinir dans une sous-classe
- private: pas visible de l'extérieur

Sous-classe et protected

La visibilité protected veut dire accessible par les classes du même package ou accessible par les sous-classes

Cela permet de déclarer des méthodes que l'on veut accessibles par les sous-classes mais pas de l'extérieur

Il **ne faut pas** déclarer **un champs protected**, car dès qu'il existe une sous-classe, on ne peut plus changer le code de la super-classe

Il faut éviter que la classe abstraite soit publique

Rafinement de l'abstraction

De l'abstraction la plus pure à l'implantation

- Interface
  - Que des méthodes abstraites (public)
- Interface avec des méthodes par défaut
  - Méthodes abstraites et méthode concrètes (public)
- Classe abstraite
  - Champs + méthodes abstraites et concrètes
- Classe
  - Champs et méthodes

on peut définir des méthodes statiques partout