La prog. objet

Rémi Forax

La programmation objet

Le but principal de la programmation objet est d'aider à la conception et la maintenance de logiciels.

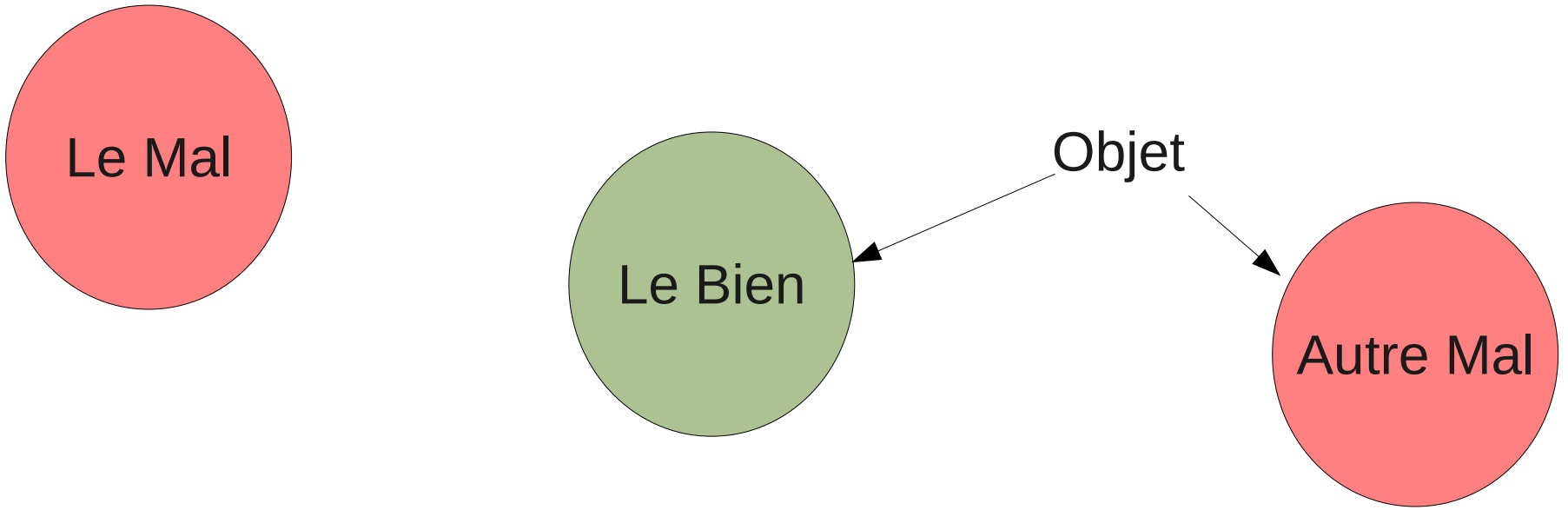
- Modularité
- Extensibilité
- Polymorphisme
- Réutilisabilité

Un objet

- Un objet est la plus petite unité modulaire, extensible et réutilisable
- Il représente :
  - un objet physique (table, chaise, etc)
  - un concept (comparateur, factory, manager)
- Il possède :
  - des données propres qu'il ne montre à personne
  - des fonctionnalités accessibles à tous

Le Monde selon Bush

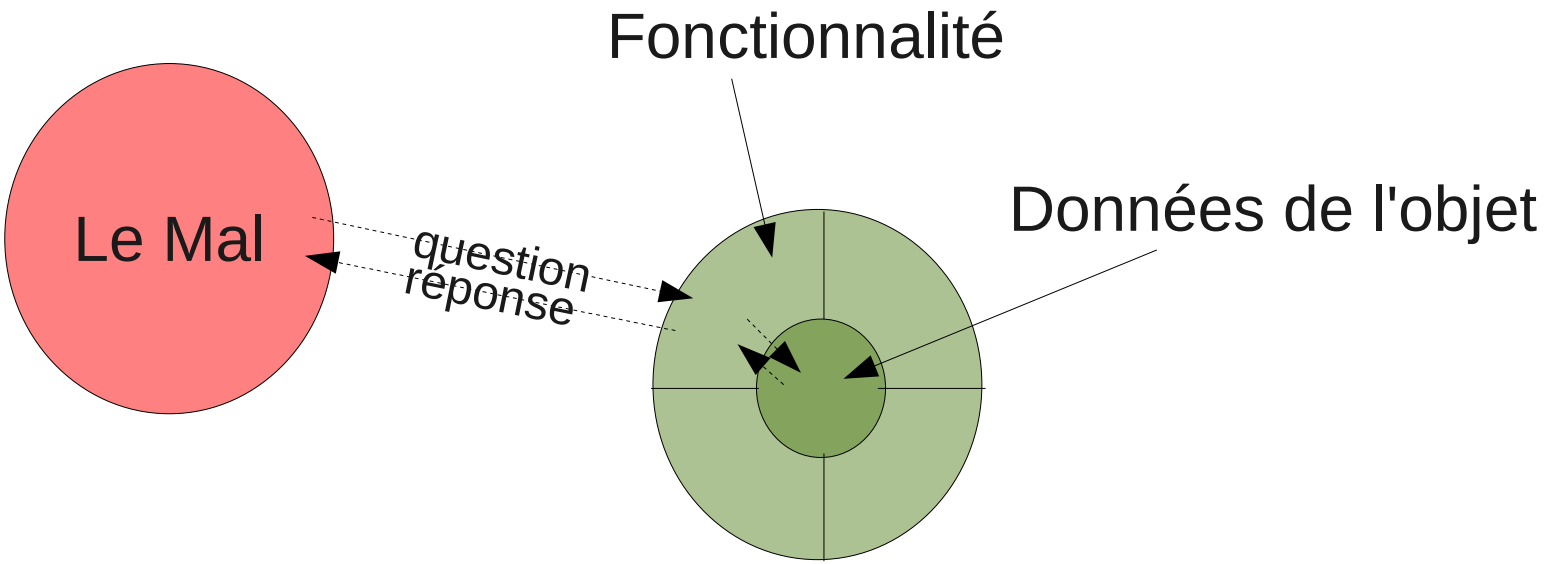
Le Bien et le Mal® :



Un objet (le bien) évolue dans un monde cerné par le mal (les autres)

Le Monde selon Bush (2)

- Cependant, il doit faire du commerce avec les autres :



- Un objet protège son **intégrité** des autres objets, il n'expose que des fonctionnalités.

Membres d'un objet

- Membres d'un objet:
  - des champs qui définissent ses données
  - des méthodes qui manipule les données

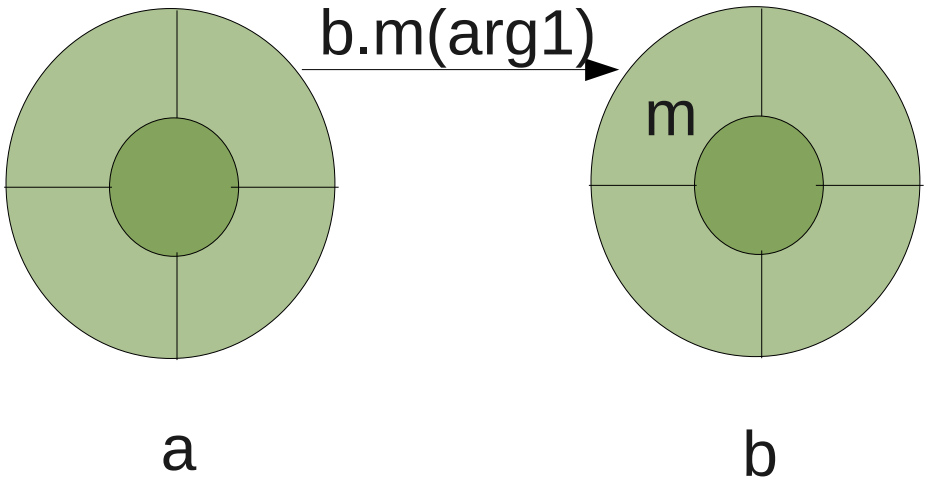
```
class AutreClass {
  void autreMethod(Point p) {
    System.out.println(p.x);
    System.out.println(p.x);
    p.translate(2,3);
  }
}
```

```
class Point {
  void translate(int dx,int dy) {
    x+=dx;
    y+=dy;
  }
  int x;
  int y;
}
```

Accès aux membres

- L'accès aux méthodes d'une classe se fait par la notation “.” sur une référence à un objet.

- En C : m(b,arg1)
- En Java : b.m(arg1)



Notation objet

- Par rapport à une fonction classique, on privilégie un des paramétres qui devient l'objet sur lequel on appel la méthode
- En C : equals(o1,o2)
- En Java : o1.equals(o2)

```
class Matrix {
  boolean equals(Matrix m) {
    ...
  }
}
```

```
class Main {
  public void toto() {
    Matrix m1=...
    Matrix m2=...
    if (m1.equals(m2)) {
      // blah blah
    }
  }
}
```

this

- **this** (ou self, it) correspond à la référence de l'objet sur lequel la méthode courante a été appelée

```
class Main {
  public void toto() {
    Matrix m1=...
    Matrix m2=...
    if (m1.equals(m2)) {
      // blah blah
    }
  }
}
```

```
class Matrix {
  boolean equals(Matrix m) {
    if (this==m)
      return true;
    ...
  }
}
```

- Ici, **this** aura la valeur de m1 et m la valeur de m2

Principes de la lutte

- Principaux principes de la programmation objets :
  - 1 objet 1 responsabilité
  - Encapsulation
  - Immutabilité
  - Type et Classe
  - Découplage interface/implantation
  - Abstraction et concept
  - Principe de localité et polymorphisme
  - Objet/pas objet

1 objet/1 responsabilité

- Si l'on veut réutiliser des objets il ne faut pas qu'il apporte trop de fonctionnalité
- 2 objets => 2 responsabilité, etc
- Utiliser la délégation entre objet (le fait qu'un objet en connaisse un autre) pour séparer les responsabilités

Exemple

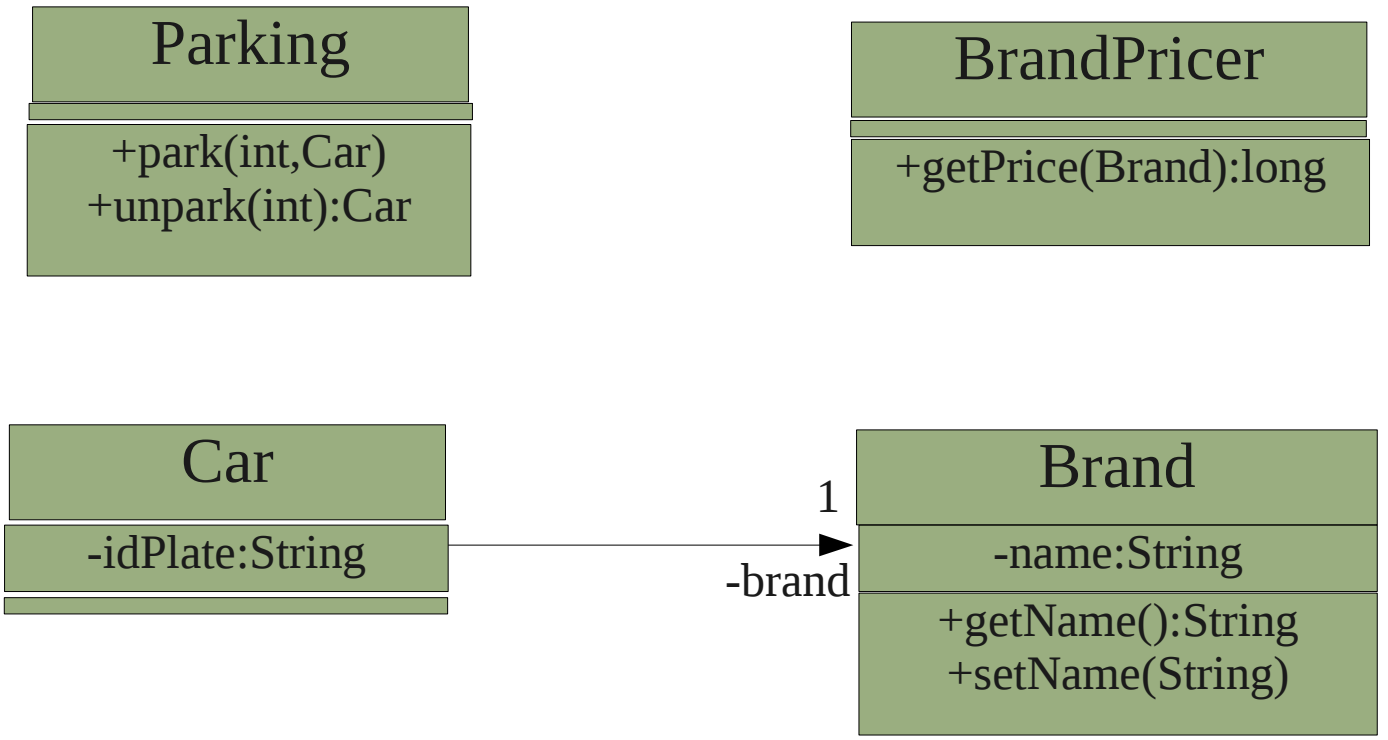
- On veut une application qui gère les places de parking et calcule à tout instant combien le parking va rapporter si toutes les voitures sortent maintenant
- Le prix à payer dépend uniquement du type de voiture

Exemple (suite)

- Ici, 4 responsabilité, 4 objets:
  - Parking: stocker des voitures
  - Car: la voiture en elle-même
  - Brand: le modèle de voiture
  - BrandPricer: indique le prix en fonction d'un modèle

Exemple (fin)

- Diagramme de classe correspondant



Encapsulation

- L'encapsulation est un principe qui garantie qu'aucun champ d'un objet ne pourra être modifier pour corrompre l'état d'un objet sans une vérification préalable

- L'état ne pourra être modifié (s'il est modifiable) que par des méthodes vérifiant les données rentrées

Encapsulation

- Exemple d'un point en coordonnée polaire

```
class AutreClass {
  void autreMethod(Point p) {
    p.init(2.0,3.0);
    System.out.println(p.theta);

    p.rho=-1; // aie aie !!
  }
}
```

```
class Point {
  void init(double x,double y) {
    rho=Math.hypot(x,y);
    theta=Math.atan2(x,y);
  }
  double rho;
  double theta;
}
```

- La classe Point ne garantie pas l'encapsulation !

Visibilité des membres

- Certain langage comme Smalltalk ne permette pas l'accès au champs
- D'autre C++, Java demande à ce que l'on écrive des modificateurs de visibilité

- La visibilité permet d'assurer l'encapsulation en interdisant l'accès aux champs directement

Visibilité et encapsulation

- Un champs est **toujours** privé !!

```
class Point {
  public void init(double x,double y) {
    rho=Math.hypot(x,y);
    theta=Math.atan2(x,y);
  }
  private double rho;
  private double theta;
}
```

```
class AutreClass {
  void autreMethod(Point p) {
    p.init(2.0,3.0);
    System.out.println(p.theta); // ne compile pas
    p.rho=-1; // ne compile pas
  }
}
```

Mais il y a un problème

- Supposons que l'on souhaite que rho ne soit jamais égal à zéro
- On peut obtenir la valeur des champs avant l'initialisation

```
class Point {
  public void init(double x,double y) {
    rho=Math.hypot(x,y);
    theta=Math.atan2(x,y);
  }
  public String toString() {
    return rho+","+theta;
  }
  private double rho;
  private double theta;
}
```

```
class AutreClass {
  void autreMethod(Point p) {
    System.out.println(p.rho);  // affiche 0,0 oups
    p.init(2.0,3.0);
  }
}
```

Constructeur

- On ne peut créer un objet sans appeler de constructeur
- Le constructeur est écrit pour garantir les invariants

```
class Point {
  public Point(double rho,double theta) {
    if (rho<=0)
      throw new IllegalArgumentException(
        "illegal rho "+rho);
    this.rho=rho;
    this.theta=theta;
  }
  public String toString() {
    return rho+","+theta;
  }
  private double rho;
  private double theta;
}
```

Constructeur

- Un constructeur est une méthode particulière
- Pour garantir que les invariants d'un objet sont conservés, il faut pouvoir initialiser un objet avec des valeurs particulières
- Le constructeur va être appelé par **new**
  une fois la mémoire reservé pour initialiser l'objet.

Constructeur (2)

- Un constructeur possède le même nom que la classe et pas de type de retour

```
public class Point {
  public Point(double x,double y) {
    this.x=x;
    this.y=y;
  }
  double x; // pas private, arg !!
  double y;
}
```

```
public class AnotherClass {
  public static void main(String[] args) {
    Point p=new Point(2,3);
    System.out.printf("%d %d\n,p.x,p.y);

    Point p2=new Point();
    // cannot find symbol constructor Point()
  }
}
```

Constructeur par défaut

- Si aucun constructeur n'est défini, le compilateur rajoute un constructeur **public** sans paramètre

```
public class Point {
  private double x;
  private double y;

  public static void main(String[] args) {
    Point p=new Point(); // ok
  }
}
```

- Contrairement au C++, Il n'est pas nécessaire de mettre obligatoirement un constructeur par défaut.

Appel inter-constructeur

- Appel à un autre constructeur se fait avec la notation **this(...)**

```
public class Counter {
  public Counter(int initialValue) {
    this.counter=initialValue;
  }
  public Counter() {
    this(0);
  }
  public int increment() {
    return counter++;
  }
  private int counter;
}
```

```
public class AnotherClass {
  public static void main(String[] args) {
    Counter c1=new Counter();
    System.out.println(c1.increment());
    Counter c2=new Counter(12);
    System.out.println(c2.increment());
  }
}
```

- **this(...)** doit être la première instruction

Champs final

- Il est possible de déclarer un champs constant (**final**)

```
public class Point {
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public Point() {
  } //variable x might not have been initialized

  private final int x;
  private final int y;
}
```

- Le compilateur vérifie que la valeur est assignée une seule fois et par chaque constructeur

Différents types de méthodes

- Classification des méthodes d'un objet
  - Constructeur
    - Initialise l'objet
  - Accesseur
    - Getter (getX)
      - Export les données (souvent partiellement)
    - Setter (setX)
      - Import les données (en les vérifiants)
  - Méthode métier (business method)
    - Effectue des calculs en fonction des données

Accesseurs

```
public class Point {
  public Point(double x,double y) {
    this.x=x;
    this.y=y;
  }
  public double getX() {
    return x;
  }
  public void setX(double x) {
    this.x=x;
  }
  public double getY() {
    return y;
  }
  public void setY(double y) {
    this.y=y;
  }
  private double x;
  private double y;
}
```

Les accesseurs sont des méthodes permettant l'accès à des attributs

```
Point p=new Point(1,1);
System.out.printf("%f,%f\n",p.getX(),p.getY());
```

Intérêt des accesseurs

Cela permet de changer l'implantation sans changer le code d'appel.

```
Point p=new Point(1,1);
System.out.printf("%f,%f\n"
  ,p.getX(),p.getY());
```

```
public class Point {
  public Point(double x,double y) {
    init(x,y);
  }
  private void init(double x,double y) {
    rho=hypot(x,y);
    theta=atan2(x,y);
  }
  public double getX() {
    return rho*cos(theta);
  }
  public void setX(double x) {
    init(x,getY());
  }
  public double getY() {
    return rho*sin(theta);
  }
  public void setY(double y) {
    init(getX(),y);
  }
  private double rho;
  private double theta;
}
```

Beans ?

- Norme qui permet d'interagir avec des outils extérieurs
- Utilise les accesseurs (get/is,set)
- Exemple :
  Script (PHP, JavaScript, JSP-EL)
  Builder GUI (Swing)
  Base de données (EJB)

Pas d'accesseur partout

- Mettre un accesseur veux dire écrire du code qu'il faudra maitenanir, donc on écrit un accesseur que si nécessaire

- Mettre un set* (mutator) que si l'on est sur que l'objet va changer
- Limiter les objets avec des mutators (cf mutable/inmutable)

Langage Typé

- Un langage (Pascal/C/C++/Java/C#/ML) typé statiquement lorsque l'utilisateur déclare les types.

```
int i=3;
int j=4;

j=i+j;
```

déclaration

utilisation

- Le compilateur vérifie lors du *type-checking* qu'il n'y a pas d'erreur d'utilisation

Langage Dynamique

- Un language dynamique (Shell/Perl/PHP/ Python/SmallTalk/Ruby) ou un script est un language pour lequel lors de l'exécution en fonction des valeurs l'opération a effectué est exécuter

```
i=3;
j=4;

j=i+j;
```

À l'exécution, i et j sont des entiers donc + est le plus des entiers

- Par abus, on parle de langages dynamiquement typé

Rappel sur les types

- Un type est une information indiquée au compilateur
  - Assurant que les méthodes appelées existeront lors de l'exécution
    - « on ne prend pas des vessies pour des lanternes »
  - permet de calculer la taille des objets
  - permet de générer un meilleur code pour l'exécution
- Lors de l'**exécution**, les types n'**existent plus** seules les classes subsistent

Classe et Objet

Il existe plusieurs sortes de langages Objet :

- Les langages à base de *classe* :
  moule à partir duquel on crée des objets.
- Les langages à base de *prototypage* :
  un objet délègue les fonctionnalités qu'il ne comprend pas à son prototype (Javascript)
- Les langages à base de *mixins* :
  un objet possède un type particulier s'il implante toutes les fonctionnalités (OCAML).

Langage à base de Classe

- Une classe définit :
  - les données d'un objet (champs)
  - les fonctionnalités d'un objet (méthodes)

- La création d'un objet se fait par **instanciation** (“démoulage”) d'une classe.
- Les champs de l'objet peuvent alors être initialisées par des valeurs.

Concession au monde objet

- En Java:
  - Les types primitifs ne sont pas des objets
  - On peux déclarer un contexte sans objet (**static**)
- En C# en plus,
  - on peut déclarer des pointeurs de fonctions (*delegate*)

Les Types et Java

- Philosophiquement, Java sépare les types primitifs des types Objets.
  - Les types primitifs :
    Ex: boolean, int, double
  - Les types objet
    Ex: String, int[], Point

- Les types primitifs sont manipulés par leur valeur, les types objets par référence.

Les Types objets

- Les types objets sont définis comme une “composition” de types primitifs et de types objets.

- Les types objets sont
  - soit définis par l'utilisateur
    SeaWhales, Truck
  - soit prédéfinis et existant dans l'API du JDK.
    Date, String, Object

Type ou classe

- Le type est une information pour le compilateur qui l'ui permet de savoir quelles sont les opérations que l'on peut effectuer sur une référence

- La classe est une description d'un objet regroupant:
  le nombre et le type des champs
  le code de chaque méthode

Type ou classe

- Le type est une information pour le compilateur qui l'ui permet de savoir quelles sont les opérations que l'on peut effectuer sur une référence
- La classe est une description d'un objet regroupant:
  - le nombre et le type des champs
  - le code de chaque méthode

Ce sont les informations utilisées par **new** pour créer un objet

Type et classe

- Les variables (locales ou champs) ont un Type
- Les références ont une classe

```
public static void main(String[] args) {

   String s=args[0];

   Whale whale=new Whale(2,3);
}
```

Type

Classe

Méthode

- Les méthodes en Java peuvent être :
  - **strictfp** ou **synchronized**
  - à nombre constant ou variables d'arguments
  - surchargées
    (plusieurs méthodes avec le même nom)

- Il existe de plus un type de méthode particulier qui permet d'initialiser un objet appelé constructeur.

Surcharge de méthodes

- Nommée aussi surdéfinition (*overloading*), consiste à fournir plusieurs définitions pour une méthode

```
public class PrintStream {
  public void println(String text) {
    ...
  }
  public void println(double value) {
    ...
  }
}
```

Le compilateur cherche la meilleure méthode en fonction du type des arguments

```
public static void main(String[] args) {
  PrintStream out=System.out;
  out.println("toto");
  out.println(3.0);
}
```

Surcharge et Typage

- Si aucune méthode ne possède les types exactes, le compilateur prend une méthode approchée en fonction du typage.

```
public class PrintStream {
  public void println(String text) {
    ...
  }
  public void println(double value) {
    ...
  }
}
```

```
public static void main(String[] args) {
  PrintStream out=System.out;
  out.println(3);   // int -> double
  out.println(out); // erreur
}
```

Limitation sur le type de retour

- Le type de retour n'est pas considéré lors d'un appel, il est donc impossible de différencier des méthodes uniquement en fonction de leur type de retour.

```
public class BadOverloading {
  public int f() {
    ...
  }
  public double f() {
    ...
  }
  // f() is already defined in BadOverloading
}
```

```
public static void main(String[] args) {
  BadOverloading bo=new BadOverloading();
  bo.f();
}
```

Quand doit-on surcharger ?

- Règle empirique : on effectue une surcharge entre deux méthodes si celles-ci ont la même sémantique.

```
public class Math {
  public float sqrt(float value) {
  }
  public double sqrt(double value) {
  } // ok
}
```

```
public class List {
  public void remove(Object value) {
  }
  public void remove(int index) {
  } // c'est Mal, pas la même sémantique
}
```

Varargs

- Il est possible de définir des méthodes à nombre variables d'arguments en utilisant la notation ...

```
public class PrintStream {
  public void printf(String text, Object... args) {
    ...
  }
}
```

- Reçoit les arguments dans un tableau

```
  public static void main(String[] args) {
    PrintStream out=System.out;
    out.printf("%d\n",2);
  }
```

Varargs (2)

- La notation ... doit être utilisée que par le dernier argument (comme en C)

```
public static int min(int... array) {
  if (array.length==0)
    throw new IllegalArgumentException("array is empty");
  int min=Integer.MIN_VALUE;
  for(int i:array)
    if (i>min)
      min=i;

  return min;
}
public static void main(String[] args) {
  min(2,3); // tableau contenant deux valeurs
  min(2);   // tableau contenant une valeur
  min();    // tableau vide
  min(new int[]{2}); // utilise le tableau passé
}
```

Varargs et null

- Il y a une ambiguité dans le cas d'un Object... si l'on passe **null**

```
public class PrintStream {
  public void printf(String text, Object... args) {
    ...
  }
}
```

```
public static void main(String[] args) {
  PrintStream out=System.out;
  out.printf("",null); // non-varargs call of varargs method with
                       // inexact argument type for last parameter
}
```

- Le compilateur émet un warning et envoie **null** comme argument

```
  out.printf("",(Object[])null); // enlève l'ambiguité
```

Varargs & compatibilité

- Les varargs sont considérés comme des tableaux par la VM
- Il n'est donc pas possible d'effectuer une surcharge entre tableau et varargs :

```
public class VarargsOverloading {
  private static int min(int... array) { }
  private static int min(int[] array) {
  } // min(int...) is already defined in VarargsOverloading
}
```

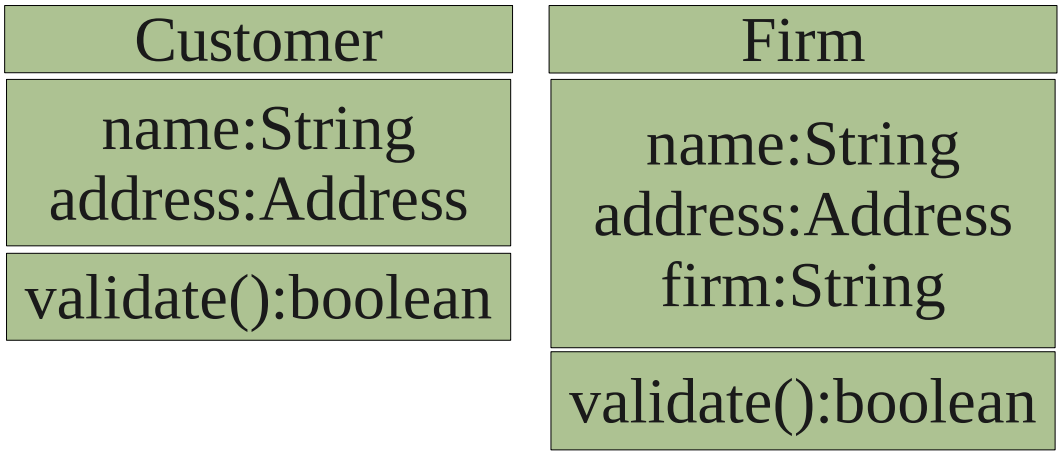
Les Types objet (2)

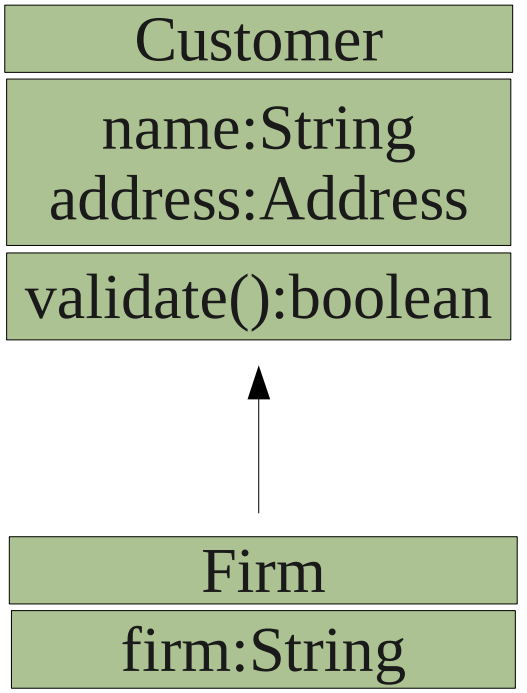
Il existe 4 façons de définir un type objet :

- Définir une classe
- Définir une interface
  (type que l'on ne peut pas instancier)
- Définir une énumération
  (ensemble fini de valeurs,
  pas instanciable non plus)
- Défirnir une annotation
  (sorte interface spéciale)

Héritage

- Exemple

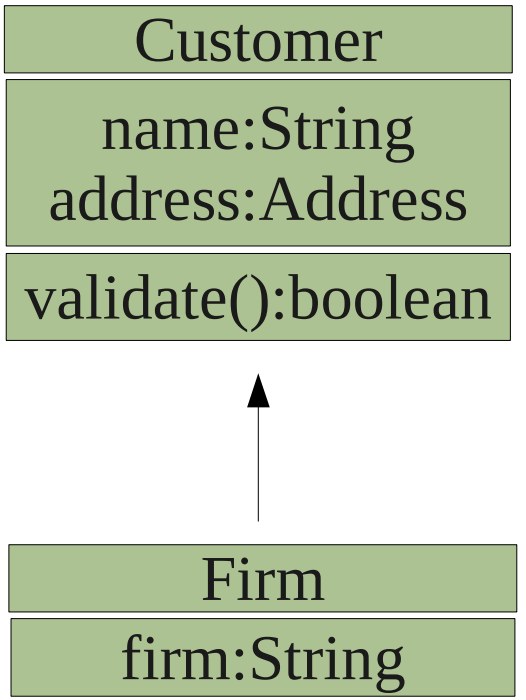




- Actuellement, l'héritage entre classes (concrètes) est relativement rare !!

Définition de l'héritage

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



L'héritage concrètement

- 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;
}
```

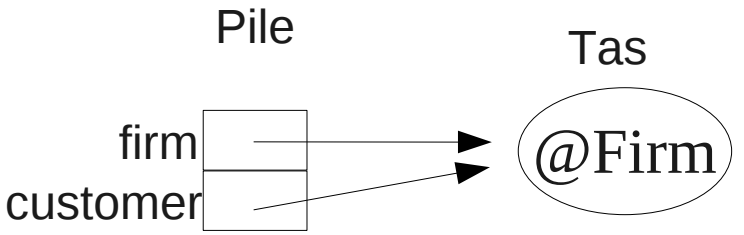
Héritage et Object

- 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 celle-ci hérite de Object (directement ou indirectement)

C'est quoi le sous-typage

- Le sous-typage, c'est le fait de pouvoir considérer une référence à une sous-classe comme étant une référence à une super-classe
- Et, heu, concrètement :

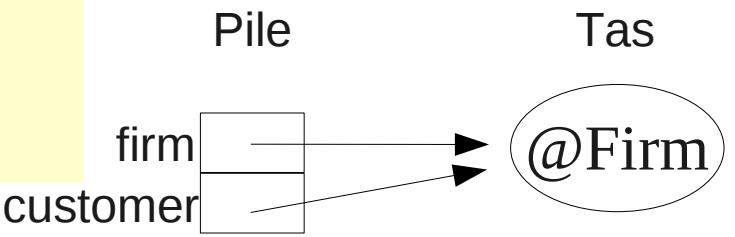
```
public static main(String[] args) {
  Firm firm=new Firm();
  System.out.println(firm);
  Customer customer=firm; // sous-typage
  System.out.println(customer);
}
```



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`**

Sous-Typage

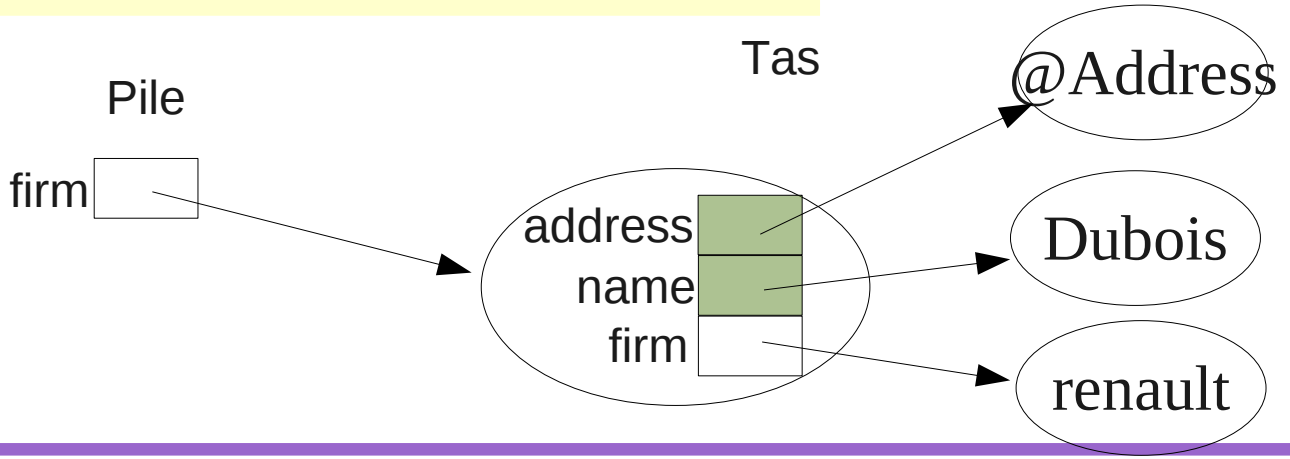
- En Java, tous les types objets sont sous-types de **Object**.
- La relation de soustypage vient de :
  - **L'héritage**, si A hérite de B, alors A est un sous-type de B
  - **L'implémentation d'interface**, si A implante B alors A est un sous-type de B
  - Plus des relations sur les **tableaux**
  - Les **wildcards** sur les **types paramétrés**

Héritage des membres

- Une classe hérite de tous les membres de la super-classe

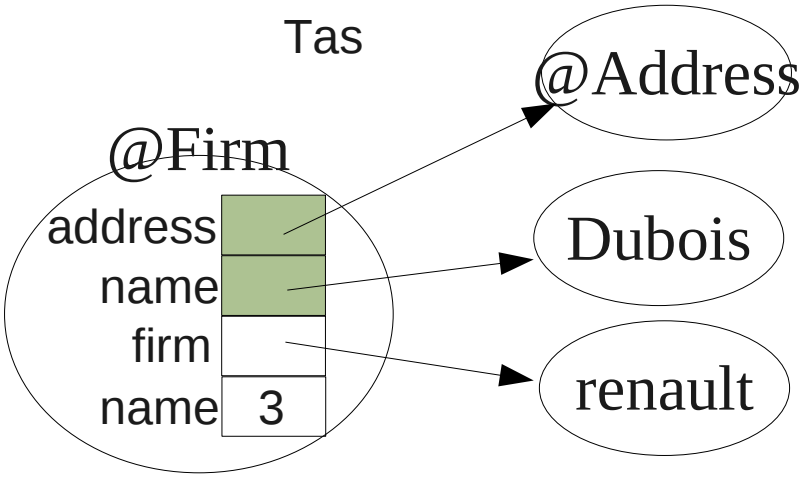
```
public class Firm extends Customer {
  final String firm;

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



Héritage 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;
}
```

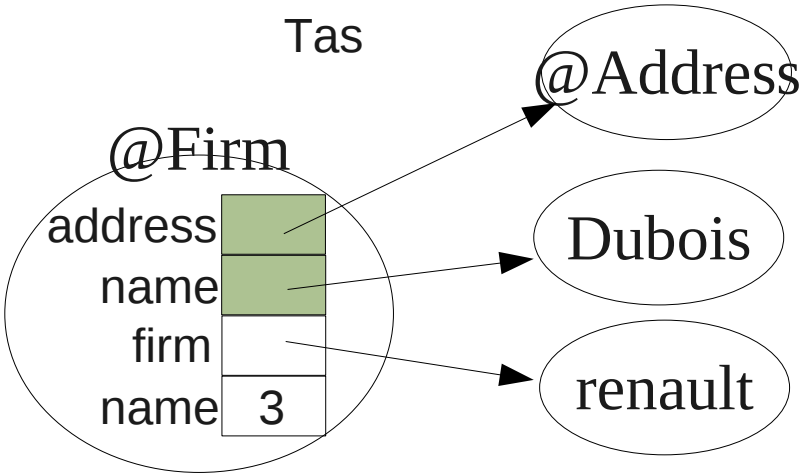
```
public class Customer {
  public String g() {
    return this.name;
  }
  final String name;
  final Address address;
}
```

- Les deux champs cohabitent

Masquage de champs

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

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



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

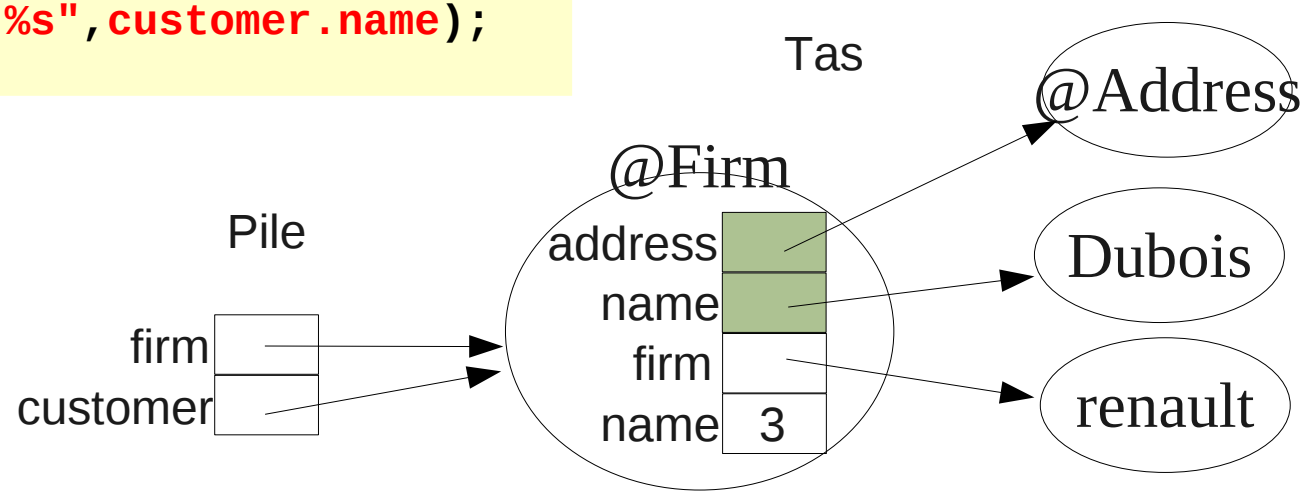
```
public class Customer {
  public String g() {
    return this.name;
  }
  final String name;
  final Address address;
}
```

Champs et Sous-typage

- On accède au champs en fonction du **type** 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

- La première instructeur d'un constructeur et un appel au constructeur de la superclasse

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

  public Firm() {  // implicite
    super();       // bang
  }
} // cannot find symbol  constructor Customer()
```

- Le constructeur implicite fait appel au constructeur par défaut de la superclasse

super(...) et appel implicite

- **super(...)** doit être la première instruction du constructeur

- Si l'instruction **super(...)** n'existe pas, le compilateur ajoute un appel au constructeur par défaut de la classe mêre (**super()**)

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() {
    super(DEFAULT_NAME,DEFAULT_ADDRESS); // erreur
  }
  private final String Address DEFAULT_ADDRESS=null;
  private final static String DEFAULT_NAME=”toto”;
}
```

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

Initialisation des champs

- L'appel au constructeur permet d'initialiser les champs hérités

```
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, c'est que l'**on n'hérite pas des constructeurs**
- Le constructeur de la classe hérité a pour mission de :
  - demander l'initialisation de la classe mêre au travers de son constructeur
  - d'initialiser ses propres champs

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 hérités

Redéfinition ou Surcharge

- Exemple de surcharge et de redéfinition :

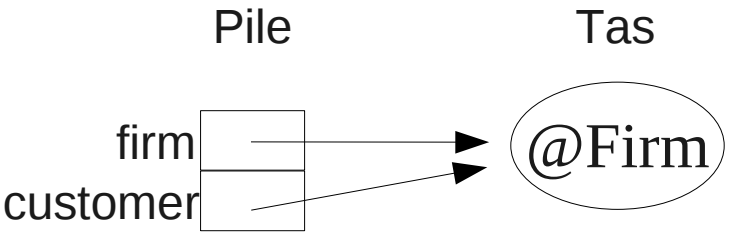
```
public class A {
  public void m(int a) {...}    // surcharge
  public void m(double a) {...} // surcharge
}

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

- **m(double)** de **B** redéfinie **m(double)** de **A**, le reste est de la surcharge

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

Redéfinition et **super.**

- La notation **super.** permet d'avoir accès aux membres non **static** de la superclasse

```
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.

Accès hors de la classe

- Il n'est pas possible d'accèder aux méthodes redéfinies hors de la sous-classe
- Il n'est pas possible d'accéder à la méthode validate de Customer

```
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());
} // cannot find symbol  class customer
```

- **super.** se marche que dans la sous-classe

Méthode **final**

- Si une méthode est déclarée **final**, celle-ci ne peut être redéfinie
  - Important 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
  ...
}
```

- Toutes les méthodes se comportent comme si elles étaient **final**

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'implanter **plusieurs interfaces**

Interface

- Une interface définit un type sans code
- On utlise le mot-clé **interface**

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

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

Instantiation d'interface

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

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

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

Implantation d'interface

- Implanter 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 element);
}
public class FixedSizeList implements List {
  public FixedSizeList(int capacity) {
    array=new Object[capacity];
  }
  public Object get(int index) {
    return array[index];
  }
  public void set(int index,Object element) {
    array[index]=element;
  }
  private final Object[] array;
}
```

Implantation d'interface (2)

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

```
public interface List {
  public Object get(int index);
  public void set(int index,Object element);
}
public class FixedSizeList implements List {
  // FixedSizeList is not abstract and does not override
  // abstract method set(int,java.lang.Object)
  public FixedSizeList(int capacity) {
    array=new Object[capacity];
  }
  public Object get(int index) {
    return array[index];
  }
  private final Object[] array;
}
```

Interface et Sous-typage

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

```
public class AtWork  {
  private void print(List list) {
    for(int i=0;i<list.capacity();i++) {
      System.out.println(list.get(i));
    }
  }
  public static void main(String[] args) {
    List l=new FizedSizeList(5);
    ...
    print(l);
  }
}
```

- Permet d'écrire une méthode **générique**

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 {
}
```

Interface, méthodes et champs

- Les méthodes déclarées dans une interface sont obligatoirement :
  - abstraite (abstract)
  - publique (public)
- Les champs déclarés dans une interface sont obligatoirement :
  - constant (final)
  - public (public)
  - statique (static)

Méthode publique

- Impossible d'implanter une méthode d'une interface avec une visibilité autre que **public**

```
public class FixedSizeList implements List {
  public FixedSizeList(int capacity) {
    array=new Object[capacity];
  }
  // get(int) in FixedSizeList cannot implement
  // get(int) in List; attempting to assign
  // weaker access privileges; was public
  Object get(int index) {
    return array[index];
  }
  ...
  private final Object[] array;
}
```

Interface et constantes

- Eviter pour récuperer des constantes d'implanter une interface

```
public interface WheelConstant {
  int FRONT_WHEEL=0;
  int BACK_WHEEL=1;
}
public class Bicycle implements WheelConstant {
  Bicycle(Object... values) {
    front=(Wheel)values[FRONT_WHEEL];
    back=(Wheel)values[BACK_WHEEL];
  }
  ...
```

- Ici, on déclarera plutôt une énumération (cf plus loin)

Interface et static

- Il n'est **pas** possible de définir une **méthode statique** dans une interface
  (James Gosling trouve ça pas beau !)

- Vous pouvez voter pour changer les choses

@Override

- En 1.5, @**Override** indique que l'on redéfinie une méthode, marche pas pour les méthodes d'interface
- En 1.6, @**Override** pour dire qu'une méthode implante une méthode définie par l'interface.

```
public class Bicycle implements Managed {
  @Override public void init(Object... values) {
    ...
  } // 1.6 ok
    // 1.5: method does not override a method from
    // its superclass
```

Design: Interface ou héritage

- Hériter d'une classe :
  - La sous-classe est “une sorte” de classe de base
- Implanter une interface pour définir :
  - une fonctionalité transversale (Comparable, Closable)
  - Ou un ensemble de fonctionnalités où un objet implantant plusieurs est posssible (DataInput, DataOutput)
- Appel à une méthode d'une interface plus lent !

Classe abstraite

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

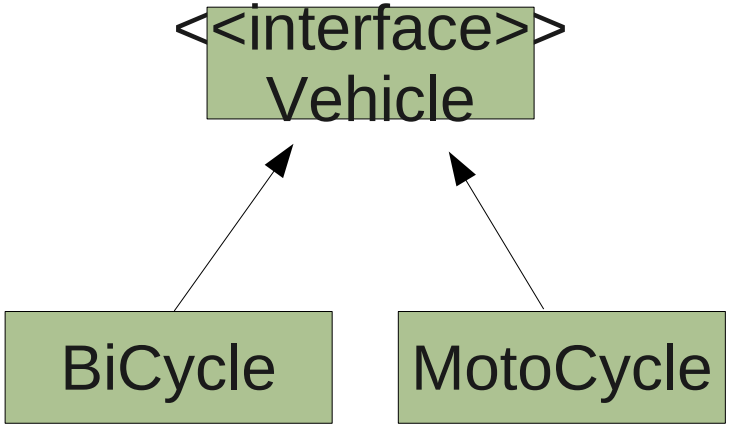
```
public interface List {
  public boolean isEmpty();
  public int size();
}

public abstract class AbstractList implements List {
  public boolean isEmpty() {
    return size()==0;
  }
}
```

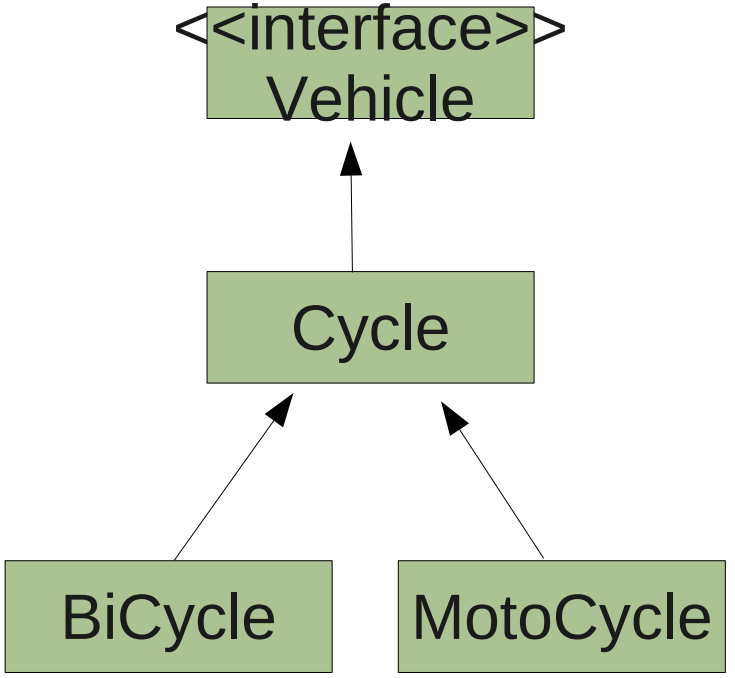
- Une classe abstraite est une classe partiellement implantée donc non instantiable

Raffinement de l'abstraction

- Une classe abstraite peut s'intercaler dans l'arbre d'héritage entre l'interface et les classes concrètes



AVANT



APRES

Intérêt des classe abstraite

- Permet de partager du code commun à des sous-classes

```
public abstract class Cycle implements Vehicle {
  public void backflip() {
    blockBrake(front);
  }
  protected abstract void blockBrake(Wheel wheel);
  private Wheel front,back;
}
```

- Le code commun peut supposer la présence de certaine méthodes (donc abstraites)

La classe Objet

- Toutes les classes héritent de **java.lang.Object** directement ou indirectement.
- Un objet en Java possèdent donc par héritage toute les méthodes définies dans la classe **Object**

```
public class ClassExample {
  public static void main(String[] args) {
    Truc truc=new Truc();
    Object o=truc;
  }
}
```

- Toute classe est un **sous-type** de **Object**

Classe et Objet

- La méthode **getClass()** sur un **Object** permet d'obtenir un objet de type **Class** représentant la classe de l'objet

```
public class ClassExample {
  public static void main(String[] args) {
    String s="toto";
    Object o="tutu";                    // sous-typage
    System.out.println(o.getClass()); // java.lang.String
    boolean test=s.getClass()==o.getClass();
    System.out.println(test);           // true
  }
}
```

- L'object **Class** permet de rechercher les membres d'un objet par leur nom

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=args[0];
    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**

Assurer le transtypage

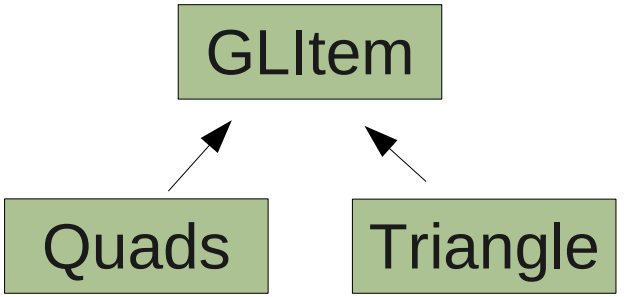
- Il est possible d'assurer un transtypage sans lever d'exception en utilisant **instanceof**

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

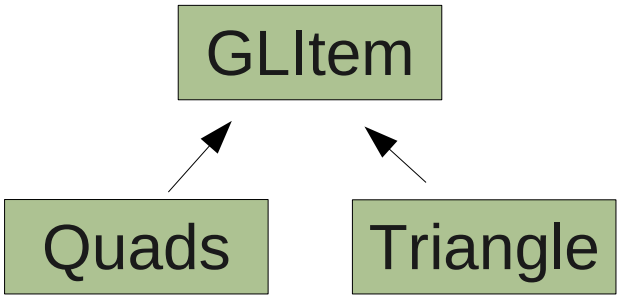



```
public static void renderLoop(GLItems[] items) {
  glBegin();
  for(GLItem item:items) {
    if (item instanceof Quads)
      renderQuads((Quads)item);
    else
      renderTriangle((Triangle)item);
  }
  glEnd();
}
```

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

Utiliser le polymorphisme !

- Item definie 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

Note sur le transtypage

- Utilisation classique du transtypage :
  Les containers générique utilisent **Object**

```
public static void main(String[] args) {
  ArrayList list=new ArrayList();
  list.add("toto");
  String s=(String)list.get();
}
```

- Le transtypage n'est pas necessaire en 1.5

```
public static void main(String[] args) {
  ArrayList<String> list=new ArrayList<String>();
  list.add("toto");
  String s=list.get();
}
```