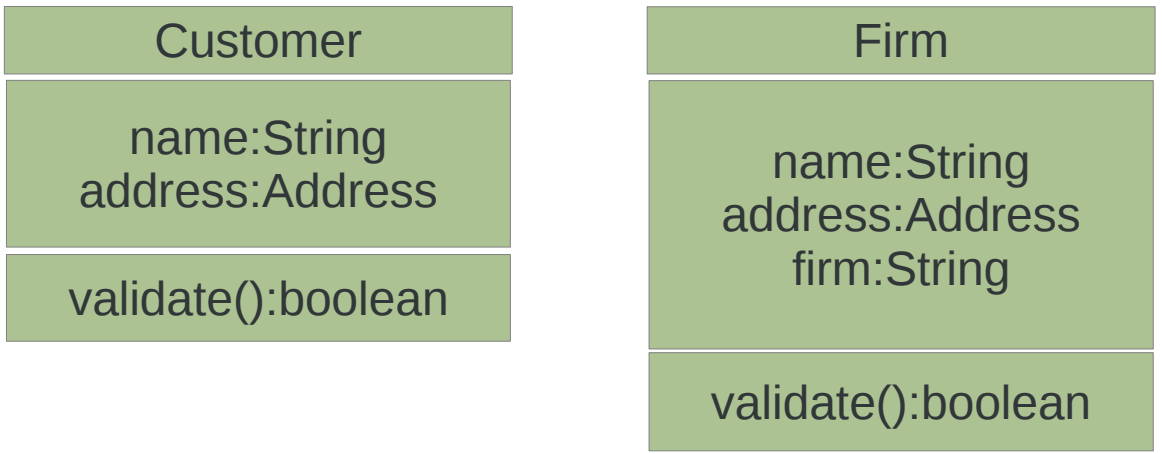
Héritage, Redéfinition & Type abstrait

L'héritage: un exemple

On souhaite modéliser le client d'un application de commerce en ligne

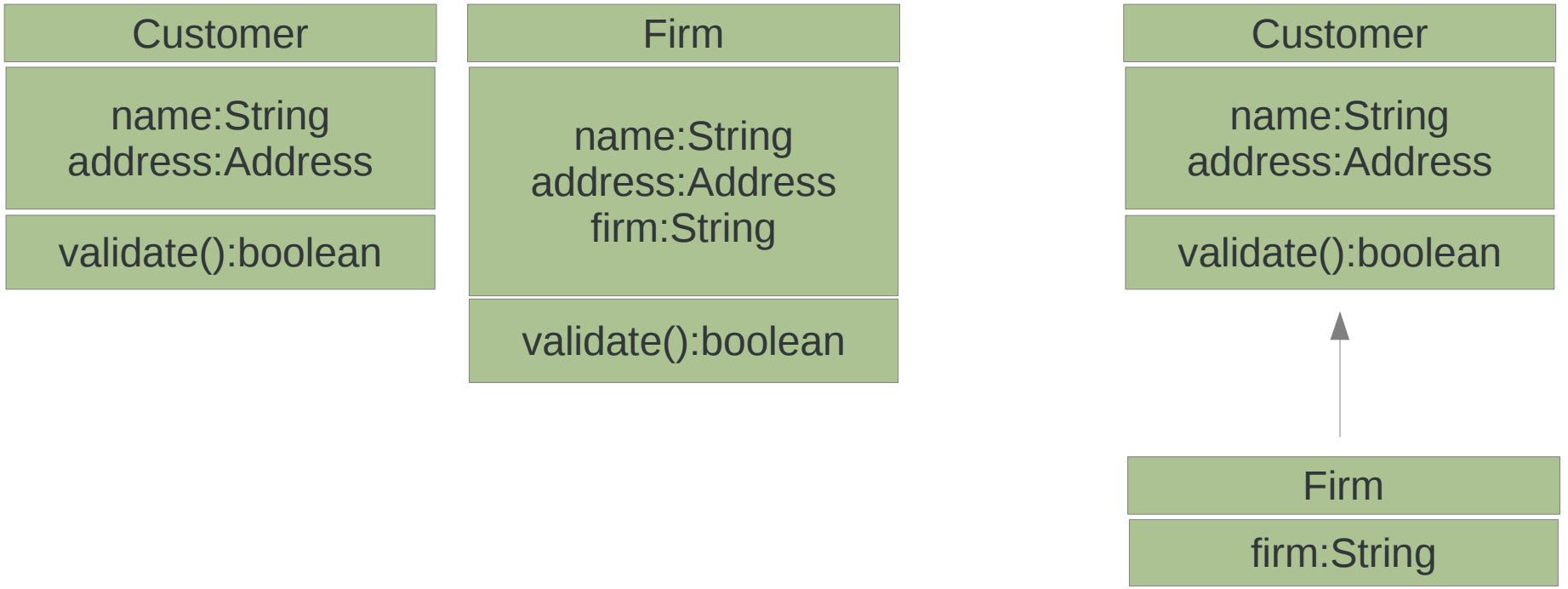
- L'application à 2 sortes de clients, des particuliers et des entreprises

On arrive donc au design suivant:



L'héritage: un exemple

On peu remarquer un objet de la classe Firm est comme un objet de la classe Customer mais avec un champ firm en plus

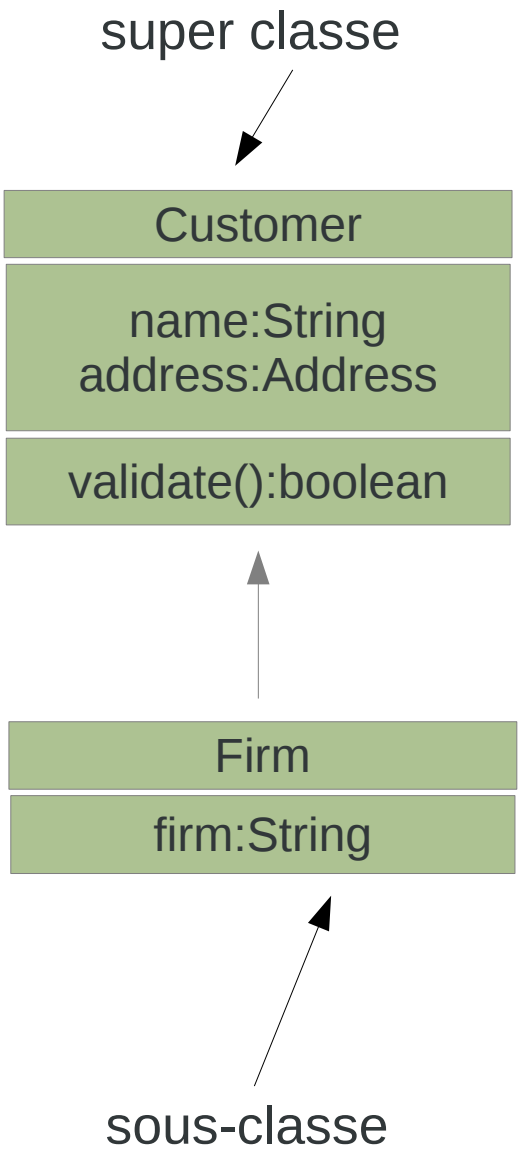


Vocabulaire

Une sous-classe hérite d'une super-classe

La sous-classe est la classe qui hérite

la super-classe est la classe dont on hérite



Dans le code ...

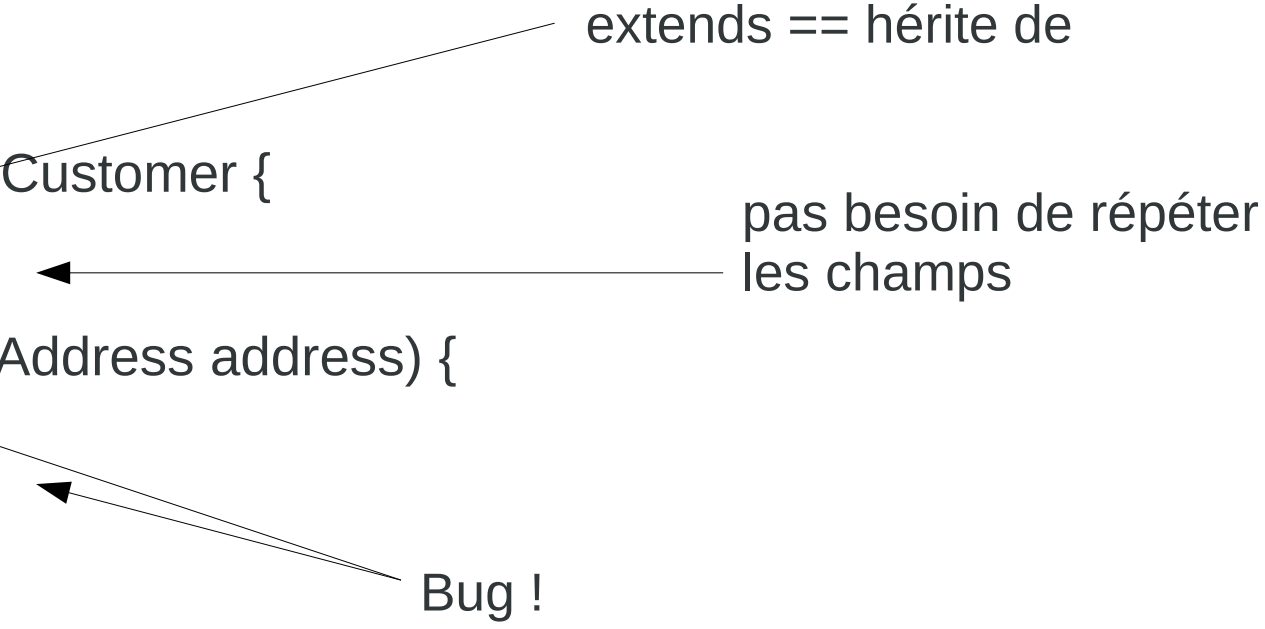
```
public class Customer {
  private final String name;
  private final Address address;

  public Customer(String name, Address address) {
    this.name = name;
    this.address = address;
  }
}
```

sans Héritage !

```
public class Firm  {
  private final String name;
  private final Address address;
  private final String firm;

  public Firm(String name, Address address) {
    this.name = name;
    this.address = address;
    this.firm = firm;
  }
}
```

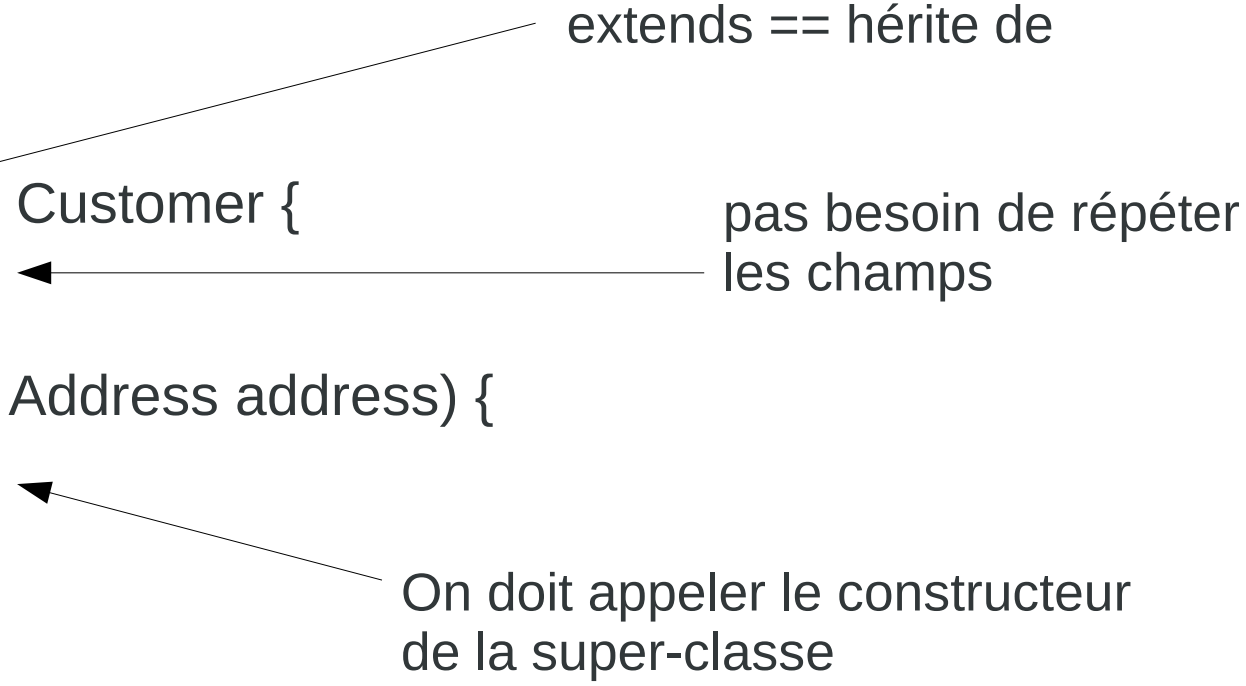
Dans le code ...

```
public class Customer {
  private final String name;
  private final Address address;

  public Customer(String name, Address address) {
    this.name = name;
    this.address = address;
  }
}

public class Firm extends Customer {
  private final String firm;

  public Firm(String name, Address address) {
    this.name = name;
    this.address = address;
    this.firm = firm;
  }
}
```



super() c'est super !

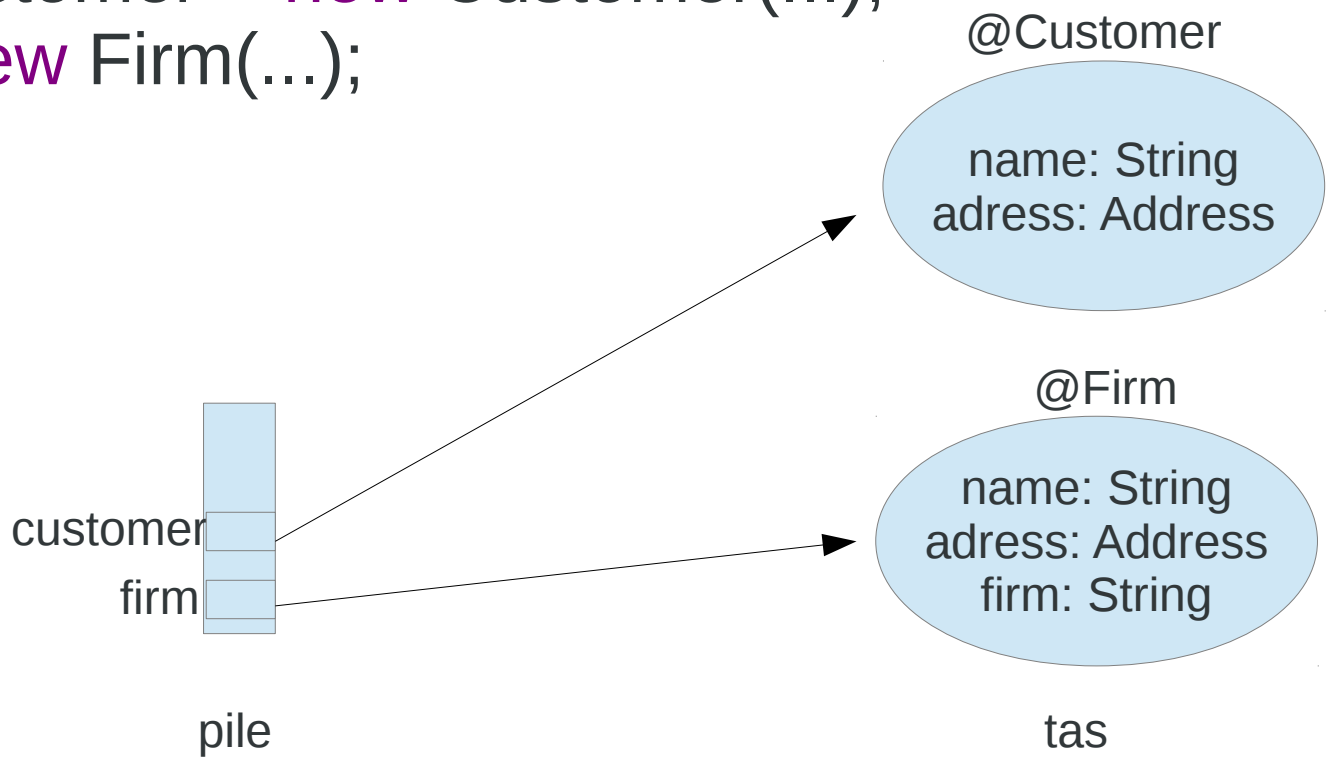
```
public class Customer {
  private final String name;
  private final Address address;

  public Customer(String name, Address address) {
    this.name = name;
    this.address = address;
  }
}

public class Firm extends Customer {
  private final String firm;

  public Firm(String name, Address address) {
    super(name, address);
    this.firm = firm;
  }
}
```



On hérite des membres

A l'édition de liens, la VM aggège les champs hérité des sous-classes

```
public static void main(String[] args) {
  Customer customer = new Customer(...);
  Firm firm = new Firm(...);
}
```



On hérite des méthodes et ...

```
public class Customer {
  private final String name;
  private final Address address;

  public Customer(String name, Address address) {
    this.name = name;
    this.address = address;
  }

  public void validate() {
     return AddressDB.exists(address) && !name.isEmpty();
  }
}

public class Firm extends Customer {
  private final String firm;

  public Firm(String name, Address address) {
    super(name, address);
    this.firm = firm;
  }
}
```

BUG !

On **doit** redéfinir les méthodes

Problème !

- On hérite de validate donc on peut écrire
  Firm firm = new Fim(...);
  boolean isValid = firm.validate();

  mais validate() est Customer::validate() qui ne valide pas le champs Firm::firm !

On **doit** redéfinir la méthode validate() dans Firm

Seconde tentative !

```
public class Customer {
  private final String name;
  private final Address address;
  ...
  public void validate() {
     return AddressDB.exists(address) && !name.isEmpty();
  }
}

public class Firm extends Customer {
  private final String firm;
  ...
  @Override
  public boolean validate() {
    return AddressDB.exists(address) &&
              !name.isEmpty() &&
              !firm.isEmpty();
  }
}
```

Compile pas !!

super. , c'est super !

```
public class Customer {
  private final String name;
  private final Address address;
  ...
  public void validate() {
     return AddressDB.exists(address) && !name.isEmpty();
  }
}

public class Firm extends Customer {
  private final String firm;
  ...
  @Override
  public boolean validate() {
    return super.validate() &&
              !firm.isEmpty();
  }
}
```

En terme objet,
Firm délégue la
responsabilité de tester
la validité des champs de
Customer à Customer::validate()

Héritage => sous-typage

Firm hérite de Customer donc Firm est un sous-type de Customer

```
public static void billing(Customer customer) {
  ...
}
public static void main(String[] args) {
  Customer customer = new Customer(...);
  billing(customer);
  Firm firm = new Firm(...);
  billing(firm);   // sous-typage permet la ré-utilisation de code
}
```

Partout où l'on attend un objet de type Customer, on peut donner un objet de type Firm à la place !

Donc l'héritage c'est ...

3 choses (indissociables)

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

si on ne veut pas un 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
  - si on ne met pas “extends”, le compilateur ajoute extends java.lang.Object
- soit indirectement
  - par ex, Firm hérite de Customer qui hérite de Object

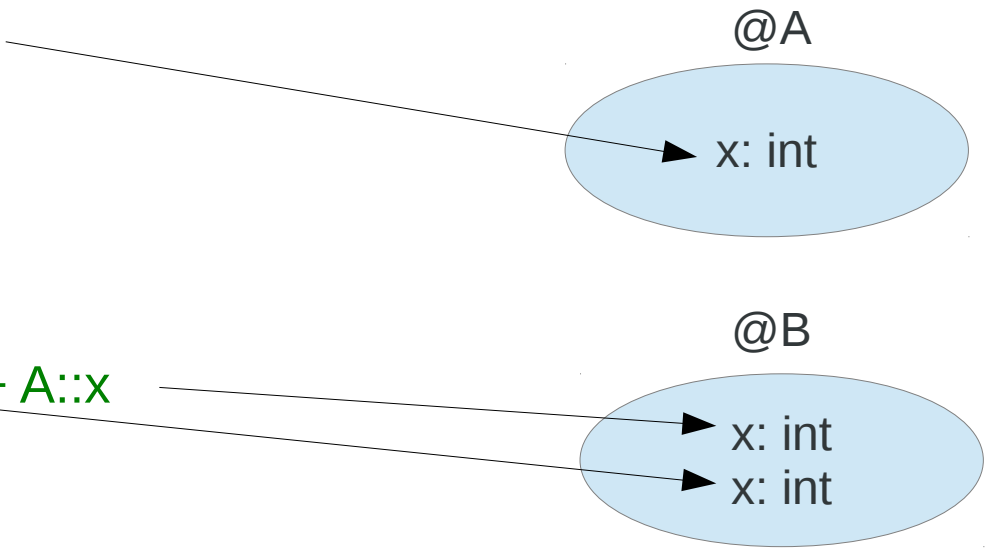
donc toutes les classes sont sous-type de Object

et il faut redéfinir equals/hashCode/toString si c'est nécessaire !

Champs de même nom ?

Il est possible d”avoir des champs de même nom dans des super-classes/sous-classes.

```
public class A {
  int x;
  public int getValue() {
    return this.x;                  // A::x
  }
}
public class B extends A {
  int x;
  public int getValue() {
    return this.x + super.x;        // B::x + A::x
  }
}
```




```
A a = new B();
a.x  // A::x
a.getValue()  // B::getValue
```

Les champs sont accèdés en fonction du type de la référence
On redéfinie pas les champs !

Constructeur vs Methode

Un constructeur est une méthode d'initialisation qui retourne toujours void

Mais contrairement aux méthodes, lors de l'héritage, **on hérite pas des constructeurs** !

Il faudra donc souvent écrire explicitement le constructeur

Héritage et constructeur

Le code ci-dessous ne marche pas !

```
public class Firm extends Customer {
  // pas de constructeur
}
```

le compilateur ajoute un constructeur public

```
  public Firm() {
    super();          // <- oups
  }
```

mais ce constructeur ne compile pas car Customer à un constructeur qui prend 2 paramètres !

Héritage et constructeur (suite)

Si l'on veut un constructeur sans paramètre, il faut écrire le constructeur à la main
par ex:

```
public class Firm extends Customer {
  public Firm() {
    super("<no name>", new Address());
  }
}
```

dans ce cas, le compilateur n'en ajoutera pas un qui ne compile pas :)

super 'dot'

Le mot clé **super.** :

- à la même valeur que **this** à l'exécution
- il est typé comme la super-classe (ici comme Customer)
- l'appel n'est pas virtuelle/polymorphe

```
public class Customer {
  ...
  public void validate() {
     return AddressDB.exists(address) && !name.isEmpty();
  }
}
public class Firm extends Customer {
  ...
  @Override
  public boolean validate() {
    return super.validate() && !firm.isEmpty();
  }
}
```

L'appel super.validate() permet donc d'appeler une méthode de la super-classe

Méthode finale

Il est possible d'utiliser le mot-clé final lors de la déclaration d'une méthode

Cela empèche de redéfinir cette méthode

Cela permet

- la sécurité, si je ne peux pas hériter,
  je ne peut pas redéfinir checkPassword par ex.
- de ne pas avoir un appel virtuel/polymorphe,
  car la méthode ne peut être redéfinie

Classe finale

On peut utiliser le mot clé “final” lors de la déclaration d'une classe

Il est alors impossible d'hériter de celle-ci

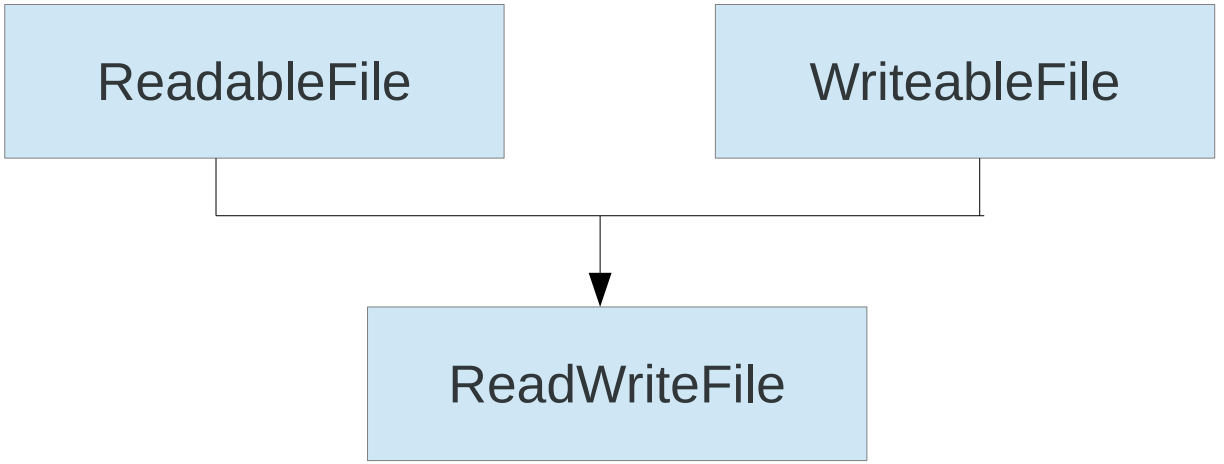
Cela permet

- la sécurité.
- d'être sûr de la mutabilité de la classe, pour cette raison java.lang.String, java.lang.Integer, etc sont finales

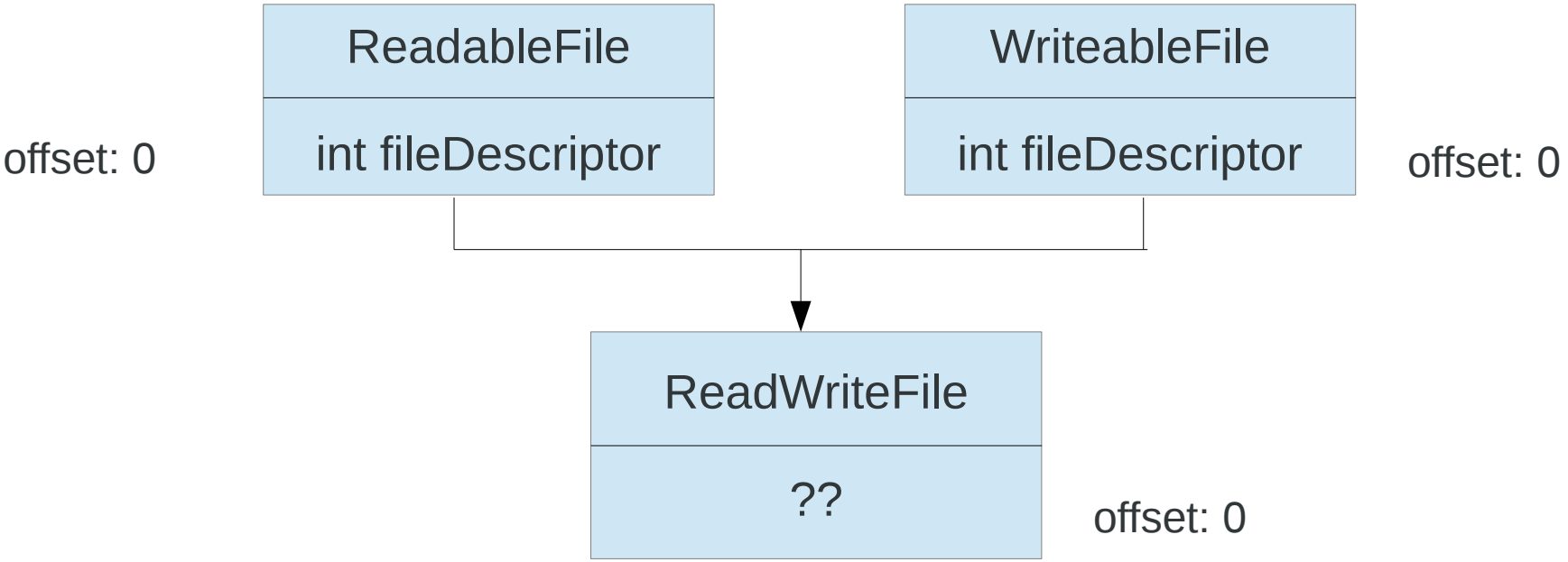
Héritage simple

En Java (ou C#), contrairement à C++, Il est possible d'hériter que d'une seule classe,
il n'y a pas d'héritage multiple

Et comment on fait, si on veut des fichiers que l'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

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

Exemple

On veut dessiner un tableau composée de rectangles et d'ellipses sur une surface graphique

```
public class DrawingArea {
  public void drawRect(int x,int y, int w, int h) { ... }
  public void drawEllipse(int x, int y, int w, int h) { ... }
}

public class Rectangle {
  private final int x, y, width, height;
  ...
}

public class Ellipse {
  private final int x, y, width, height;
  ...
}
```

Exemple (suite)

Première tentative

```
public static void drawAll(DrawingArea area, Object[] array) {
  for(Object o: array) {
    if (o instanceof Rectangle) {
      Rectangle r = (Rectangle)o;
      area.drawRect(r.x, r.y, r.width, r.height);
    } else
      if (o instanceof Rectangle) {
        Ellipse e = (Ellipse)o;
        area.drawRect(e.x, e.y, e.width, e.height);
      } else {
         throw new AssertionError();
      }
  }
}
```

Code avec un problème de localité !

Exemple (suite)

Deuxième tentative, on utilise l'héritage:

```
public class Rectangle {
  private final int x, y, width, height;
  public void draw(DrawingArea area) {
    area.drawRect(x, y, width, height);
  }
}

public class Ellipse extends Rectangle {
  @Override
  public void draw(DrawingArea area) {
    area.drawEllipse(x, y, width, height);
  }
}

public static void drawAll(DrawingArea area, Rectangle[] array) {
  for(Rectangle r: array) {
    r.draw(area);
  }
}
```

Astuce
On utilise le polymorphisme

Mais pourquoi Ellipse hérite de Rectangle ?

Astuce !
On utilise le sous-typage

Exemple (fin)

dernière tentative, on utilise une interface

```
public interface Shape {
  public void draw(DrawingArea area);
}
public class Rectangle implements Rectangle {
  private final int x, y, width, height;
  @Override
  public void draw(DrawingArea area) {
    area.drawRect(x, y, width, height);
  }
}

public class Ellipse implements Rectangle {
  @Override
  public void draw(DrawingArea area) {
    area.drawEllipse(x, y, width, height);
  }
}

public static void drawAll(DrawingArea area, Shape[] shapes) {
  for(Shape shape: shapes) {
    shape.draw(area);
  }
}
```

Astuce
On utilise le polymorphisme

Astuce !
On utilise le sous-typage

Interface et contrat

Une interface spécifie un contrat que les classes qui implante l'interface doivent respecter

- il faut que la classe implante toutes les méthodes abstraites

Une interface permet le sous-typage et le polymorphisme sans l'héritage des champs et des méthodes

- peut être vue comme une forme simplifié d'héritage

Interface & instanciation

Une interface ne peut pas être instancier
Shape shape = new Shape();
ne compile pas

Une interface représente une classe, c'est un type abstrait
Shape shape = new Rectangle(...);
Shape shape = new Ellipse(...);

Interface & classe

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

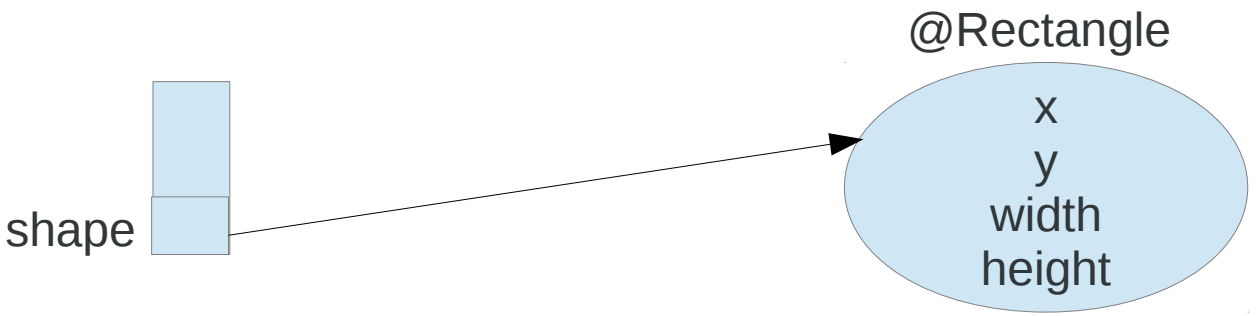
sinon on pourrait appeler une méthode qui n'a pas de code à travers l'interface !

```
public class StupidShape implements Shape {
  // ne compile pas, il manque la méthode draw
}
```

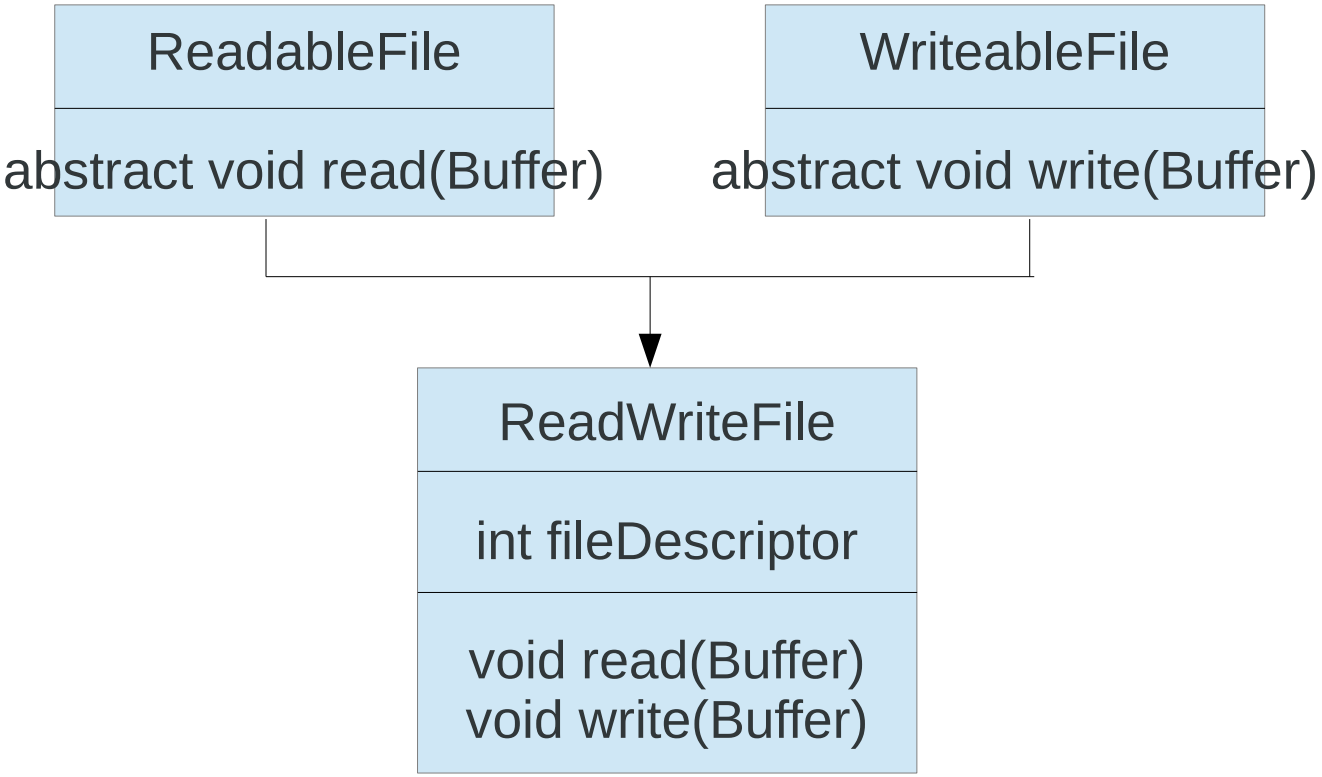
Interface & type abstrait

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

A l'exécution, une référence typé avec l'interface est forcément une référence à une classe qui implante l'interface
Shape shape = ...



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 fournit

```
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 fourni 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 plante

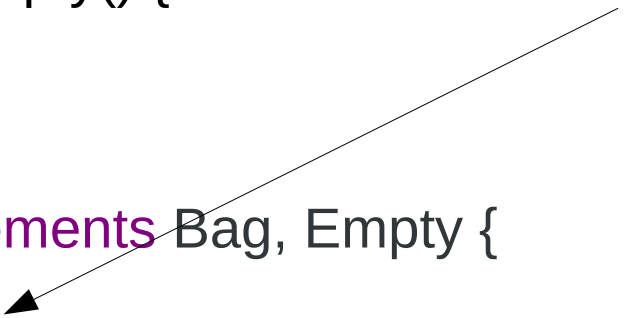
```
public interface Empty {
  public default boolean isEmpty() {
    return true;
  }
}
public class EmptyBag implements Bag, Empty {
  // problème, 2 méthodes isEmpty() par défaut
}
```

Résoudre le conflit

Il faut fournir une implantation pour résoudre le conflit

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

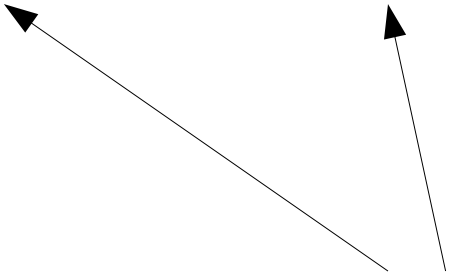
SuperInterface.super permet d'accéder à l'implantation par défaut dans une interface



Trait

On appel un Trait une capacité que l'on peut ajouter à une classe

```
public class Sequence
                implements HasContains, HasIsEmpty {
  private int size;
  private Object[] elements;
  ...
  @Override
  public int size() {
     return size;
  }
  @Override
  public Object get(int index) {
    return elements[index]
  }
}
```



Trait

HasContains et HasIsEmpty sont des traits que l'on combine pour créer la classe Sequence

```
public interface HasContains {
  public abstract int size();
  public abstract Object get(int index);
  public default boolean contains(Object o) {
    for(int i=0; i< size(); i++) {
      if (get(i).equals(o)) {
        return true;
      }
    }
    return false;
  }
}
```

En Java, un trait est simplement une interface avec des méthodes par défaut

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

Cela permet de partager du code (les méthodes par défauts) sans partager de détail d'implantation (les champs restent dans la classe)

Membre d'une interface

Un champ d'une interface est forcément static final, donc une constante

Une méthode d'une interface est forcément public

Une méthode d'une interface est abstract par défaut (sauf si le mot-clé default)

Une méthode peut être static (Java 8)

Hello World avec une interface

```
public interface HelloWorld {
  public static void main(String[] args) {
    System.out.println("hello world");
  }
}
```

permet plutôt d'avoir des méthodes factory qui vont créer des objets de l'interface en utilisant l'implantation par défaut

Héritage d'interface

Une interface peut hériter elle même d'autres interfaces, l'interface est alors 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
}
```

Classe abstraite

Il est possible de définir une classe dont certaine méthode sont abstraites

```
public abstract class ArrayBasedSequence
                                        implements Sequence {
  private Object[] array;
  private final int size;
  ...
  public int size() {
     return size();
  }
  public abstract add(Object o); // définie dans les sous-classes
}
```

Cela permet de partager des champs et du code entre des classes

Classe abstraite et instantiation

Comme une interface, une classe abstraite ne peut être instancier

```
Sequence sequence =
        new ArrayBasedSequence();
```

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 un méthode abstract et static ne veut rien dire

- abstract: on doit la rédéfinir
- 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 accessible par les sous-classes mais pas de l'extérieur

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

Rafinement de l'abstraction

Du plus pure vers l'implantation

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

on peut définir des méthodes statiques partout

Exemple

```
public interface Performer {
  public abstract Object performs();
}
public interface Procedure extends Performer {
  public default void procede() {
    return performs();
  }
}
public abstract class CountedProc implements Procedure {
  private int counter;
  public final Object performs() {
    counter++;
    return apply();
  }
  protected Object apply();
}
public class CountedSayHello extends CountedProc {
   protected Object apply {
     return "hello";
   }
}
```

```
CountedSayHello r1 =
    new CountedSayHello();
Performer r2 = r1;
r2.performs();  // hello
Procedure r3 = r1;
r3.procede();
CountedProc r4 = r1;
r4.getCounter(); // 2
```