Héritage

Sous-typage

- Relation entre types : *T1* est sous-type de *T2* s'il est possible d'utiliser un objet de type *T2* à la place d'un objet de type *T1*.
- Il y a sous-typage si toute instance de *T2* peut remplir staisfaire les même demandes qu'une instances de *T1*.
- Cette relation existe sur les types objets mais pas entre types primitifs et objet (conversion n'est pas sous-typage).
- Le sous-typage permet d'écrire des algorithmes génériques (valable pour tout sous-type)

Covariance et Contravariance

- **Covariance:** le sous-typage du « composant » varie dans le sens du sous-typage d'un type à son sous-type
- **Contravariance:** le sous-type du « composant » varie dans le sens du super-typage d'un type à son sous-type
- Une sous-type `B` de `A` doit pouvoir faire le même travail que `A`:
  - covariance pour les types de retour et l'exception lancée
  - contravariance pour les paramètres

Théorie (indépendant de Java)

- Pour avoir sous-typage entre `A` et `B`
- À toute méthode `m` de `A` :

```
R m(P) throws T
```

- doit correspondre une méthode `m` de `B` :

```
R' m(P') throws T'
```

- où
  - `R'` doit être un sous-type de `R`
  - `T'` doit être un sous-type de `T`
  - `P'` doit être un super-type de `P`

en Java

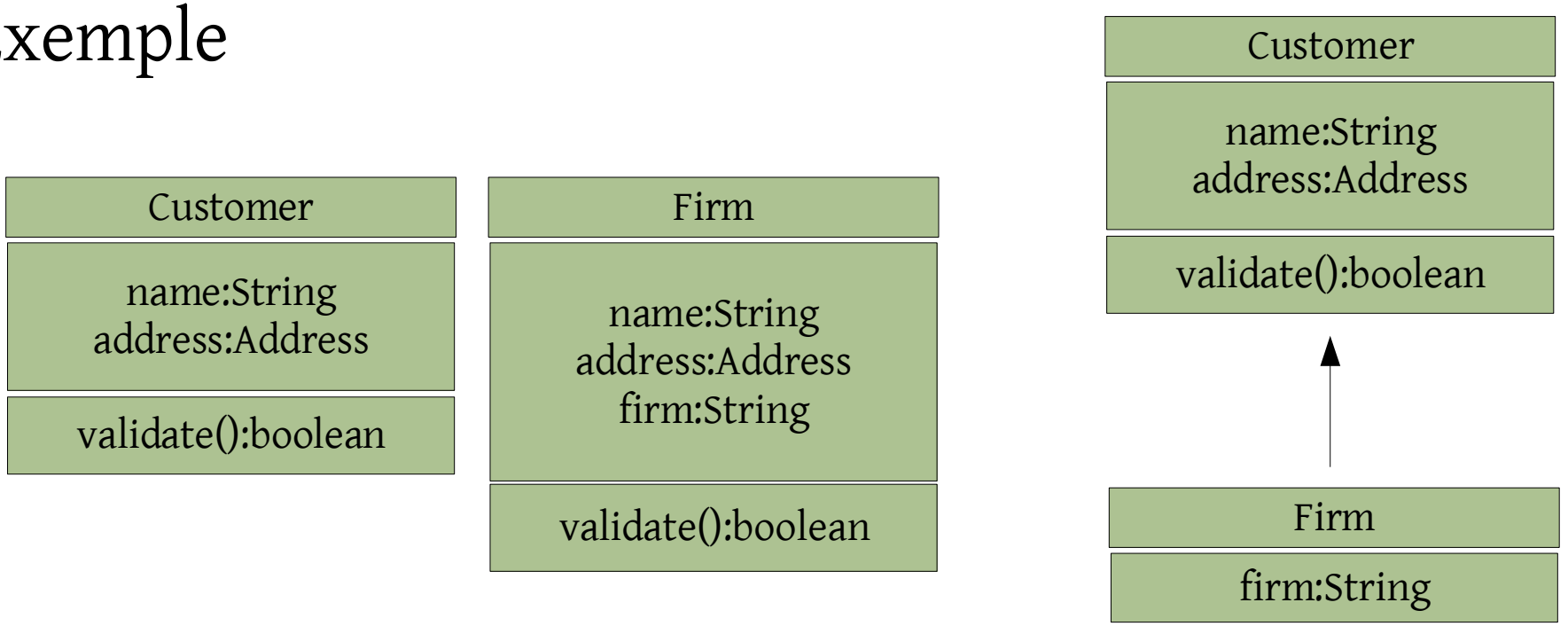
- En Java, tous les types objets sont sous-types de **Object**.
- La relation de sous-typage vient de :
  - **L'héritage**, si A hérite de B, alors A est un sous-type de B
  - **L'implémentation d'interface**, si A implante B alors A est un sous-type de B
  - Des relations sur les **tableaux d'objets**
  - Des relations sur les génériques avec les *wildcard*

    `ArrayList<Integer>` est un sous-type de

    `ArrayList<? extends Number>`

Héritage

- Exemple



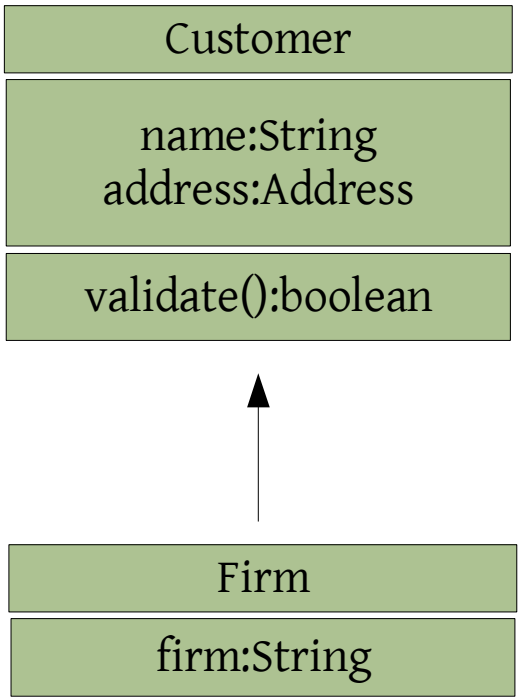
- Actuellement, l'héritage entre classes (concrètes) est relativement rare !!
- Un peu moins rare est la redéfinition de méthode concrètes (`toString`, `equals`, `paintComponent`)

Héritage Simple

- En Java, contrairement au C++, on hérite d'une seule classe
- Il n'est pas possible d'hériter de plusieurs classes
- Le mécanisme d'**interface** (propre à Java) permet de passer outre cette restriction

Définition de l'héritage

- L'héritage, c'est 3 choses :
  - Récupération des membres visibles (héritage structurel)
    (**Firm** possède les
    champs, méthodes et inner de
    Customer)
  - Possibilité de redéfinir les méthodes
    (changer le code de **validate()**)
  - Sous-typage
    (**Firm** est un **Customer**)



Interface

- Une interface définit un type sans code
- On utilise le mot-clef **interface**

```
public interface List {
   public Object get(int index;
   public void set(int index,Object o);
}
```

- Une interface déclare des méthodes sans indiquer le code (implémentation) de celles-ci
  - On dit alors que les méthodes sont abstraites

Design: Interface

- Implémenter une interface pour définir :
  - l'interface (au sens de la chimie) entre deux modules du projet
  - une fonctionnalité transversale (`Comparable`, `Cloneable`)
  - un ensemble de fonctionnalités où plusieurs implémentations sont possibles (`ArrayList`, `LinkedList`)
  - des plugins (*service provider*)

Instantiation d'interface

- Il n'est pas possible d'instantier une interface car celle-ci ne définit pas le code de ses méthodes

```
public interface List {
  public Object get(int index);
  public void set(int index,Object o);
}

public class Main {
  public static void main(String[] args) {
    List list=
      new List(); // illégal
  }
}
```

Implémentation d'interface

- Implémenter une interface consiste à déclarer un classe qui fournira le code pour l'ensemble des méthodes abstraites

```
public interface List {
  public Object get(int index);
  public void set(int index,Object o);
}

public class FixedArrayList implements List {
  public FixedArrayList(int capacity) {
    data = new Object[capacity];
  }
  public Object get(int index) {
    return data[index];
  }
  public void set(int index,Object o) {
    data[index] = o;
  }
  private final Object[] capacity;
}
```

Documentation

- La documentation de l'interface précise les fonctions réalisées par toute implémentation
- Une classe qui implémente une interface se doit de respecter cette documentation
- Dans certains cas très rare, on peut, en le spécifiant, ne pas respecter les contraintes des méthodes (`IdentityHashMap`)

Documentation

subSequence

```
CharSequence subSequence(int start,
                         int end)
```

Returns a new `CharSequence` that is a subsequence of this sequence. The subsequence starts with the `char` value at the specified index and ends with the `char` value at index `end - 1`. The length (in `char`s) of the returned sequence is `end - start`, so if `start == end` then an empty sequence is returned.

Parameters:

`start` - the start index, inclusive
`end` - the end index, exclusive

Returns:

the specified subsequence

Throws:

`IndexOutOfBoundsException` - if `start` or `end` are negative, if `end` is greater than `length()`, or if `start` is greater than `end`

Implémentation d'interface

- Le compilateur vérifie que toutes les méthodes de l'interface sont implémentées par la classe

```
public interface List {
  public Object get(int index);
  public void set(int index,Object o);
}

public class NullList implements List {
  // NullList is not abstract and does not override
  // abstract method set(int,java.lang.Object...)

  public Object get(int index) {
    return null;
  }
}
```

Implémentation d'interface

- Quand le documentation de l'interface le permet, on peut lancer `UnsupportedOperationException` comme implémentation

```
public interface List {
  public Object get(int index);
  /** optional operation */
  public void set(int index,Object o);
}

public class NullList implements List {
  public void set(int index,Object o) {
    throw new UnsupportedOperationException();
  }

  public Object get(int index) {
    return null;
  }
}
```

Interface et Sous-typage

- Le fait qu'une classe implémente une interface implique que la classe est un sous-type de l'interface

```
public class AtWork  {
  String toString(List o,int size) {
    StringBuilder builder = new StringBuilder();
    for(int i=0;i<size;i++)
      builder.append(o.get(i));
    return builder.toString();
  }
  public static void main(String[] args) {
    List list=new FixedArrayList(12); // sous-typage
    System.out.println(toString(list,12));
  }
}
```

- Permet d'écrire une méthode générique, ou de demander une fonctionnalité

Héritage d'interface

- Une interface peut hériter d'une ou plusieurs interfaces
- Les méthodes de cette interface correspondent à l'union des méthodes des interfaces héritées

```
public interface Channel  {
  void close();
}
public interface ReadChannel extends Channel {
  int read(byte[] buffer);
}
public interface WriteChannel extends Channel {
  int write(byte[] buffer);
}
public interface RWChannel extends ReadChannel,
WriteChannel {
}
```

Héritage multiple d'interfaces

- Au niveau programmatique, s'il y a deux fois la même méthode dans deux interfaces différentes, l'héritage multiple ne pose pas de problème car il n'y a pas d'implémentation
- En revanche, les documentations des deux interfaces doivent être semblables voire compatible.

```
interface Holder1<T> {
  /** @return hold
   * object or null
   * if none is net */
  public T get();
}
```

```
interface Holder2<T> {
/** @return hold object
 * @throws NoSuchElementExn
 * if none is set */
  public T get();
}
```

Remarque sur les interfaces

- Rien ne vous empêche d'ajouter des méthodes dans votre implémentation
- Il faut que la variable soit typée du nom de l'implémentation pour utiliser ces méthodes
- Conséquences :
  - une interface n'oblige pas un objet à être non mutable
  - on peut masquer certaines méthodes en typant une implémentation comme une certaine interface
  - mais on n'est pas à l'abris d'un cast !

Remarque sur les interfaces

```
public interface ReadOnlyHolder<T> {
  /** @return hold object or null if none is net */
  public T get();
}

public class Holder<T> implements ReadOnlyHolder<T> {
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t=t;
  }
  public ReadOnlyHolder<T> readOnlyView() {
    return this;
  }
}
```

Interface, méthodes et champs

- Les méthodes déclarées dans une interface sont obligatoirement et implicitement :
  - abstraite (`abstract`) et publique (`public`)
- Les champs déclarés dans une interface sont obligatoirement et implicitement :
  - constant (`final`), public (`public`) et statique (`static`)
- Les classes internes déclarées dans une interface sont obligatoirement et implicitement :
  - publique (`public`) et statique (`static`)

Exemple

```
public interface Holder<T> {
  T get();
  void set(T t);
  class DefaultHolder<T> implements Holder<T> {
    private T t;
    // cannot implement get in Holder: weaker access privileges
    T get() { return t; }
    public void set(T t) { this.t = t; }
  }
}
public interface SocketSystemCallConstant {
  int PF_UNIX=1;
  int PF_LOCAL=1;
  int PF_INET=2;
  int PF_INET6=10;
  int PF_IPX=4;
  …
}
```

Autres remarques

- Éviter, pour récupérer des constantes, d'hériter d'une interface.
- Au « moins pire », utiliser l'import statique.
- Il n'est **pas** possible de définir une **méthode statique** dans une interface
  (James Gosling trouve ça pas beau !)
- Il est possible et encore moins beau de définir une méthode statique dans une classe interne d'une interface
- On utilise **pas** l'annotation **@Override** pour dire qu'une méthode implémente une méthode définie par l'interface

Interface interne

- Il est possible de définir des interfaces à l'intérieurs de classe ou d'interface
- Une interface interne est toujours statique

```
public class C {
  public interface IInC { // interface interne statique
  }
  public Class CInC { // classe interne non statique
  }
}
public interface I {
  public class CInI { // classe interne statique
  }
  public interface IInI { // interface interne statique
  }
}
```

L'héritage de classes

- En Java, l'héritage se fait en utilisant **`extends`**.

```
public class Firm extends Customer {
  final String firm;
}
```

```
public class Customer {
  public Customer(String name, Address address) {
    this.name=name;
    this.address=address;
  }
  public boolean validate() {
    return name!=null && address.validate();
  }
  final String name;
  final Address address;
}
```

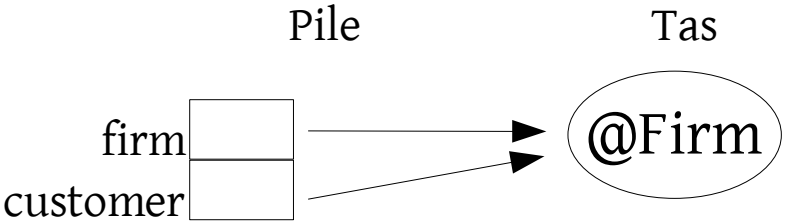
Héritage et Object

- En Java, toutes classes héritent de `java.lang.Object`, directement ou indirectement.
- Directement : si l'on déclare une classe sans héritage, le compilateur rajoute `extends Object`.
- Indirectement : si l'on hérite d'une classe cette dernière hérite de `Object` (directement ou indirectement)

Héritage et sous-typage

- Si `Firm` hérite de `Customer` alors `Firm` est un sous-type de `Customer`

```
public static main(String[] args) {
  Firm firm=new Firm();
  System.out.println(firm);
  Customer customer=firm; // sous-typage
  System.out.println(customer);
}
```

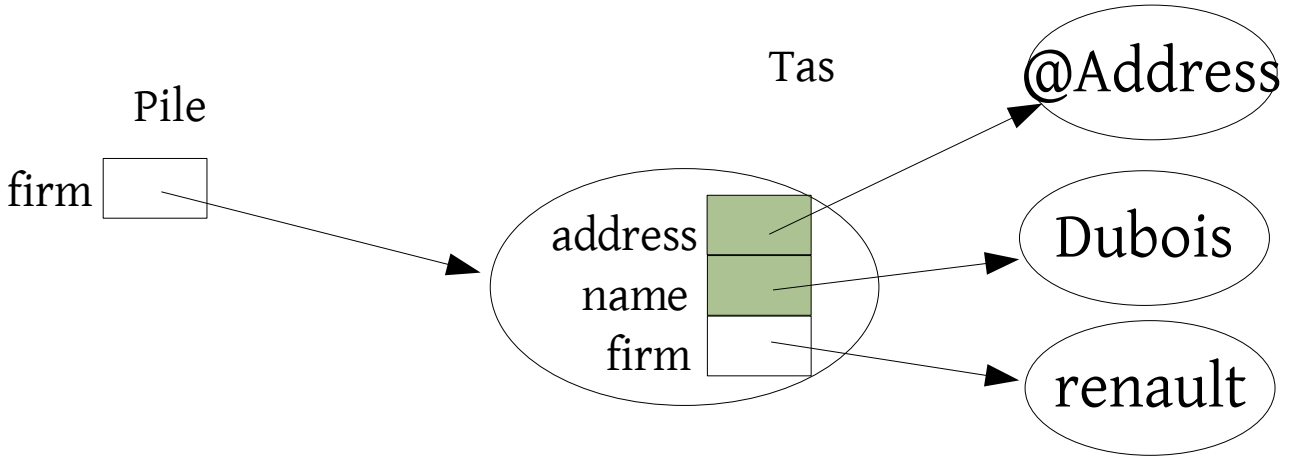


- Partout on l'on attend un objet de type `Customer`, il est possible de donner un objet de type `Firm`

Héritage des membres

- Une classe hérite de tous les membres **visibles** de sa superclasse

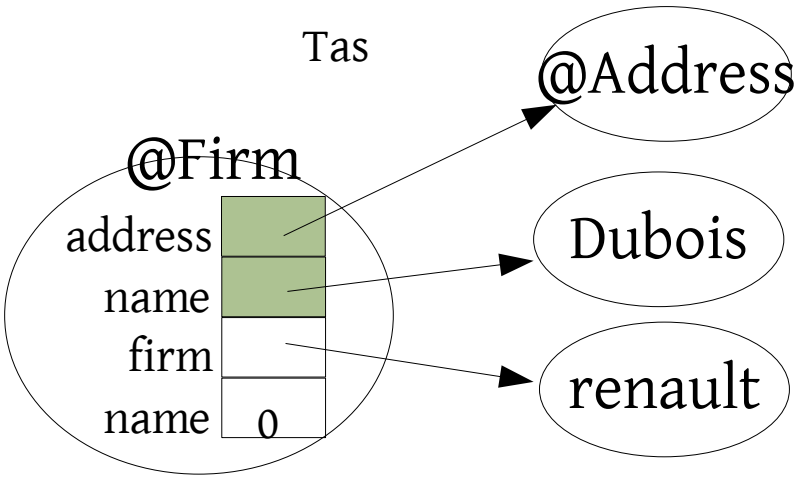
```
public class Firm extends Customer {
  final String firm;

  public static main(String[] args) {
    Firm firm=new Firm();
    System.out.println(firm.firm+" "+
          firm.name" "+firm.address);
  }
}
```



Masquage de champs

- Il est possible de nommer un champs de façon identique dans une sous-classe

```
public class Firm extends Customer
{
  public int f() {
    return this.name;
  }
  final String firm;
  final int name;
}
```



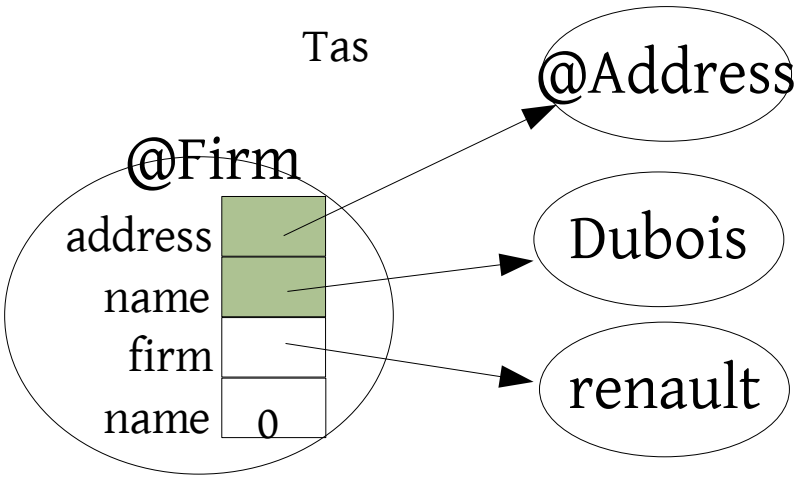
- Les deux champs cohabitent

```
public class Customer {
  public String g() {
    return this.name;
  }
  final String name;
  final Address address;
}
```

Masquage de champs

- On dit que le champs `name` de `Firm` **masque** le champ name de `Customer`

```
public class Firm extends Customer
{
  public int f() {
    return this.name;
  }
  final String firm;
  final int name;
}
```



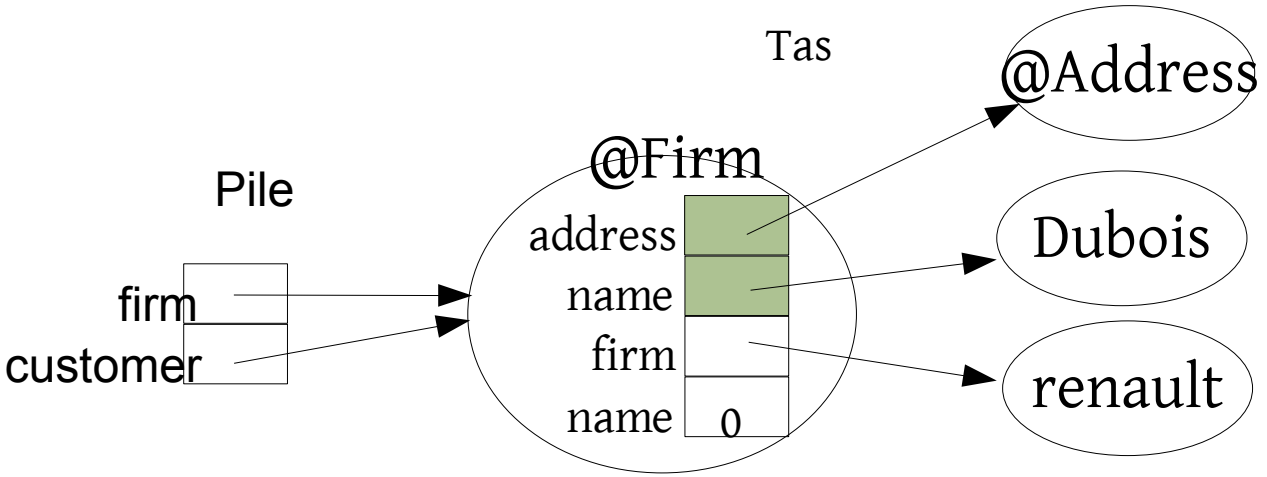
- **super.** permet d'accéder au champs de la super classe

```
public class Customer {
  public String g() {
    return this.name;
  }
  final String name;
  final Address address;
}
```

Champs et sous-typage

- On accède au champs en fonction du **type déclaré** de la référence

```
public static main(String[] args) {
   Firm firm=new Firm();
   System.out.printf("%d",firm.name);

   Customer customer=firm; // sous-typage
   System.out.printf("%s",customer.name);
}
```



Héritage et constructeur

- Un constructeur doit, comme première instruction, faire appel au constructeur de la superclasse
- Sinon, le constructeur par défaut est appelé
- Si l'on n'écrit pas de constructeur le compilateur en ajoute un qui fait appel au constructeur par défaut de la superclasse.
- Dans ces deux derniers cas, la superclasse doit avoir un constructeur par défaut

```
public class Firm extends Customer {
   final String firm;
   public Firm() {
      super(); // bang
   }
} // cannot find symbol  constructor Customer()
```

implicite

Constructeur et superclasse

- L'appel au constructeur de la superclasse se fait avec `super(...)`
- Les arguments de super sont compilés dans un contexte statique

```
public class Firm extends Customer {
  public Firm(String firm, String name, Address address) {
    super(name,address); // appel à Customer(String,Address)
    this.firm=firm;
  }
  private final String firm;
}
```

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

- Notez que les champs sont private

Constructeur vs méthode

- La grosse différence entre un constructeur et une méthode est que l'**on n'hérite pas des constructeurs**
- Le constructeur de la classe héritée a pour mission de :
  - demander l'initialisation de la classe mère au travers de son constructeur
  - d'initialiser ses propres champs (il **doit** initialiser tous les champs `final`)

Sous-typage et tableau

- En Java, les tableaux possèdent un sous-typages exotiques (cf JLS) :
  - Un tableau est un sous-type de `Object`, `Serializable` et `Cloneable`
  - Un tableau de `U[]` est un sous-type de `T[]` si `U` est un sous-type de `T`.

```
public class InterfaceTyping {
  public static void main(String[] args) {
    Object[] o=args;
    double[] array=new int[3]; // illégal
    Integer[] array2=new int[3]: // illégal
  }
}
```

  - la méthode `clone` renvoie le bon type et non `Object`

ArrayStoreException

- Comme le sous-typage sur les tableaux existe, cela pose un problème :

```
public class InterfaceTyping {
  public static void main(String[] args) {
    Object[] o=args;
    o[0]=new Object();
    // ArrayStoreException à l'exécution
  }
}
```

- Il est possible de considérer un tableau de `String` comme un tableau d'`Object` mais il n'est pas possible d'ajouter un (vrai) `Object` à ce tableau (test à l'exécution)

Polymorphisme

- Le polymorphisme consiste à considérer les fonctionnalités suivant le type réel d'un objet et non suivant le type de la variable dans laquelle il est stocké
- Le sous-typage permet de stocker un objet comme une variable d'un supertype, le polymorphisme fait en sorte que les méthodes d'objet soient appelées en fonction du type réel de l'objet
- On appelle ce mécanisme « appel virtuel »

À quoi ça sert ?

- Le polymorphisme fait en sorte que certaines parties de l'algorithme soit spécialisée en fonction du type réel de l'objet

```
public class Polymorphic {
  private static void print(Object[] array) {
    for(Object o:array)
      System.out.println(o);
      // appel dynamiquement String.toString()
  }
  public static void main(String[] args) {
    print(args);
  }
}
```

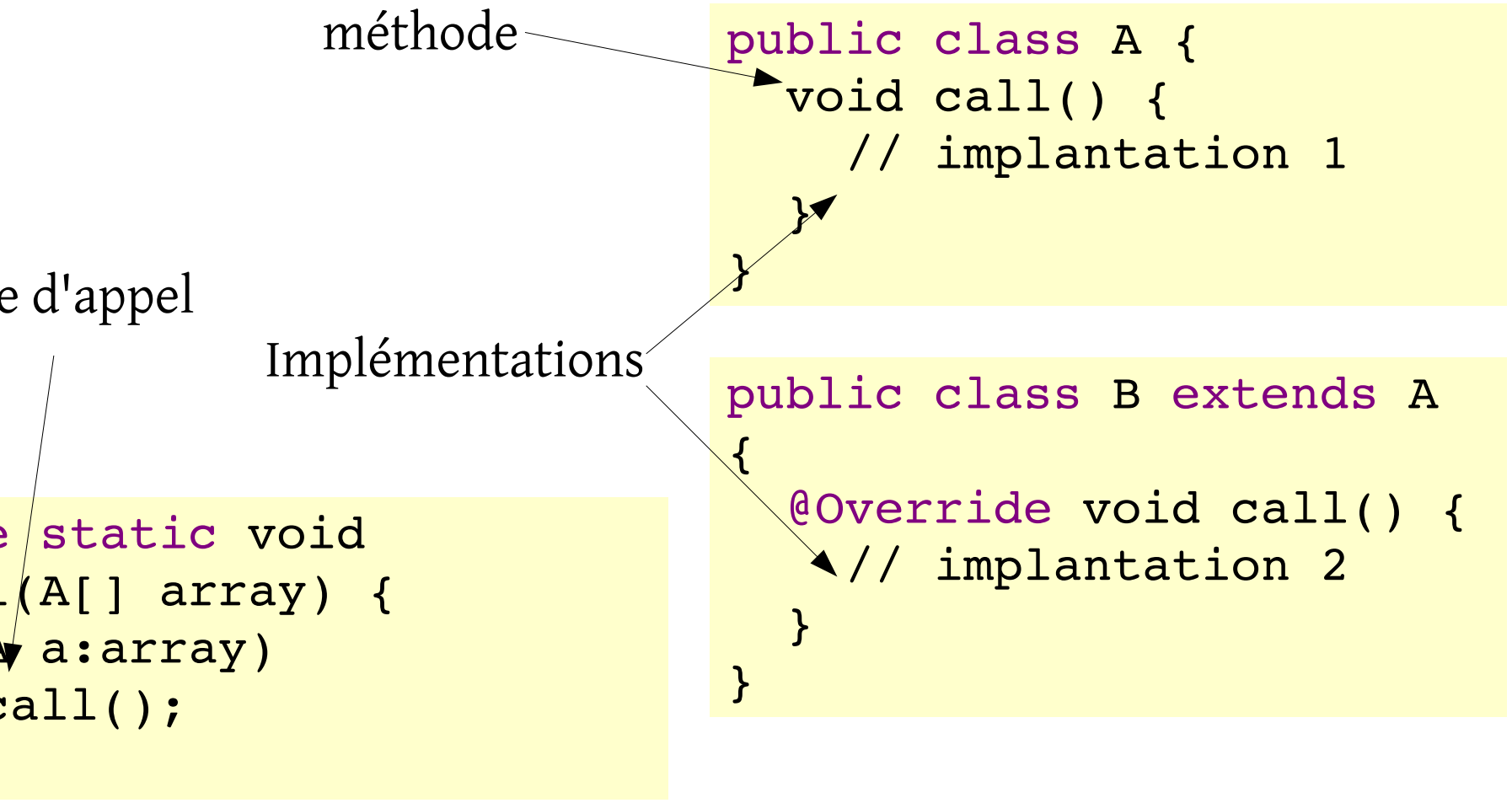
- En Java, seul l'appel de méthode est polymorphe par rapport à l'objet sur lequel la méthode est appelée (receveur : objet avant le point)

Condition d'appel virtuel

- Il n'y a pas d'appel virtuel (invokeSpecial) si la méthode est :
  - **statique** (pas de receveur)
  - **private** ou de paquetage dans un paquetage différent (pas de redéfinition possible, car non visible)
  - si l'appel utilise **super**
  - les appels de constucteurs, et appels de blocs d'initialisation générés par la compilateur
- Dans les autres cas, l'appel est virtuel (invokeVirtual)
  - this.truc() comme truc() est un appel virtuel

Implantation et site d'appel

- On distingue la méthode, les implémentations de cette méthode (*callee*) et le site d'appel à la méthode (*caller*)



Condition de la redéfinition

- Il y a redéfinition de méthode s'il est possible pour un site d'appel donné d'appeler la méthode redéfinie en lieu et place de la méthode choisie à la compilation
- En plus, elles doivent être toutes les deux paramétrés ou toutes les deux non paramétrées
- Le fait qu'une méthode redéfinisse une autre dépend :
  - Du nom de la méthode
  - Des modificateurs de visibilité des méthodes
  - De la signature des méthodes
  - Des exceptions levées (throws) par la méthode

Visibilité et redéfinition

- Il faut que la méthode redéfinie ait une visibilité au moins aussi grande (covariance) (ε<protected<public)

```
public class A {
  protected void call() {
    // implantation 1
  }
}
```

compilation

```
private static void
callAll(A[] array) {
  for(A a:array)
    a.call();
}
```

exécution

```
public class B extends A {
  @Override public void call() {
    // implantation 2
  }
}
```

Héritage de méthode

- En tant que membre, une sous-classe hérite des méthodes de la superclasse

```
public class Firm extends Customer {
  public Firm(String firm, String name, Address address) {
    super(name,address);
    this.firm=firm;
  }
  private final String firm;

  public static main(String[] args) {
    Address address=...
    Firm firm=new Firm("renault","Dubois",address);
    System.out.println(firm.validate()); // true
    Firm firm=new Firm(null,"Dubois",address);
    System.out.println(firm.validate()); // true
  }
}
```

- `validate()` ne verifie pas le champs `firm`

Redéfinition de méthode

- Il est possible de redéfinir le code de `validate()` dans `Firm`

```
public class Firm extends Customer {
  public Firm(String firm, String name, Address address) {
    super(name,address);
    this.firm=firm;
  }
  public boolean validate() {
    return firm!=null && name!=null && address.validate();
  } // marche pas
  private final String firm;
}
```

- Permet d'avoir le code adapté à la sémantique de la méthode

Assurer la redéfinition

- L'annotation `@Override` indique au compilateur de générer une erreur si une méthode ne redéfinit pas une autre

```
public class Firm extends Customer {
  public Firm(String firm, String name, Address address) {
    super(name,address);
    this.firm=firm;
  }
  public @Override boolean validate() {
    ...
  }
  private final String firm;
}
```

Redéfinition vs Surcharge

- La surcharge correspond à avoir des méthodes de même nom mais de profils différents dans une même classe
- La redéfinition correspond à avoir deux méthodes de même nom et de même profils dans deux classes où l'une hérite de l'autre
- Le compilateur choisit à *compile time* la méthode à appeler parmi les différentes surcharges en fonction des types déclarés des arguments
- C'est la méthode redéfinie par le type réel qui est appelée à *runtime*

Redéfinition ou Surcharge

- Exemple de surcharge et de redéfinition :

```
public class A {
  public void m(CharSequence a) {...}    // surcharge
  public void m(List<Character> a) {...} // surcharge
}

public class B extends A {
  public void m(Object a) {...}   // surcharge
  public void m(CharSequence a) {...} // redéfinition
}
```

- `m(CharSequence)` de `B` redéfinit `m(CharSequence)` de `A`, le reste est de la surcharge
- Règle de programmation (pas toujours dans le JDK) deux méthodes surchargées doivent effectuer la même chose (même doc)

Redéfinition ou Surcharge

- Exemple de surcharge et de redéfinition :

```
public class A {
  public void m(CharSequence a) {...}    // surcharge
  public void m(List<Character> a) {...} // surcharge
}

public class B extends A {
  public void m(Object a) {...}   // surcharge
  public void m(CharSequence a) {...} // redéfinition
}

public static void main(String[] args) {
  B b = new B();
  A a = b;
  String s = args[0];
  a.m(s); // B.m(CharSequence)
  Object o = s;
  a.m(o); // erreur
  b.m(o); // B.m(Object)
  b.m(Arrays.asList(new Character[] {'a','b','c'}));
    // A.m(List<CharSequence>);
```

Méthodes variadiques

- Avec `...`, on peut définir des méthodes variadiques

```
public class Variadique {
  public static void printAll(int... a) {
    for(int i=0;i<a.length;i++)
      System.out.println(a[i]);
  }

  public static void main(String... args) {
    printAll(1,2,3,4);
  }
}
```

- Contrairement au C, un tableau contenant les paramètres est créé lors de l'appel et passé à la méthode
- `A...` et `A[]` sont considérés comme le même type pour la redéfinition et `main`

Méthodes variadiques

- On peut passer un `T[ ]` à une méthode demandant un `T...`
- C'est d'ailleur cet appel qui est prioritaire :

```
public class Variadique {
  public static int numArgs(Object... a) {
    return a.length;
  }

  public static void test() {
    String[] args = {"un","deux","trois"};
    System.out.println(numArgs(1,2,3,4)); // 4
    System.out.println(numArgs(args)); // 3
  }
}
```

Appel virtuel & compilation

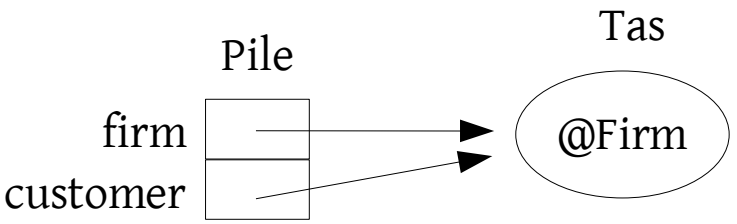
- En java, le mécanisme d'appel polymorphe (appel virtuel) est décomposé en deux phases :

  1) Lors de la **compilation**, le compilateur choisit la méthode la **plus spécifique** (erreur en cas d'ambigüité) en fonction du **type déclaré** des arguments, et si aucune ne convient, une fonction variadique

  2) Lors de **l'exécution**, la VM choisit la méthode en fonction du **type réel** du receveur, et seulement du receveur (l'objet avant le .)

Méthodes et Sous-typage

- On accède aux méthodes en fonction du type réel de la référence




```
public static main(String[] args) {
  Firm firm=new Firm(...);
  System.out.println(firm.validate());    // Firm::validate()
  Customer customer=firm; // sous-typage
  System.out.println(customer.validate());// Firm::validate()
}
```

- Comme il y a redéfinition, la méthode `validate()` de `Customer` n'est pas accessible depuis un objet de type `Firm`

Redéfinition et `super.`

- La notation `super.` ou `super()` permet d'avoir accès aux membres non static de la superclasse (champs, méthodes et constructeurs)

```
public class Firm extends Customer {
  public Firm(String firm, String name, Address
address) {
    super(name,address);
    this.firm=firm;
  }
  public boolean validate() {
    return firm!=null && super.validate();
  }
  private final String firm;
}
```

- `super.super.validate()` ne marche pas
- Contrairement à `this.`, l'appel n'est **pas** polymorphe

Accès hors de la classe

- Il n'est pas possible d'accéder aux méthodes redéfinies hors de la sous-classe

```
public static main(String[] args) {
  Firm firm=new Firm(...);
  System.out.println(firm.validate());
  Customer customer=firm; // sous-typage
  System.out.println(Customer.super.validate());
} // erreur
```

- `super.` se marche que dans la sous-classe
- En revanche, une méthode peut utiliser `super.` pour appeler une autre méthode de la superclasse.

Exemple bidon

- C'est assez rare de devoir faire cela :

```
public class A {
  public void f() {...}
  public void g() {...}
}

public class B extends A {
  @Override public void f() {
    super.g();
  }
  @Override public void g() {
    super.f();
  }
}
```

Méthode `final`

- Si une méthode est déclarée `final`, celle-ci ne peut être redéfinie
  - Important pour assurer la pérénité de son code
  - Important de temps en temps en terme de sécurité
  - Peu voire pas d'impact en terme de performance

```
public class PasswordValidator {
  public final boolean checkPassword(char[] password) {
    ...
  }
}
public class YesValidator extends PasswordValidator {
  @Override public boolean checkPassword(char[] password) {
    // illégal
    return true;
  }
}
```

Classe `final`

- Si une classe est déclarée `final`, il est impossible de créer des sous-classes
  - Mêmes raisons que pour les méthodes

```
public final class PasswordValidator {
  public  boolean checkPassword(char[] password) {
    ...
  }
}
public class YesValidator extends PasswordValidator {
  // illégal
  ...
}
```

- Beaucoup de classes du JDK sont finales (`String`, `Class`), certaines pour raison de performances (plus d'actualité).

Sous-typage et classes internes

- On accède aux classes internes en fonction du **type déclaré** de la référence

```
public class Firm extends Customer {
  final String firm;
  class X {
    int y;
  }
}
```

```
public class Customer {
  final String name;
  final Address address;
  class X {
    int x;
  }
}
```

```
public static main(String[] args) {
  Firm firm=...
  System.out.println(firm.new X().y);
  Customer customer=firm; // sous-typage
  System.out.println(customer.new X().x);
}
```

- Les classes internes comme les champs ne peuvent être redéfinis

Les membres statiques

- On accède aux membres statiques en indiquant la classe
- Ils ne peuvent donc être redéfinis

```
public class Firm extends
Customer {
  final String firm;
  static int m() {
  }
  static int x;
}
```

```
public class Customer {
  final String name;
  final Address address;
  static String m() {
  }
  static String x;
}
```

```
  public static main(String[] args) {
    System.out.printf("%d\n",Firm.m());
    System.out.printf("%s\n",Customer.m());
    System.out.printf("%d\n",Firm.x);
    System.out.printf("%s\n",Customer.x);
  }
```

Héritage Multiple

- Problèmes de l'héritage multiple :
  - Si on hérite de deux méthodes ayant même signature dans deux super classes, quelle code choisir ?
  - Performance en cas de sous-typage
- Solution :
  - Il n'y a **pas d'héritage multiple** en **Java**
  - Java définit des **interfaces** et permet à une classe d'implémenter **plusieurs interfaces**

Classe abstraite

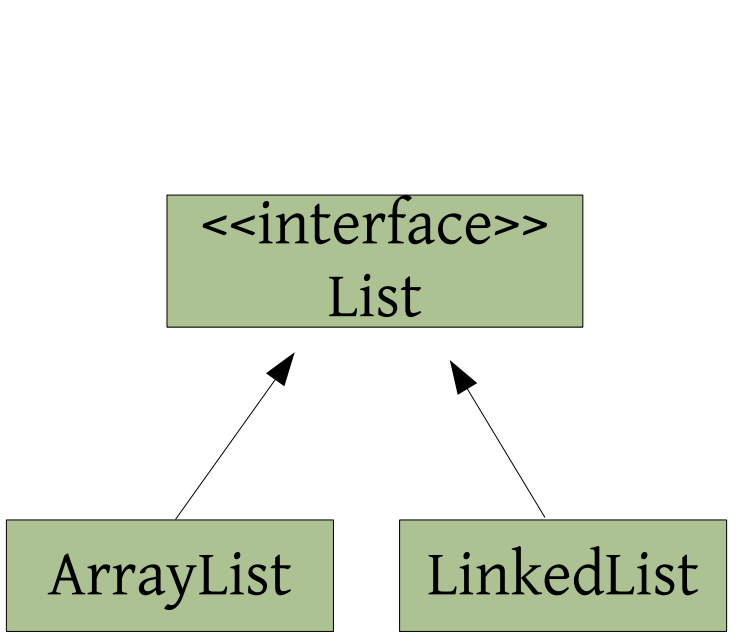
- Il est possible de définir en Java des classes ayant des méthodes abstraites

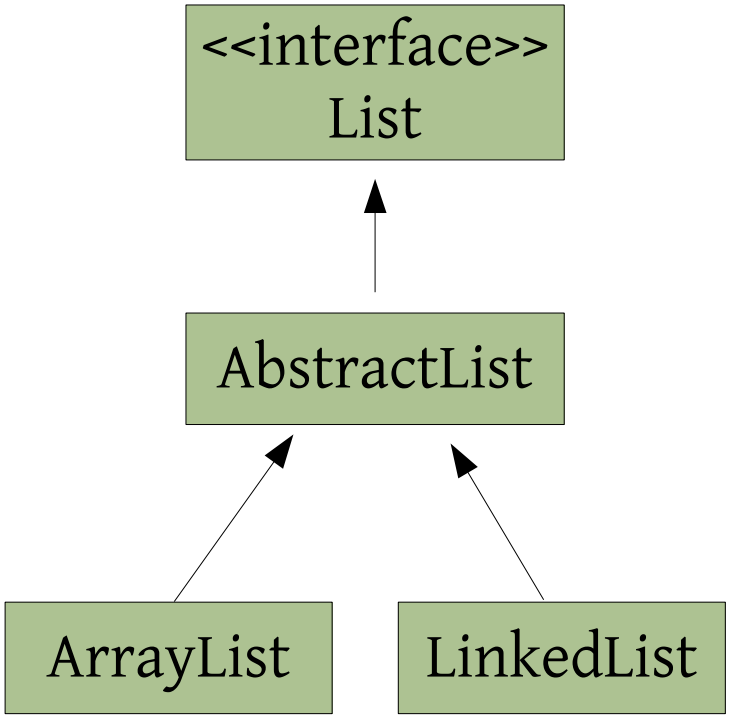
```
public interface List {
  public Object get(int index);
  public void set(int index,Object o);
}

public abstract class NullList implements List {
  public Object get(int index) {
    return null;
  }
}
```

- Une classe abstraite est une classe partiellement implantée donc non instantiable (même si elle ne contient aucune méthode abstraite)

Raffinement de l'abstraction

- Une classe abstraite peut s'intercaler dans l'arbre d'héritage entre l'interface et les classes concrètes pour y mettre le code commun
- Elle permet aussi de simplifier l'implémentation d'interfaces (voir `AbstractList`,`Set`...)





Intérêt des classe abstraite

- Ainsi, pour créer une nouvelle implémentation de java.util.List, il suffit d'hériter d'AbstractList et d'implémenter les méthodes
  - T get(int)
  - int size()
- On peut aussi créer une classe abstraite pour effectuer des actions communes à plusieurs problèmes, et dont seul une seule partie change : ce sera les méthodes abstraites ; les différentes classes seront obtenues en sous-classant et implémentant les méthodes abstraites
- On utilise cela quand on ne désire pas utiliser ces actions communes dans d'autres modules

Exemple

```
public abstract class Computer {
  int bigCalculus(int x,int y) {
    ... f(...) ... f(...) ...
  }
  abstract int f(int x);
  class LambdaComputer extends Computer {
    @Override int f(int x) { ... }
  }
  class AlphaComputer extends Computer {
    @Override int f(int x) { ... }
  }
  private final static Computer lambda = new LambdaComputer();
  private final static Computer alpha = new AlphaComputer();
  // ici, on utiliserait des classes anonymes
  public static int alpha(int x,int y) {
    return alpha.bigCalculus(x,y);
  }
  public static int lambda(int x,int y) {
    return lambda.bigCalculus(x,y);
  }
}
```

Autre design

```
public class Computer {
  public interface Element {
    int f(int x);
  }
  public static int bigCalculus(Element element,int x,int y) {
    ...
  }
}
```

```
private final static Computer lambda = new Computer.Element() {
    public int f(int x) { ... }
}
private final static Computer alpha = new Computer.Element() {
    public int f(int x) { ... }
}
public static int alpha(int x,int y) {
    return Computer.bigCalculus(alpha,x,y);
  }
public static int lambda(int x,int y) {
  return Computer.bigCalculus(lambda,x,y);
}
```

Test dynamique de type

- Il est possible de tester si un objet est un sous-type d'un type particulier grâce à l'opérateur `instanceof`

```
public class ClassExample {
  public static void main(String[] args) {
    Object o;
    if (args.length==0)
      o="tutu";
    else
      o=new Object();
    boolean test=o instanceof String;
    System.out.println(o); // true ou false
  }
}
```

Le transtypage

- On appelle transtypage le fait de voir une référence sur un type comme une référence sur un sous-type

```
public class ClassExample {
  public static void main(String[] args) {
    Object o;
    if (args.length==0)
      o="tutu";
    else
      o=new Object();
    String s=(String)o;
    // peut faire à l'exécution un ClassCastException
  }
}
```

- Attention cette opération peut lever une exception `ClassCastException` (décelable seulement à *runtime*)

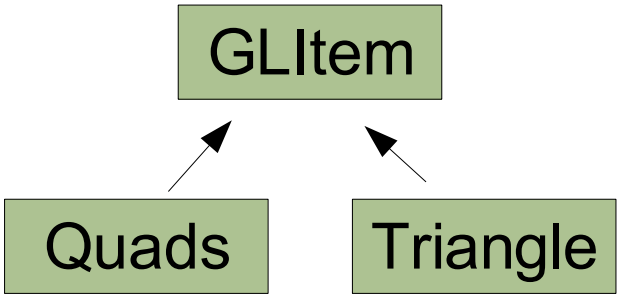
Assurer le transtypage

- Il est possible d'assurer un transtypage sans lever d'exception en utilisant `instanceof`

```
public class ClassExample {
  public static void main(String[] args) {
    Object o;
    if (args.length==0)
      o="tutu";
    else
      o=new Object();

    if (o instanceof String) {
      String s=(String)o;
      doSomeStuff(s);
    }
  }
}
```

Mauvais usage du `instanceof`

- Le `instanceof` ne doit pas se substituer au polymorphisme

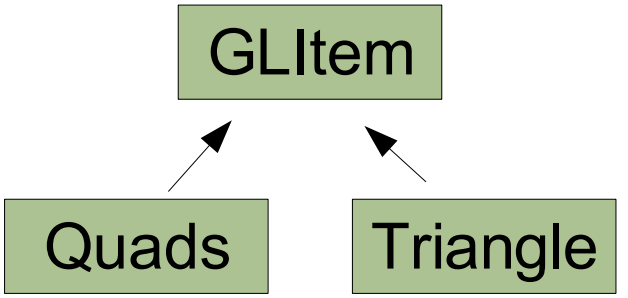



```
public static void renderLoop(GLItems[] items) {
  glBegin();
  for(GLItem item:items) {
    if (item instanceof Quads)
      renderQuads((Quads)item);
    else
      renderTriangle((Triangle)item);
  }
  glEnd();
}
```

- Moins maintenable, et si grosse hiérarchie, beaucoup moins performant

Utiliser le polymorphisme !

- Item definit une méthode abstraite render, Quads et Triangle l'implante

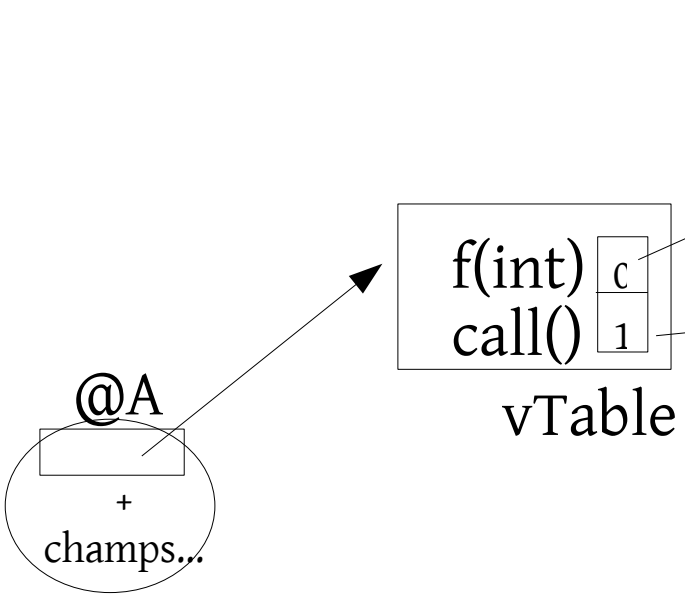



```
public static void renderLoop(GLItems[] items) {
   glBegin();
   for(GLItem item:items) {
     item.render();
   }
   glEnd();
}
```

- Pour utiliser le polymorphisme, il faut avoir accès au code des classes
- *Design pattern* du visiteur.

Implantation du polymorphisme

- Chaque objet possède en plus de ces champs un pointeur sur une table de pointeurs de fonctions
- La machine virtuelle attribue à chaque méthode un index dans la table en commençant par numéroter les méthodes des superclasses

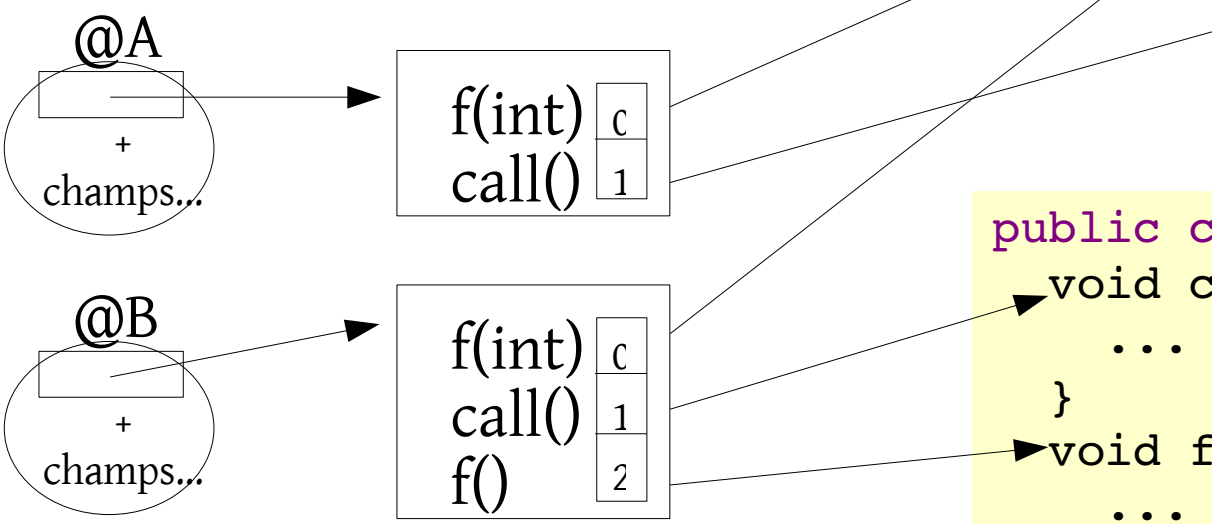



```
public class A {
  void f(int i){
     ...
  }
  void call(){
     ...
  }
}
```

vTable

- Deux méthodes redéfinies ont **le même index**
- L'appel polymorphe est alors :

object.vtable[index](argument)




```
public class A {
  void f(int i){
     ...
  }
  void call(){
     ...
  }
}
```

```
public class B extends A {
  void call(){
     ...
  }
  void f(){
     ...
  }
}
```

- Problème si héritage multiple (interfaces en java) !

Rappel : Classe Anonyme

- Il est possible d'implanter une interface sans donner de nom à la classe

```
interface Filter {
  boolean accept(File file);
}

public class Test {
  public File[] subDirectories(File directory) {
    directory.listFiles(new Filter() {
      public boolean accept(File file) {
        return file.isDirectory();
      }
    });
  }
}
```

- Ici, on crée un objet d'une classe implantant l'interface Filter

Pour le debug

- Utiliser des classes anonymes permet de debuger à la printf du code :

```
public static void main(String[] args) {
   ArrayList<String> list = new ArrayList<String>() {
      @Override public String get(int i) {
        System.err.println("index : "+i);
        return super.get(i);
      }
   };
   ...
}
```