

Développement Logiciel

L2-S4

Rappels

anastasia.bezerianos@lri.fr

Les transparents qui suivent sont inspirés du cours de Rémi Forax (Univ. Marne la Vallée)
(transparents utilisés avec son autorisation)

Plan

- Rappels et approfondissement
 - Méthodologie: un objet = une responsabilité
 - Encapsulation (et visibilité), p.20
 - Les constructeurs, p.27
 - Les accesseurs (get/set), p.33
 - Classe et objet, p.37
 - Notion d'héritage, méthodes et constructeurs, p.50
 - Redéfinition, surcharge, p.67
 - Interfaces, classes abstraites, p.75
 - Classe objet, transtypage, p.86

Rappels et approfondissement

POO - Objet

- Tous les langages de programmation fournissent des abstractions
- POO - outils pour représenter des éléments dans l'espace problème
- ... objets



POO - Objet (I)

I. Toute chose est un objet.

Un objet est une variable amélioré : il stocke des données, on peut effectuer des requêtes sur cet objet, lui demander de faire des opérations sur lui-même.

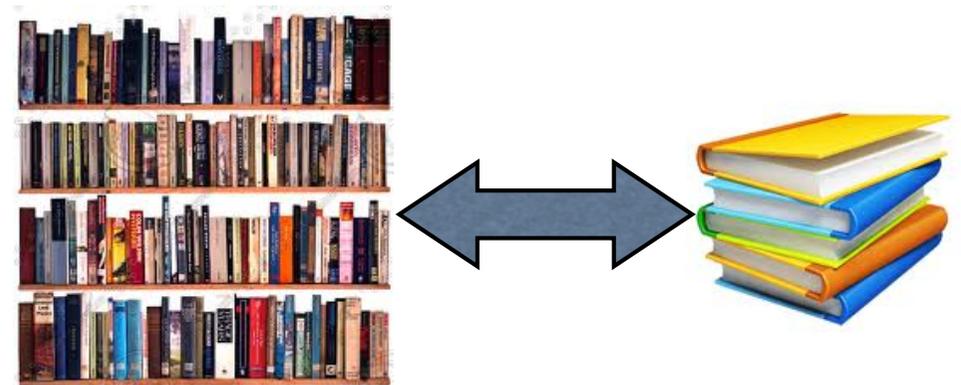
On peut prendre n'importe quel composant conceptuel du problème qu'on essaye de résoudre (un chien, un immeuble, un service administratif, etc...) et le représenter en tant qu'objet dans le programme.



POO - Objet (2)

2. Un programme est un ensemble d'objets se disant les uns aux autres quoi faire en s'envoyant des messages.

Pour qu'un objet effectue une requête, on envoie un message à cet objet. Plus concrètement, on peut penser à un message comme à un appel de fonction appartenant à un objet particulier.



POO - Objet (3)

3. Chaque objet a son propre espace de mémoire composé d'autres objets.

On crée un nouveau type d'objet en créant un paquetage contenant des objets déjà existants. Ainsi, la complexité d'un programme est cachée par la simplicité des objets mis en oeuvre.

POO - Objet (4)

4. Chaque objet est d'un type précis.

Dans le jargon de la POO, chaque objet est une instance d'une classe (type, patron). La plus importante caractéristique distinctive d'une classe est :

Quels messages peut-on lui envoyer ?

POO - Objet (5)

5. Tous les objets d'un type particulier peuvent recevoir le même message.

Un objet de type “cercle” est aussi un objet de type “forme géométrique”, un cercle doit accepter les messages destins aux formes géométriques.



Classe et Objet

- Des objets semblables sont groupés ensemble dans une **classe d'objets**.
- Avec le mot clef **Class** on peut créer des types de données abstraits, c.à.d. des classes (concept fondamental dans la POO).
- On utilise les types de données abstraits exactement de la même manière que les types de données prédéfinis.
 - On peut créer des variables d'un type particulier (objets/instances) et les manipuler (envoyer des messages/requêtes : on envoie un message et l'objet se débrouille pour le traiter).
 - Les membres (objets) d'une classe partagent des caractéristiques communes : chaque voiture dispose d'une couleur, une marque, etc...
 - Cependant, chaque objet a son propre état : chaque voiture a une couleur différente et une marque différente, etc.

Membres d'un objet

- Membres d'un objet :
 - des champs qui définissent ses données (caractéristiques)
 - des méthodes qui manipulent les données (traiter les messages)

```
class oneClass{  
    void oneMethod (Point p){  
        System.out.println(p.x);  
        System.out.println(p.y);  
        p.translate(2,3);  
    }  
}
```

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

Accès aux Membres

- L'accès aux membres d'une classe (champs & méthodes) se fait par la notation "." sur une référence à un objet.

```
class oneClass{
    void oneMethod (Point p){
        System.out.println(p.x);
        System.out.println(p.y);
        p.translate(2,3);
    }
}
```

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

Notation Objet

- Par rapport à une fonction classique, on privilégie un des paramètres qui devient l'objet sur lequel on appelle la méthode.
- C: `equals(o1, o2)`
- java: `o1.equals(o2)`

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

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

this

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

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

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

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

Principes de la POO

- un objet, une responsabilité
- encapsulation
- immutabilité
- type et classe
- découplage interface / implantation
- abstraction et concept
- principe de localité et polymorphisme
- objet / pas objet

un objet, une responsabilité

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

un objet, une responsabilité

exemple (I)

- 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.

- responsabilités?



un objet, une responsabilité

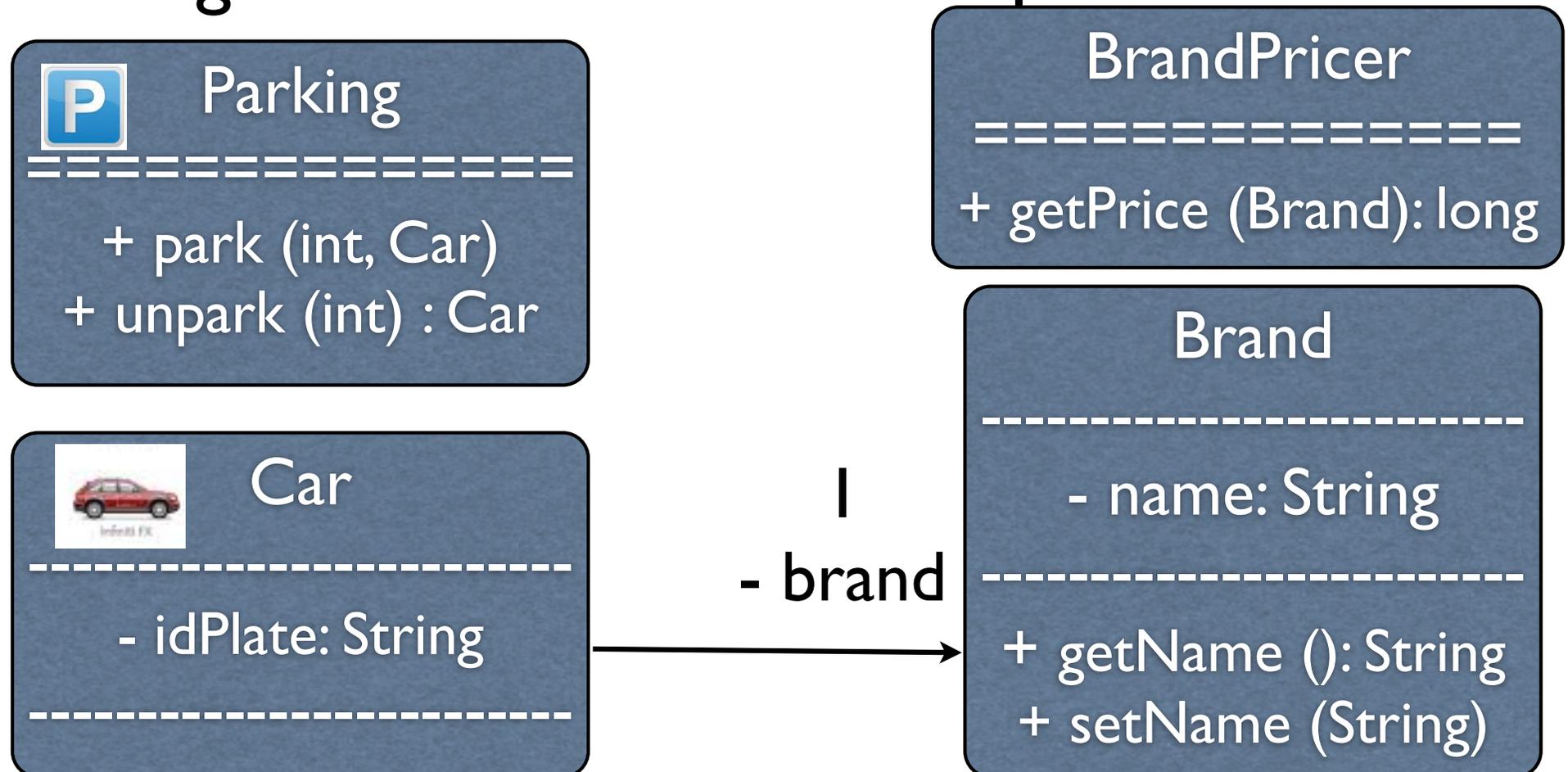
exemple (2)

- Ici, 4 responsabilités, 4 objets
 - Parking : stocker les voitures
 - Car : la voiture elle même
 - Brand : le modèle de voiture
 - BrandPricing : indique le prix en fonction d'un modèle.



un objet, une responsabilité exemple (3)

- Diagramme de classe correspondant

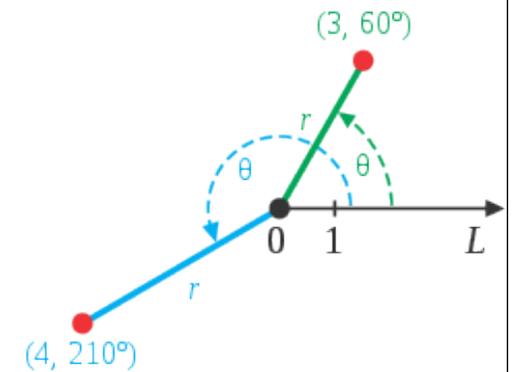


encapsulation

- Un principe qui garantie qu'aucun champ d'un objet ne pourra être modifié pour corrompre l'état d'un objet sans une vérification préalable.
- L'état ne pourra être modifié (s'il est modifiable) que par les méthodes vérifiant les données rentrées.

encapsulation

- Exemple d'un point en coordonnée polaire



```
class oneClass{  
    void oneMethod (Point p){  
        p.init(2.0,3.0);  
        System.out.println(p.theta);  
  
        p.rho = -1; // aie aie !!  
    }  
}
```

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

- La classe Point ne garantie pas l'encapsulation

visibilité des membres

- Certain langage comme Smalltalk ne permette pas l'accès au champs
- D'autre comme C++, Java demande à ce qu'on écrive des modificateurs de visibilité
- La visibilité permet d'assurer l'encapsulation en interdisant l'accès aux champs directement

Modificateur de visibilité

- Il existe 4 modificateurs de visibilité :
 - ▶ un membre **private**, n'est visible qu'à l'intérieur de la classe.
 - ▶ un membre **sans modificateur** est visible par toutes les classes du même package.
 - ▶ un membre **protected** est visible par les classes héritées et celles du même package.
 - ▶ un membre **public** est visible par tout le monde.
- Il existe un ordre de visibilité
private < protected < public

visibilité et maintenance

- Quelques règles :
 - ▶ un champ n'est jamais **protected** ou **public** (sauf les constants)
 - ▶ la visibilité de paquetage est utilisée si une classe partage des détails d'implantation avec une autre (ex. des classes internes)
 - ▶ une méthode n'est **public** que si nécessaire
 - ▶ une méthode **protected** permet d'implanter un service commun pour les sous classes, si celui-ci ne peut être écrit hors de la classe avec une visibilité de paquetage.

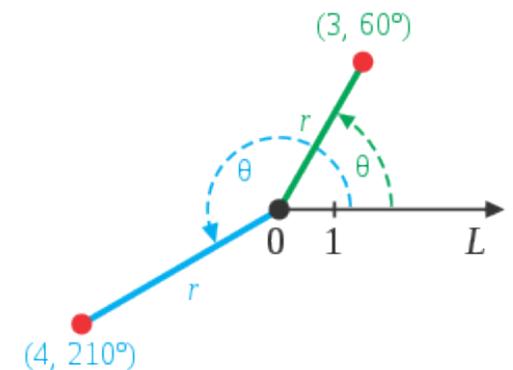
visibilité et maintenance

- Un champs est **toujours** privé !!

```
class Point{
    private double rho;
    private double theta;

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

```
class oneClass{
    void oneMethod (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

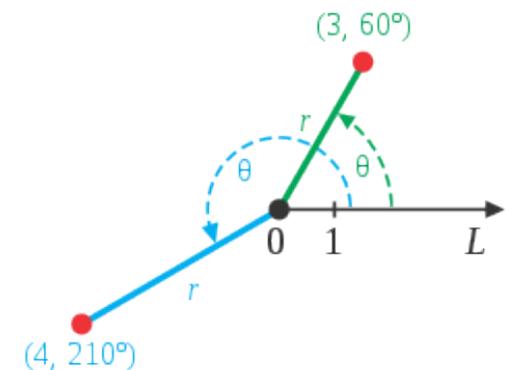
- Supposons que l'on souhaite que rho ne soit jamais égal à zéro
- On peut obtenir la valeur des champs avant l'initialisation

```
class Point{
    private double rho;
    private double theta;

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

class oneClass{
    void oneMethod (Point p){
        System.out.println(p.toString());
        // affiche 0,0 oups

        p.init (2.0,3.0);
    }
}
```



Constructeur

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

```
class Point{
    private double rho;
    private double theta;

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

Constructeur (2)

- 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 et la mémoire sera réservée pour initialiser l'objet.

Constructeur (3)

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

```
class Point{
    double x; // pas private !
    double y; // pas private !

    public Point (double x, double y){
        this.x = x;
        this.y = y;
        ...
    }
    ...
}

class anotherClass{
    public static void main (String[] args){
        Point p = new Point (2.0,3.0);
        System.out.println("%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

```
class Point{
    double x;
    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;
}
```

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

- **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(){
    } // variables x,y may have not 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 vérifiant)
- **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/champs

```
Point p = new Point (1.0,1.0);
System.out.println("%d %d\n",
                    p.getX(), p.getY() );
```

Intérêt des Accesseurs

```
public class Point{
    private double rho;
    private double theta;

    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 setX (double y){
        init(getX(),y);
    }
}
```

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

```
Point p = new Point (1.0,1.0);
System.out.println("%d %d\n",
                    p.getX(), p.getY() );
```

Pas d'accessesseur partout

- Mettre un accessesseur veut dire écrire du code qu'il faudra maintenir, donc on écrit un accessesseur 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 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 être initialisées par des valeurs.

Concession au monde objet

- En Java :
 - les types primitifs ne sont pas des objets
 - On peut déclarer un contexte sans objet (**static**)

```
public class Hello{  
  
    static Clock clock = new Clock();  
  
    public static void main (String args[]){  
        clock.sayTime();  
    }  
}
```

Les Types en Java

- 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 en Java

- Le type est une information pour le compilateur qui lui 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

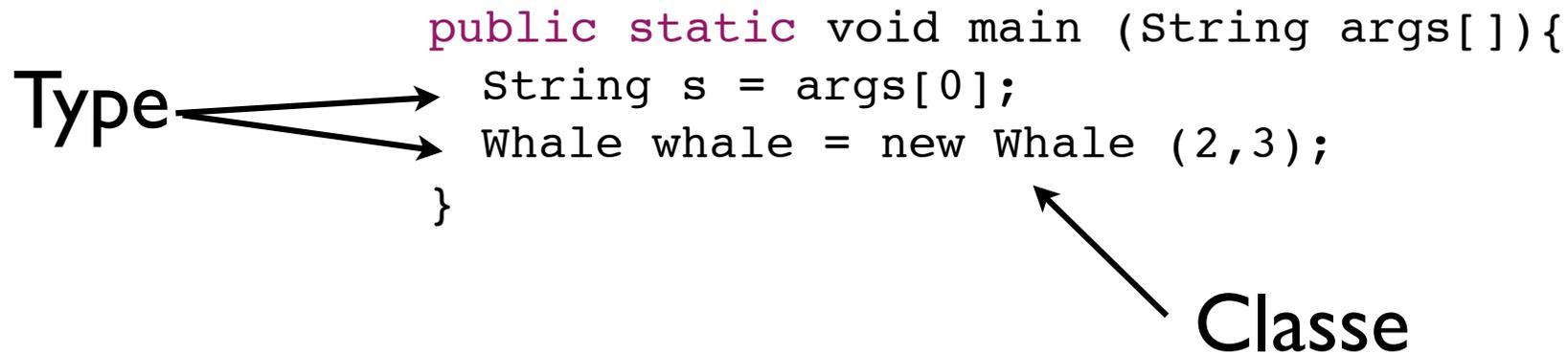
Types et classe

- 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



Méthodes

- 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);  
    out.println(2);  
}
```

Surcharge et Typage

- Le typage 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 “...”
- Reçoit les arguments dans un tableau

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

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 ambiguïté dans le cas d'un Objet ... 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'ambiguïté
```

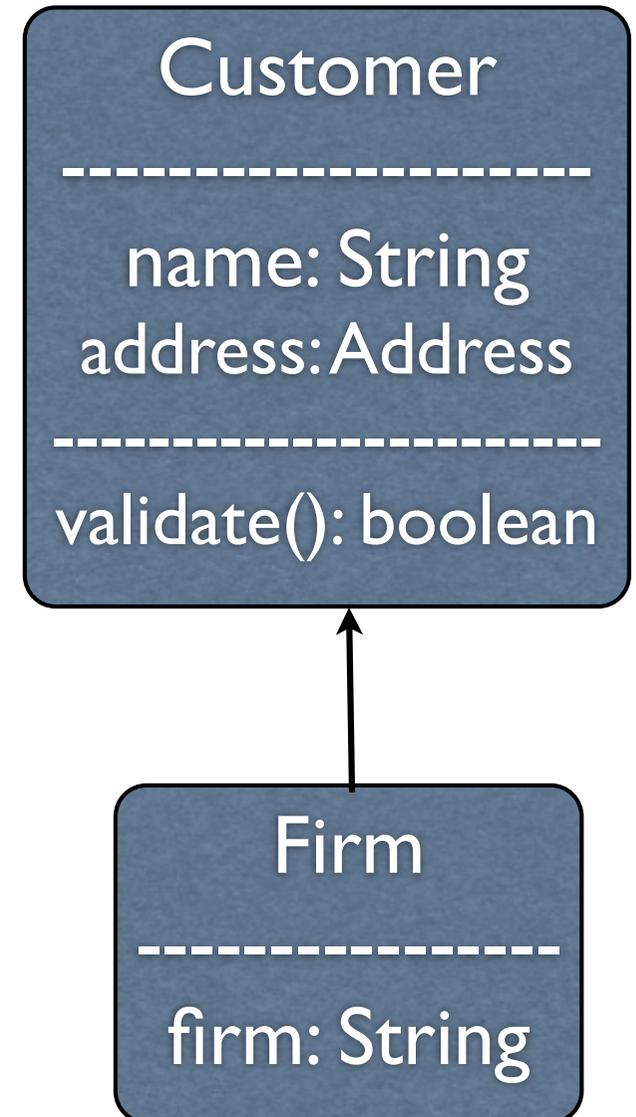
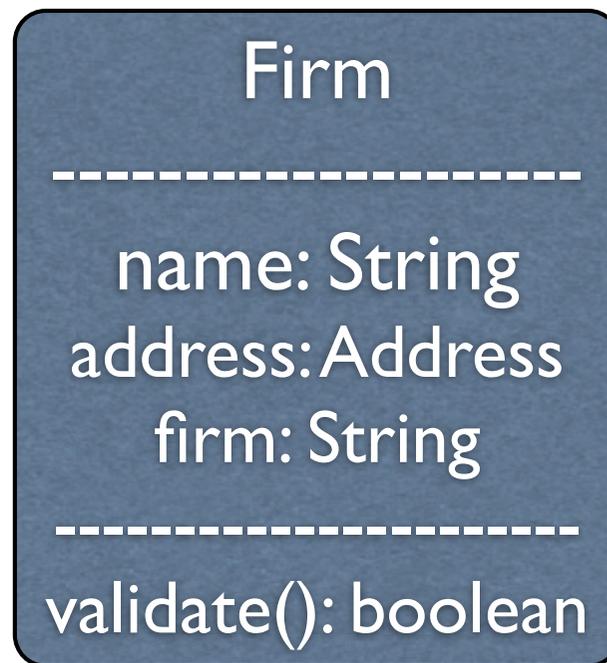
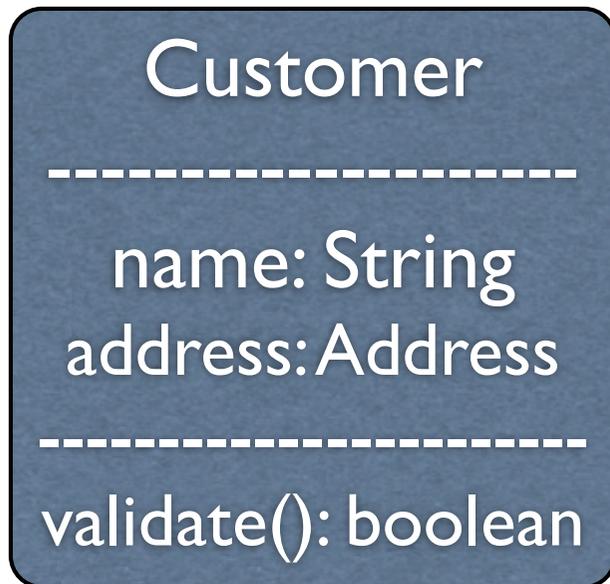
Varargs et compatibilité

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

Héritage

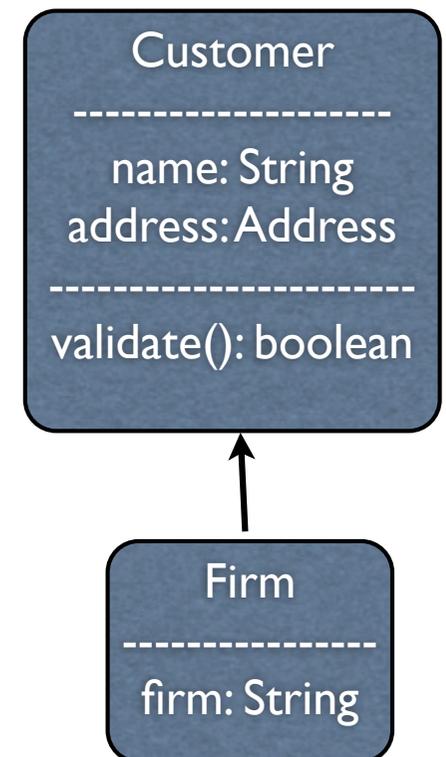
- Exemple



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

Héritage

- L'héritage, c'est 3 choses:
 - Récupération des membres (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;  
    }
```

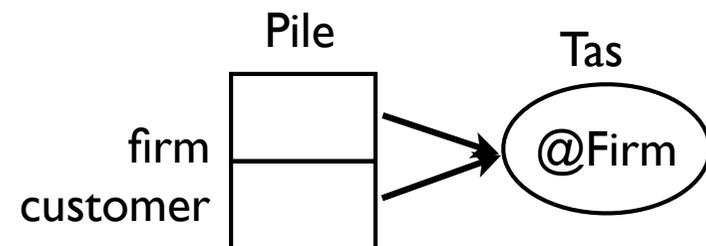
Héritage et **Object**

- 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
- Concrètement

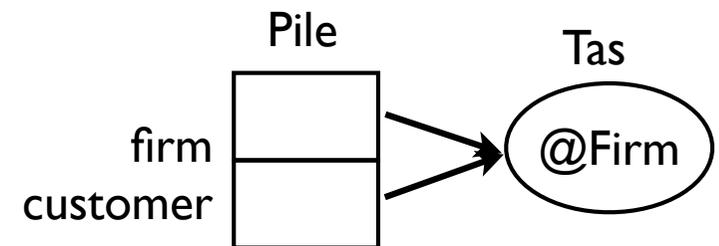
```
public static main (String[] args){  
    Firm firm = new Firm();  
    System.out.println(firm);  
    Customer customer = firm; // sous-typage  
    System.out.println(customer);  
}
```



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 où l'on attend un objet de type **Customer**, il est possible de donner un objet de type **Firm**

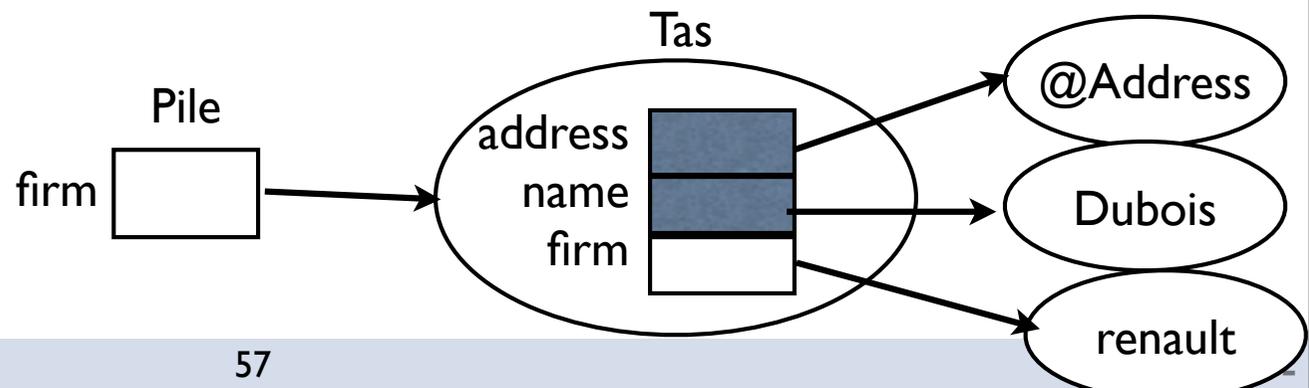
Sous-typage

- En java, tous les types objets sont sous-types de **Object**
- La relation de sous-typages vient de :
 - L'**héritage**, si A hérite de B, alors A est un sous-type de B, etc.
 - L'**implémentation d'interface**, si A implémente B alors A est un sous-type de B
 - Relations sur **les tableaux**, si A est un sous-type de B, alors A[] est un sous-type de B[]
 - Les **wildcards** sur les **types paramétrés**

Héritage des membres

- Une classe hérite 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)  
    }  
}
```

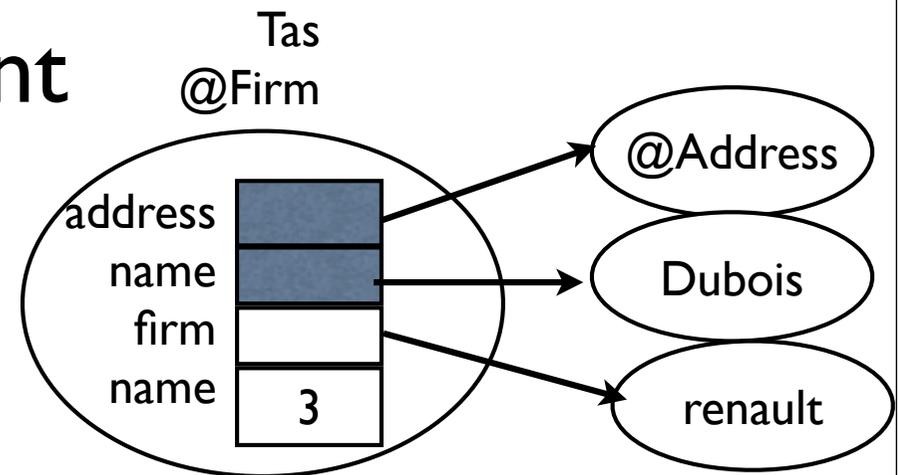


Héritage des champs

- Il est possible de nommer un champs de façon identique dans une sous-class
- Les deux champs cohabitent

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

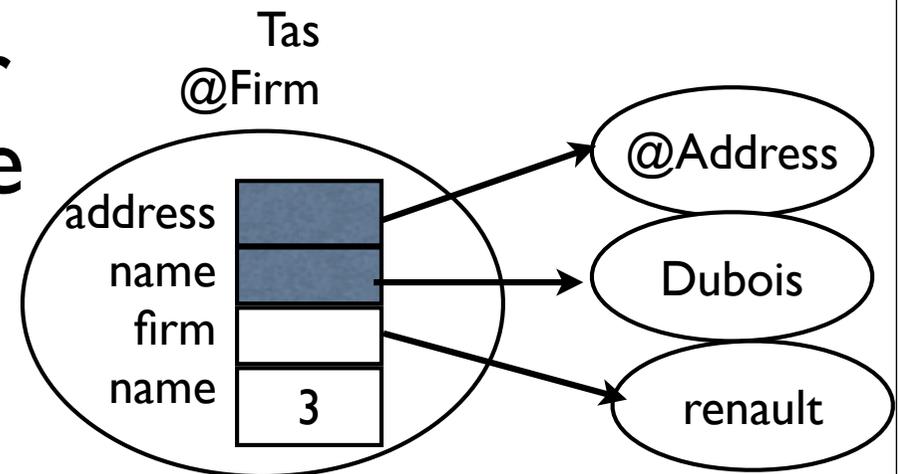
```
public class Customer {  
    final Address address;  
    final String name;  
    public String g () {  
        return this.name;  
    }  
}
```



Masquage de champs

- On dit que le champs *name* de **Firm** masque le champs *name* de **Customer**
- **super.** permet d'accéder au champs de la super classe

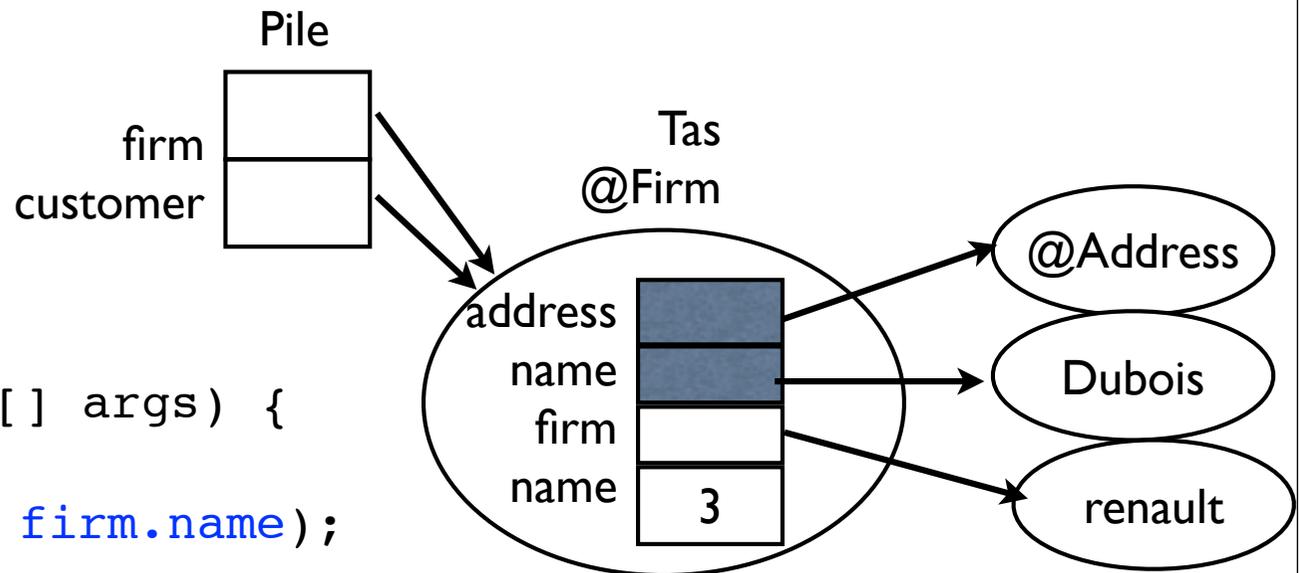
```
public class Firm extends Customer {  
    final String firm;  
    final int name;  
    public String f () {  
        return super.name;  
    }  
}
```



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

Champs et Sous-typage

- On accède au champs en fonction de **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 instruction d'un constructeur est un appel au constructeur de la super-classe

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

- Le constructeur implicite fait appel au constructeur par défaut de la super-classe

Initialisation des champs

- L'appel au constructeur permet d'initialiser les champs hérités
- Notez que les champs sont **private**

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

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

Constructeur vs Méthode

- La grosse différence entre un constructeur et une méthode, c'est que l'**on 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

Héritage de méthode

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

```
public class Firm extends Customer {
    private final String firm;
    public Firm (String firm, String name, Address address) {
        super(name, address);
        this.firm = 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 vérifie pas le champ **firm**

Redéfinition de méthode

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

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

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

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 la classe mère et une classe héritée

Redéfinition ou Surcharge

- Exemple de surcharge et de redéfinition

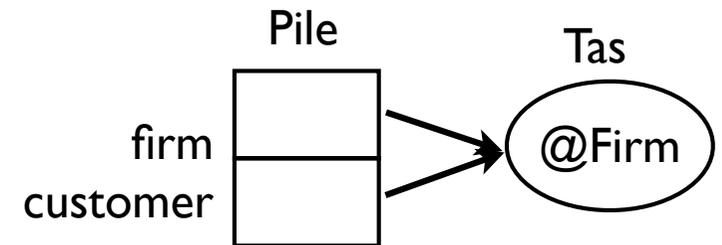
```
public class A {  
    public void m (int a) {...}    // surcharge  
    public void m (double a) {...} // surcharge  
}
```

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

- **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 de **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 super-classe

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

- **super.super.validate()** ne marche pas

Accès hors de la classe

- Il n'est pas possible d'accéder 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.** ne marche que dans la sous-classe

Méthode **finale**

- Si une méthode est déclarée **final**, celle-ci ne peut pas être redéfinie
 - Important en terme de sécurité
 - Peu, voir 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){
        return true; // illegal
    }
}
```

Classe **finale**

- 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{// illegal  
}
```

- 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 utilise 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 définit 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(); // illegal  
    }  
}
```

Implantation d'interface

- Implanter une interface consiste à déclarer une 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 the  
    // 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;  
}
```

Interfaces 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 FixedSizeList(5);
        ...
        print(l);
    }
}
```

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)
 - publique (public)
 - statique (static)

Design: Interface ou héritage

- Hériter d'une classe :
 - La sous-classe est “une sorte” de classe de base
- Implanter une interface pour définir :
 - une fonctionnalité transversale
 - ou un ensemble de fonctionnalités où un objet implantant plusieurs est possible
- Appel à une méthode d'une interface plus lent !

Classe abstraite

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

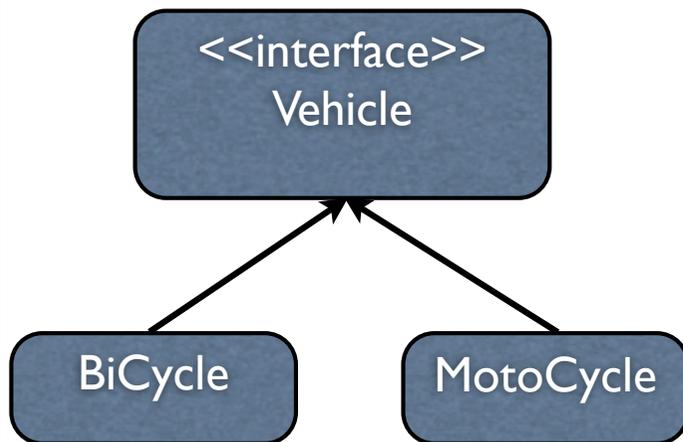
```
public interface List{  
    public boolean isEmpty();  
    public int size();  
}
```

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

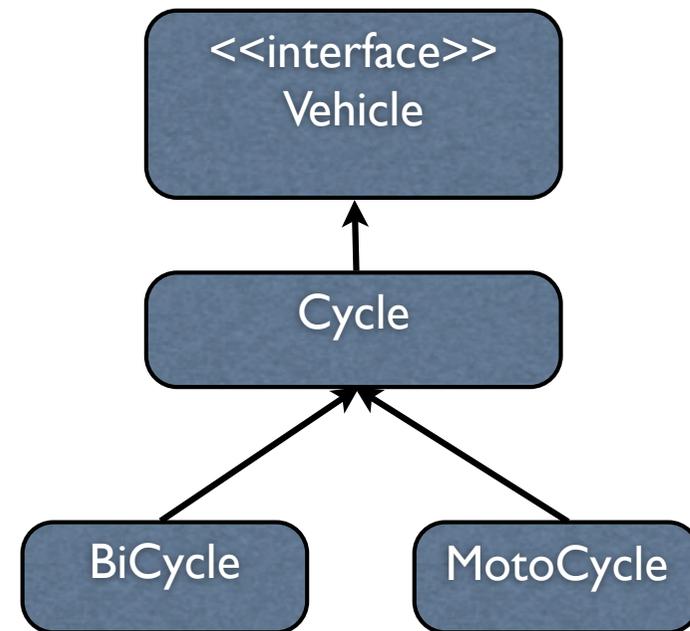
- Une classe abstraite est une classe partiellement implantée donc non instantiable

Raffinement de l'abstraction

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



Avant



Après

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 Object

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

```
public class Truc {  
    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";  
        System.out.println(o.getClass());           // java.lang.String  
        boolean test1= (s.getClass()==o.getClass());  
        System.out.println(test1);                 // false  
    }  
}
```

- L'objet **Class** (obtenu par **getClass()**) 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.lenght!=0)
            o=args[0];
        else
            o=new Object();
        boolean test=o instanceof String;
        System.out.println(o); // return true or 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.lenght!=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.lenght!=0)  
            o="tutu";  
        else  
            o=new Object();  
  
        if (o instanceof String){  
            String s=(String)o;  
            ...  
        }  
    }  
}
```

Note sur le transtypage

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

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

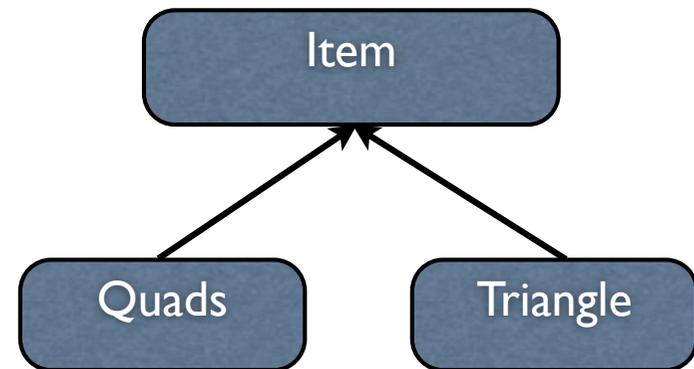
- Le transtypage n'est pas nécessaire en 1.5+

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

Utiliser le polymorphisme

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

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



- Pour utiliser le polymorphisme, il faut avoir accès au code des classes

La classe **Object**

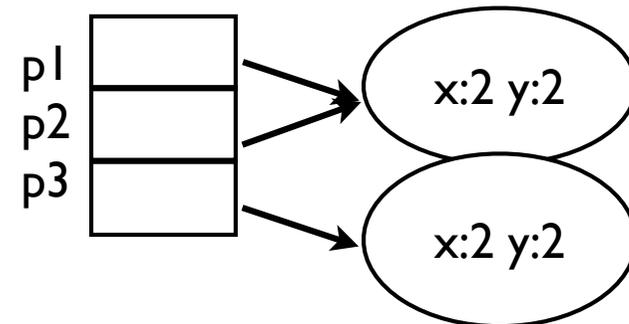
- Classe mère de toutes les classes.
- Possède des méthodes de base qu'il est possible de redéfinir:
 - `toString()` & `hashCode()`
 - `equals()`
 - `getClass()`
 - `clone()`
 - `finalize()`

Tests d'égalité

- Les opérateurs de comparaison `==` et `!=` testent les valeurs des variables:
 - Pour les types primitifs, on test leur valeurs
 - Pour les types objets, on test les valeurs de leur références

```
Point p1;  
p1=new Point(2,2);  
Point p2;  
p2=p1;  
Point p3;  
p3=new Point(2,2);  
p1==p2; //true  
p1==p3; //false  
p2==p3; //false
```

Variables locales



La méthode equals()

- Il existe déjà une méthode equals(Object) dans Objet
- Mais son implantation test les références

```
Point p1=new Point(2,2);  
Point p3=new Point(2,2);  
p1==p3;           //false  
p1.equals(p3);   //false
```

- Pour comparer structurellement deux objets, il faut changer (on dit **redéfinir**) le code de la méthode equals()

Clonage

- Permet de dupliquer un objet
- Mécanisme pas super simple à comprendre:
 - `clone()` a une visibilité **protected**
 - `clone()` peut lever une exception **CloneNotSupportedException**
 - **Object.clone()** fait par défaut une copie de surface si l'objet implante l'interface **Cloneable**
 - L'interface **Cloneable** ne **définie pas** la méthode `clone()`, c'est fait dans **Object**

Clonage (2)

- Cloner un objet sans champs contenant un objet mutable

```
public class MyInteger implements Cloneable { // nécessaire
    private final int value;
    public MyInteger(int value) { // pour Object.clone()
        this.value=value;
    }
    public @Override MyInteger clone(){
        // return new MyInteger(value); //Mal si héritage
        return (MyInteger)super.clone();//shallow copy, mais cast MyIntege
    }
    public static void main(String[] args){
        MyInteger i=new MyInteger(3);
        MyInteger j=i.clone();
        System.out.prntln(i==j);           //false
        System.out.prntln(i.equals(j)); //false
    }
}
```

Clonage (3)

- Cloner un objet avec des champs contenant des objets mutables

```
public class MyHolder implements Cloneable { // nécessaire
    private final Mutable mutable;

    public MyHolder(Mutable mutable) { // pour Object.clone()
        this.mutable=mutable;
    }

    public @Override MyHolder clone(){
        MyHolder holder = (MyHolder)super.clone();//shallow copy
        holder.mutable = this.mutable.clone(); //clone le mutable
        return holder;
    }
}
```

- On appel **clone()** sur l'objet mutable

void finalize()

- Méthode testamentaire qui est appelé juste avant que l'objet soit réclamé par le GC (*garbage collection*)

```
public class FinalizedObject {
    protected @Override void finalize(){
        System.out.println("ah, je meurs");
    }
    public static void main(String[] args){
        FinalizedObject o=new FinalizedObject();
        o=null;
        System.gc(); // affiche ah, je meurs
    }
}
```

- Cette méthode est **protected** et peut lever un **Throwable** (exception)