

Classes abstraites

Programmation objet - III

L2 informatique

F. Bertrand

Au programme d'aujourd'hui...

- Héritage et polymorphisme paramétrique
- Classes et méthodes abstraites
- La comparaison d'objets revue avec l'héritage
- Redéfinition vs surcharge de méthodes

Héritage et polymorphisme paramétrique

- Un problème fréquemment rencontré avec les conteneurs est le cas où, lorsque B est un sous-type de A, il est possible d'écrire :

```
A unA = new B();
```

- Cependant l'écriture du code suivant est impossible :

```
List<A> listeA = new ArrayList<B>();
```

- Pourquoi ?... Autoriser cette affectation permettrait d'insérer dans la liste d'éléments B des éléments sous-type de A (ex. C sous-type de A) ou même des éléments de type A.

Héritage et polymorphisme paramétrique (suite)

- Exemple :

```
class A { }
```

```
class B extends A { }
```

```
class C extends A { }
```

```
class Test {  
    public static void main(String[] args) {  
        ArrayList<A> listeA = new ArrayList<B>();  
        listeA.add(new C());  
    }  
}
```

Cette affectation déclenche une erreur à la compilation !...

... mais si elle était autorisée alors on pourrait écrire ce code → incohérence car listeA référence une liste d'objets B dans laquelle on met un objet... C !...

- Pour permettre une telle affectation une notation existe...

Héritage et polymorphisme paramétrique (suite)

- `List` : définit une liste d'objets pouvant être de types différents (similaire à `List<Object>`)
- `List<T>` : définit une liste d'objets devant être tous du type `T`
- `List<? extends T>` : ajoute une borne **supérieure** au type des objets de la liste qui doit être `T` ou un **sous-type** de `T`
`List<? extends Number> li = new ArrayList<Integer>();`
car `Integer` est un sous-type de `Number`
- C'est cette notation qu'il faut utiliser lorsqu'on veut passer une liste d'un sous-type de celui attendu.

Héritage et polymorphisme paramétrique (suite)

- Cependant la nouvelle référence ne permet pas la **modification** :

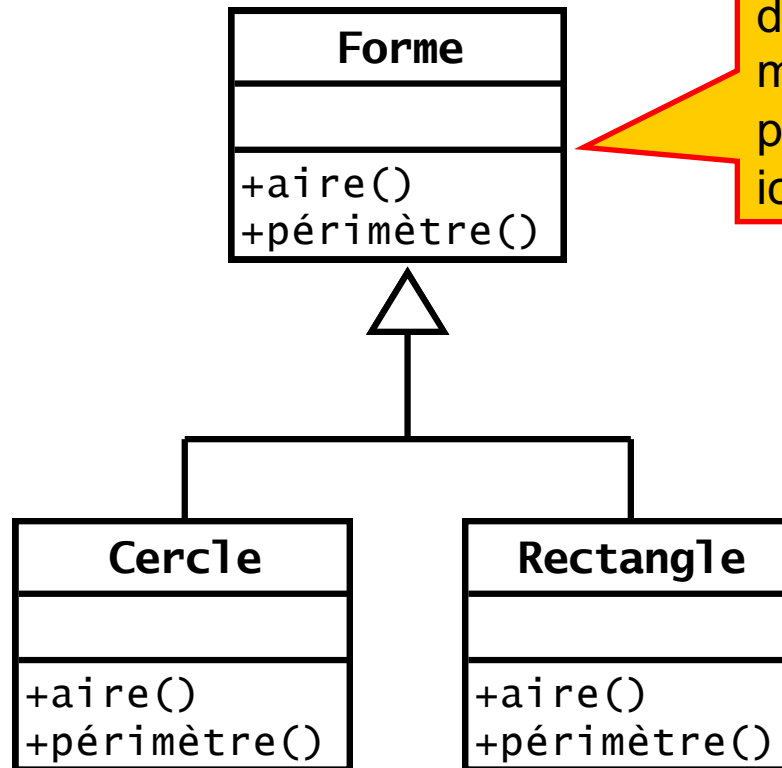
```
class A {  
    public void m1(ArrayList<A> listeA) { ... }  
    public void m2(ArrayList<? extends A> listeA) { ... }  
}  
class B extends A {  
    public static void main(String[] args) {  
        ArrayList<B> l1 = new ArrayList<B>();  
        ArrayList<? extends A> l2 = l1; // OK  
        l2.add(new B()); // erreur : modification interdite  
        A a = new A();  
        a.m1(l1); // erreur car ArrayList<B>  
        a.m2(l1); // OK mais l1 devient non modifiable dans m2  
    }  
}
```

Classes et méthodes abstraites

- Au sommet d'une hiérarchie de classes :
 - Il peut être intéressant de spécifier des méthodes que chaque sous-classe devra posséder
 - Cependant il n'est pas toujours possible de donner la définition (le code) de certaines de ces méthodes
 - Prenons un exemple...

Classes et méthodes abstraites (suite)

- Exemple



On souhaite que toute classe héritant de **Forme** définissent ces 2 méthodes... Mais on ne peut pas les implémenter ici !...

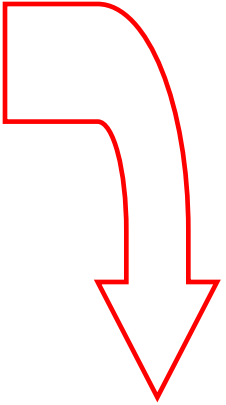
Classes et méthodes abstraites (suite)

- Pour que le compilateur accepte ces méthodes déclarées mais non définies (dans la classe *Forme*), il faut déclarer ces méthodes **abstraites** :
 - Cela s'effectue en préfixant la méthode par le mot-clé `abstract` et en ne fournissant pas de code... La méthode est simplement **déclarée**.
 - Une méthode **abstraite** doit être déclarée dans une classe **abstraite** : la classe doit alors être elle-même qualifiée `abstract`.

Classes et méthodes abstraites (suite)

- Classes précédentes implémentées avec Java :

```
public abstract class Forme {  
    private Color couleur;  
    public abstract double aire();  
    public abstract double perimetre();  
    ...  
}  
public class Cercle extends Forme {  
    private double rayon;  
    public Cercle(double rayon, Color c) {  
        super(c); this.rayon = rayon;  
    }  
    public double rayon() { return r; }  
    public double aire() { return Math.PI*r*r; }  
    public double perimetre() { return 2*Math.PI*r; }  
}
```



Classes et méthodes abstraites (suite)

- Classes précédentes implémentées avec Java (suite) :

```
public class Rectangle extends Forme {  
    private double long, larg;  
    public Rectangle (double long,double larg,Color c) {  
        super(c); this.long = long; this.larg = larg;  
    }  
    public double rayon() { return r; }  
    public double aire() { return this.long*this.larg; }  
    public double perimetre() {  
        return 2*(this.long + this.larg);  
    }  
}
```

Classes et méthodes abstraites (suite)

- Le qualificateur `abstract` devant une méthode indique que celle-ci doit être définie dans les sous-classes (sinon celles-ci devront également être déclarées abstraites)...
- Le qualificateur `abstract` devant une classe indique qu'**elle n'est pas instanciable**...
- Les classes et les méthodes abstraites permettent de :
 - Rendre compilable une classe incomplète
 - D'imposer aux sous-classes un ensemble de méthodes à implémenter

Classes et méthodes abstraites (suite)

- Utilisation de classes abstraites:

```
Forme[] formes = new Forme[3];  
formes[0] = new Cercle(2.0);  
formes[1] = new Rectangle(2.0, 3.0);  
formes[2] = new Rectangle(4.0, 1.0);
```

```
double surfaceTotale = 0;  
for(int i=0; i < formes.length; i++)  
    surfaceTotale += formes[i].aire();
```

Attention : ici il n'y a pas
instanciation d'objets !...
Mais uniquement création
d'un tableau de références

Utilisation de la
liaison dynamique

Classes et méthodes abstraites (suite)

- Pour résumer, toute méthode abstraite doit être :
 - **Déclarée** (signature) dans une classe abstraite
 - **Définie** (code) dans les sous-classes
- Mais :
 - Une classe peut être qualifiée abstraite **sans qu'elle contienne de méthodes abstraites**
 - Une classe peut hériter d'une classe abstraite sans définir les méthodes abstraites héritées à **condition qu'elle soit elle-même déclarée abstraite**

Classes et méthodes abstraites (suite)

- Exemple de classe abstraite dans la bibliothèque Java :

java.lang

Class Number

java.lang.Object

java.lang.Number

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short

```
public abstract class Number
extends Object
implements Serializable
```

← Classe abstraite

The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.

Subclasses of Number must provide methods to convert the represented numeric value to byte, double, float, int, long, and short.


Classes et méthodes abstraites (suite)

Method Summary

Methods

Modifier and Type	Method and Description
byte	<code>byteValue()</code> Returns the value of the specified number as a byte.
abstract double	<code>doubleValue()</code> Returns the value of the specified number as a double.
abstract float	<code>floatValue()</code> Returns the value of the specified number as a float.
abstract int	<code>intValue()</code> Returns the value of the specified number as an int.
abstract long	<code>longValue()</code> Returns the value of the specified number as a long.
short	<code>shortValue()</code> Returns the value of the specified number as a short.

Méthodes
concrètes



Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Classes et méthodes abstraites (suite)

- Alternative à l'utilisation d'une méthode abstraite, définir quand même un code pour que la classe se compile :

```
public class Forme {  
    private Color couleur;  
    public double aire() {  
        return 0.0;  
    }  
    public double perimetre() {  
        return 0.0;  
    }  
}
```



- **Très mauvaise idée !...**
 - Le calcul sera **faux** si on oublie de **redéfinir** la méthode

Classes et méthodes abstraites (suite)

- L'idée générale présente derrière le concept de classe abstraite est de fournir au développeur une classe **incomplète** mais qu'il pourra **adapter à ses besoins**.
- Une classe abstraite peut ainsi posséder des méthodes **implémentées** et des attributs d'instance (vs interfaces)
- Moins fréquent, le fait de qualifier abstraite une classe permet d'**interdire son instanciation**

Classes et méthodes abstraites (suite)

● Exemple d'un système de fichiers :

Dans un système de fichiers, il existe 2 types d'entités :

- Des fichiers
- Des répertoires

Ces entités possèdent des caractéristiques communes :

- Un nom
- Des droits d'accès
- Une date de création

Des particularités :

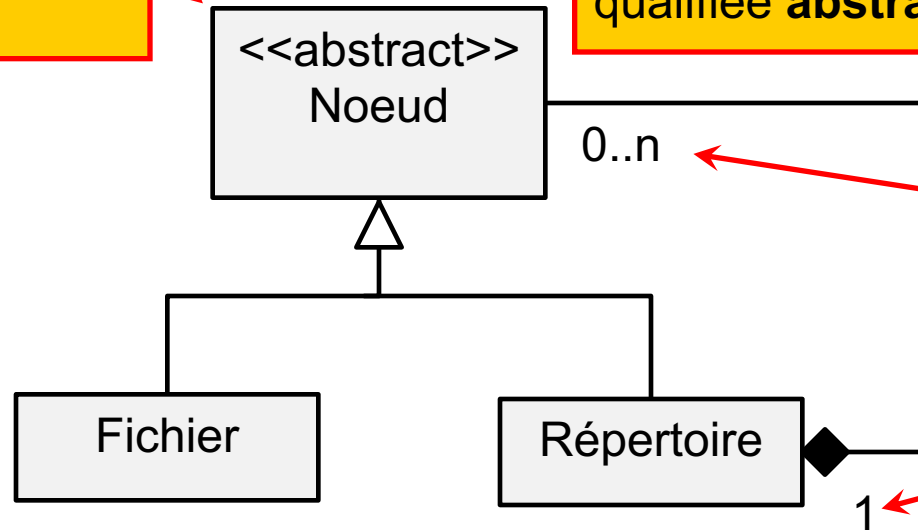
- Seuls les répertoires possèdent des éléments-fils qui peuvent être soit des répertoires, soit des fichiers

Comment représenter ces entités ?...

Classes et méthodes abstraites (suite)

Regroupement dans Nœud des caractéristiques communes

Comme cela n'a pas de sens d'instancier Nœud, la classe est qualifiée **abstraite**



L'agrégation est représentée dans la classe Répertoire par une liste de Nœud

Chaque Nœud possède une référence sur son Répertoire parent

Où placer les méthodes ajouteNoeud() et donneNoeuds() ?...

Classes et méthodes abstraites (suite)

- 1^{ière} solution : on place ces méthodes dans Répertoire.

```
public class Répertoire extends Noeud {  
    private ArrayList<Noeud> fils;  
  
    public Répertoire(String nom) {  
        super(nom);  
        this.fils = new ArrayList<Noeud>();  
    }  
  
    public void ajouteNoeud(Noeud n) {  
        this.fils.add(n);  
    }  
  
    public ArrayList<Noeud> donneNoeuds() {  
        return this.fils;  
    }  
}
```

Classes et méthodes abstraites (suite)

- L'inconvénient majeur : nécessité de tester le type dynamique de chaque Noeud avant le transtypage nécessaire à l'appel de `donneNoeuds()` :

```
abstract class Noeud {  
  
    public void explore() {  
        System.out.println(this.donneNom());  
        if (this.getClass() == Répertoire.class) {  
            Répertoire rep = (Répertoire) this;  
            for(Noeud n : rep.donneNoeuds())  
                n.explore();  
        }  
    }  
}
```

Test et transtypage peu élégants + risque d'erreur si on ajoute de nouvelles classes (ex: lien symbolique)

`donneNoeuds()` n'existe que dans la classe `Répertoire`. Si on écrit :
`this.donneNoeuds()`
alors il y aura une erreur à la compilation...

Classes et méthodes abstraites (suite)

- 2^{ème} solution : on déclare ces méthodes dans Noeud en les qualifiant abstraites.

```
public abstract class Noeud {  
    ...  
  
    public abstract void ajouteNoeud(Noeud n);  
    public abstract ArrayList<Noeud> donneNoeuds();  
}
```

```
public class Fichier extends Noeud {  
    ...  
  
    public void ajouteNoeud(Noeud n) {  
        throw new UnsupportedOperationException();  
    }  
    public ArrayList<Noeud> donneNoeuds() {  
        return new ArrayList<Noeud>();  
    }  
}
```


Normalement cette méthode ne doit pas être appelée sur une instance de Fichier...

Un fichier ne contient rien donc on retourne une liste... vide !

Classes et méthodes abstraites (suite)

```
public class Repertoire extends Noeud {  
    private ArrayList<Noeud> fils;  
  
    public Repertoire(String nom) {  
        super(nom);  
        this.fils = new ArrayList<Noeud>();  
    }  
  
    public void ajouteNoeud(Noeud n) {  
        this.fils.add(n);  
    }  
  
    public ArrayList<Noeud> donneNoeuds() {  
        return this.fils;  
    }  
}
```

Définition des
méthodes déclarées
abstraites dans Noeud



Classes et méthodes abstraites (fin)

- Avantage : plus besoin de test et de transtypage !... Et si ajout de nouvelles sous-classes, il n'y a pas nécessité d'ajouter de nouveaux tests...

```
public abstract class Noeud {  
  
    public void explore() {  
        System.out.println(this.donneNom());  
        for(Noeud n : this.donneNoeuds())  
            n.explore();  
    }  
}
```

Cet appel retourne :

- Soit une liste de nœuds si `this` référence un répertoire
- Soit une liste vide si `this` référence un fichier

Connaître la classe vs connaître le type d'un objet

- Pour connaître la classe d'un objet, Java offre la méthode `getClass` (appartenant à `Object`) qui retourne la classe à partir de laquelle a été instancié l'objet (noté `NomClasse.class`)
- Pour connaître le (ou les) types d'un objet, Java fournit l'opérateur `instanceof` qui retourne un booléen et s'utilise de la manière suivante :

NomRéférence instanceof *TypeATester*

- Soit une classe B héritant d'une classe A et b une instance de B alors :
 - `b.getClass() → B.class`
 - `b instanceof B → true` et `b instanceof A → true`

La comparaison d'objets revue avec l'héritage

- Java impose une **classe racine unique** à toutes les classes, elle est représentée par la classe `Object`
- Il est donc possible de référencer n'importe quelle instance avec une référence de type `Object`
`Object o = new MaClasse();`
- L'intérêt est de fournir certaines méthodes à l'ensemble des classes Java

La comparaison d'objets revue avec l'héritage (suite)

- Rappel : Java permet de définir des listes acceptant des instances de n'importe quel type :

Liste d'Object similaire à `ArrayList<Object>`
Jusqu'à l'introduction des types génériques (version 5), seul ce type de liste existait en Java.

```
ArrayList liste = new ArrayList();  
liste.add(new Integer(5));  
liste.add(new String("abc"));  
Integer i = (Integer) liste.get(0);
```

Liste hétérogène
mixant des
instances de
n'importe quel type

Ici la méthode `get` retourne un `Object`
donc nécessité d'effectuer un transtypage
pour retrouver le type réel de l'objet...

La comparaison d'objets revue avec l'héritage (suite)

- Pour redéfinir `equals()` de la classe `Object`, il est nécessaire de conserver sa signature :

```
public boolean equals(Object arg)
```

car certaines méthodes associées aux structures de données utilisent cette méthode.

Exemple : les méthodes `contains` et `remove` de la classe `ArrayList`

- Si on ne respecte pas cette signature, on crée une nouvelle méthode (qui **surchargera** la méthode `equals` d'`Object`) et la version définie dans `Object` continuera d'être appelée (cause fréquente d'erreur).

La comparaison d'objets revue avec l'héritage (suite)

- Exemple : spécification de la méthode `contains` dans la classe `ArrayList` :

`contains`

```
public boolean contains(Object o)
```

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

// code simplifié...

```
public boolean contains(Object o) {  
    for (int index = 0; index < size; index++) {  
        if (o.equals(elementData[index])) {  
            return true;  
        }  
    }  
    return false;  
}
```

← Tableau d'Object

Ici le compilateur détermine que `o` est du type `Object` et la seule méthode `equals` disponible dans `Object` est : `equals(Object o)`
C'est donc cette méthode ou une version redéfinie qui sera appelée à l'exécution.

La comparaison d'objets revue avec l'héritage (suite)

- Exemple des conséquences de la mauvaise redéfinition (surcharge ici) de la méthode `equals`

```
class Personne extends Object {  
    private String nom;  
  
    public Personne(String nom) { this.nom = nom; }  
  
    public boolean equals(Personne p) {  
        return (this.nom.equals(p.nom));  
    }  
  
    public static void main(String[] args) {  
        ArrayList<Personne> personnes = new ArrayList<>();  
        Personne paul1 = new Personne("Paul");  
        personnes.add(paul1);  
        Personne paul2 = new Personne("Paul");  
        System.out.println(personnes.contains(paul2));  
    }  
}
```

implicite

Surcharge de la méthode `equals` et non pas redéfinition (type du paramètre différent)

Ici l'affichage sera toujours `false` car c'est la méthode `equals` d'`Object` qui est appelée (`paul1` et `paul2` ne référencent pas le même objet)

La comparaison d'objets revue avec l'héritage (suite)

- La méthode `equals()` peut être redéfinie en utilisant :
 - Soit la méthode `getClass()` pour s'assurer que l'objet passé en paramètre appartient à la même classe que celui sur lequel est appelée la méthode `equals()`
 - Soit l'opérateur `instanceof` pour s'assurer que s'il existe des sous-classes alors elles peuvent être comparées avec des instances de la super-classe
- Les deux manières d'opérer ont chacune leurs avantages et leurs inconvénients. Le choix entre l'une ou l'autre doit être guidé par la question : *cela a-t-il un sens de comparer une instance d'une sous-classe à une instance de sa super-classe ?*... Si la réponse est oui alors il vaut mieux utiliser `instanceof`...

La comparaison d'objets revue avec l'héritage (suite)

- Version avec getClass() :

```
class Personne {  
    private String nom;  
    ...  
    public boolean equals(Object o) {  
        if (this == o)  
            return true;  
        if (o.getClass() != Personne.class)  
            return false;  
        else {  
            Personne p = (Personne) o;  
            return this.nom.equals(p.nom);  
        }  
    }  
}
```

Test pour déterminer si l'objet passé en paramètre est du même type que celui de this).

Permet d'accéder à l'objet Class correspondant

Transtypage nécessaire pour accéder à l'attribut nom

La comparaison d'objets revue avec l'héritage (suite)

- Version avec `getClass()` (suite) : un problème peut survenir si la classe `Personne` possède des sous-classes...

```
class Etudiant extends Personne {  
    private int noEtd;  
    ...  
    public boolean equals(Object o) {  
        if (this == o)  
            return true;  
        if (o.getClass() != Etudiant.class)  
            return false;  
        else {  
            Etudiant e = (Etudiant) o;  
            return super.equals(e) && this.noEtd == e.noEtd;  
        }  
    }  
}
```

Cet appel échouera car `e` est instance de la classe `Etudiant` car `e.getClass() != Personne.class`

La comparaison d'objets revue avec l'héritage (suite)

- La version avec `instanceof` solutionne ce problème...

```
class Personne {  
    private String nom;  
    ...  
    public boolean equals(Object o) {  
        if (this == o)  
            return true;  
        if (!(o instanceof Personne))  
            return false;  
        else {  
            Personne p = (Personne) o;  
            return this.nom.equals(p.nom);  
        }  
    }  
}
```

Maintenant si `o` référence une instance d'`Etudiant` alors ce test réussit car une instance d'`Etudiant` possède bien le type `Personne`

La comparaison d'objets revue avec l'héritage (fin)

- En revanche dans certains cas, permettre la comparaison d'une instance d'une sous-classe avec celle de sa super-classe n'a pas toujours de sens...

```
public class Cercle {  
    private int x,y,r;
```

```
    public boolean equals(Object o) {  
        if (this == o)  
            return true;  
        if (! (o instanceof Cercle))  
            return false;  
        else {  
            Cercle c = (Cercle) o;  
            /* comparaison de this.{x,y,r} avec c.{x,y,r} */  
        }  
    }  
}
```

```
public class CercleColoré  
    extends Cercle {  
    private Color couleur;  
    ...  
}
```

La comparaison d'une instance de Cercle avec une de CercleColoré n'a pas toujours un sens...

Ici un test avec getClass() serait plus pertinent...

Comparaison surcharge vs redéfinition

- Contrairement à une méthode redéfinie, le choix d'une **méthode surchargée** se fait toujours à la **compilation** en fonction du type **statique** des paramètres
- Par exemple, s'il existe deux méthodes `equals` dans la classe `CercleColoré` :

```
public boolean equals(Object o) { ... }  
public boolean equals(CercleColoré o) { ... }
```

Le code suivant appellera les deux versions:

```
Object o = new CercleColoré(...);  
CercleColoré cc = new CercleColoré(...);  
o.equals(cc); // appel d'equals(Object) de CercleColoré  
cc.equals(o); // appel d'equals(Object) de CercleColoré
```

La solution à ce problème est de **ne jamais surcharger** la méthode `equals`

Comparaison surcharge vs redéfinition (suite)

- Lorsque le compilateur a le choix entre plusieurs versions surchargées du même nom de méthode, il choisit celle qu'il recherchera à **l'exécution** en se fondant sur le **type statique** du (ou des) paramètre(s)
- À l'exécution, le choix d'une méthode redéfinie s'effectue en se fondant sur **le type dynamique** de l'objet appelé
- Lorsqu'on **redéfinit une méthode ne jamais modifier sa signature** car, dans ce cas, on effectue une surcharge et non une redéfinition !...

Comparaison surcharge vs redéfinition (suite)

- Exemple :

```
class A {  
    void m() { System.out.println("m() de A"); }  
    void m(Object o) {  
        System.out.println("m(Object) de A");  
    }  
}  
class B extends A {  
    void m() { System.out.println("m() de B"); }  
    void m(A a) {  
        System.out.println("m(A) de B");  
    }  
}
```

Dans la classe B, la méthode m() est redéfinie
et une nouvelle méthode m(A) est créée

Comparaison surcharge vs redéfinition (suite)

- Exemple (suite) :

```
class Test {  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = new B();  
        B b1 = (B) a2;  
        a1.m();  
        a2.m();  
        a1.m(a2);  
        a2.m(a1);  
        b1.m(a1);  
    }  
}
```

```
> java Test  
m() de A  
m() de B  
m(Object o) de A  
m(Object o) de A  
m(A a) de B
```

Ici, à la compilation, il y a :

1. détermination du type statique de a1 (ici A)
2. puis recherche dans la classe A d'une méthode m « compatible » avec un paramètre de type A

Comparaison surcharge vs redéfinition (fin)

- Pour résumer... Quand le compilateur rencontre l'appel d'une méthode **d'instance** :
 1. Il détermine le type **statique** (la classe) de la référence appelée et détermine également les types statiques des paramètres fournis
 2. Il vérifie qu'il existe une méthode dans cette classe possédant la signature requise (paramètres) ou étant compatible. Si tel n'est pas le cas alors il génère une erreur...
 3. Si la méthode existe alors il génère un test qui permettra, **à l'exécution**, de déterminer le type **dynamique** de la référence appelée et de choisir une version redéfinie de la méthode.

Pour résumer...

- La notion de **classe abstraite** représente une classe **partiellement définie** qui doit être **complétée** pour être utilisée (instanciée)
- Une classe abstraite est complétée en créant une sous-classe qui n'est **concrète** que **si on définit toutes les méthodes abstraites héritées**
- Implémentations différentes d'équals (getClass vs instanceof) selon que la comparaison avec des instances de sous-classes a un sens ou non.
- Le choix d'une méthode **surchargée** s'effectue à la **compilation** en fonction des types **statiques** des paramètres fournis.