Heritage et Polymorphisme

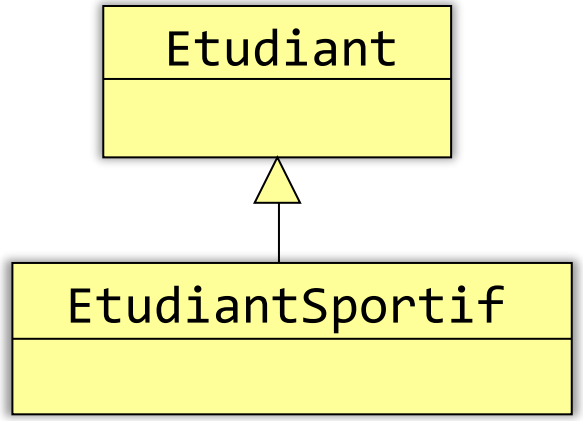
Philippe Genoud

dernière mise à jour :23/01/2024 16:51

Surclassement

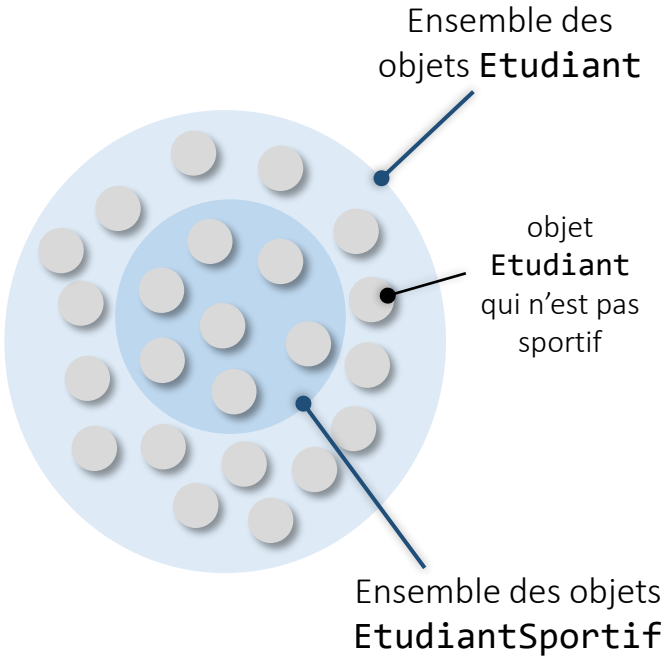
- La réutilisation du code est un aspect important de l'héritage, mais ce n'est peut être pas le plus important
- Le deuxième point **fondamental** est la relation qui relie une classe à sa super-classe :

Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A.



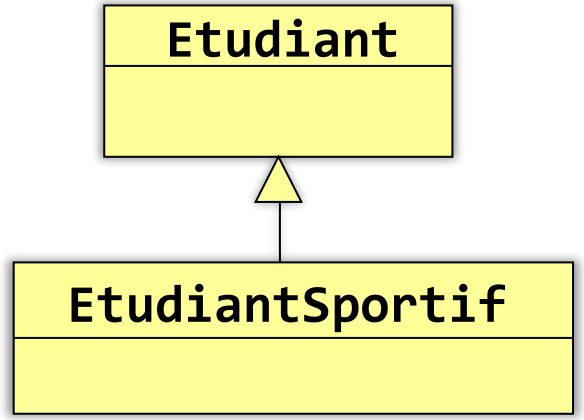
Un `EtudiantSportif` est un `Etudiant`

L'ensemble des étudiants sportifs est inclus dans l'ensemble des étudiants



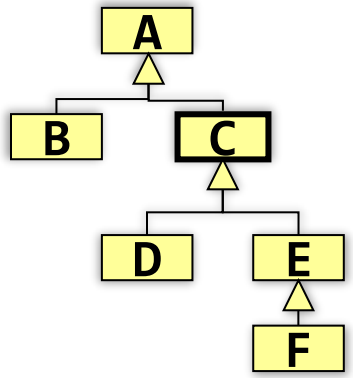
Surclassement

- Si la classe **B** est une sous-classe de la classe **A**, tout objet instance **B** peut être aussi vu comme une instance de **A**.
- Cette relation est directement supportée par le langage JAVA :
  - à une référence déclarée de type **A** il est possible d'affecter une valeur qui est une référence vers un objet de type **B** (**surclassement** ou **upcasting**)




```
Etudiant e;
e = new EtudiantSportif(...);
```

  - plus généralement à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte du type de la référence



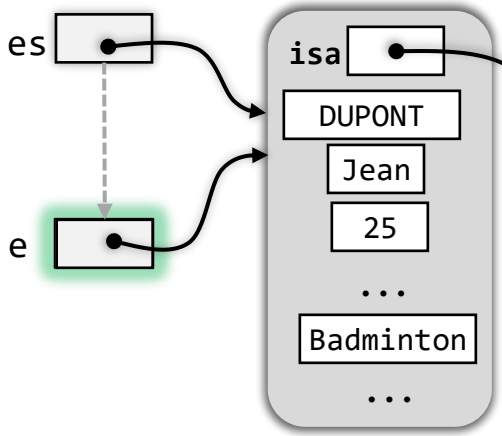

```
C c;
c = new D();
c = new E();
c = new F();
```

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

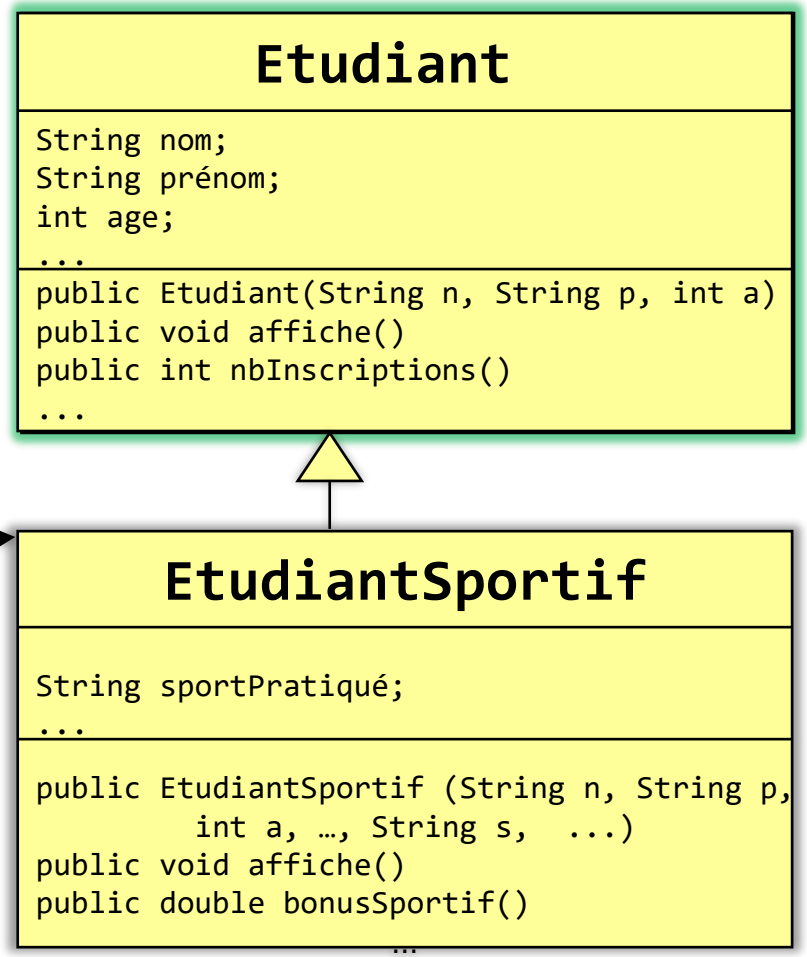
Surclassement

- Lorsqu'un objet est "sur-classé" il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner.
  - Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence.

```
EtudiantSportif es =
   new EtudiantSportif("DUPONT","Jean",
                       25,..,"Badminton",..);

Etudiant e;
e = es;  // upcasting

e.affiche();
es.affiche();

System.out.println(e.nbInscriptions());
System.out.println(es.nbInscriptions());

System.out.println(es.bonusSportif());
System.out.println(e.bonusSportif());
```



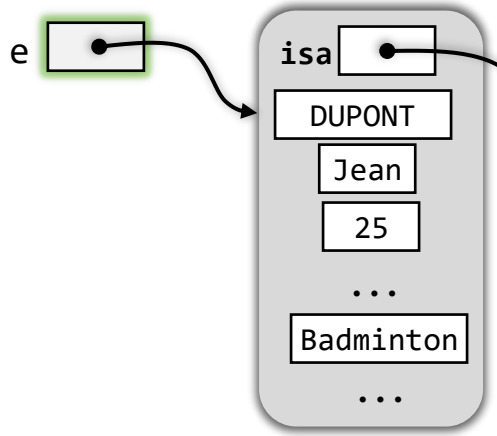
Le compilateur refuse ce message: pas de méthode **bonusSportif** définie dans la classe **Etudiant**



Lien dynamique

Résolution des messages

```
Etudiant e = new EtudiantSportif(
    "DUPONT","Jean",25,..,
    "Badminton",..);
```




```
Etudiant
public void affiche() {
    System.out.println(
       "Nom : "+nom+"\n"
       "Prénom : "+prénom+"\n"
       "Age : "+age+ ...);
}
```

```
EtudiantSportif
public void affiche() {
    super.affiche();
    System.out.println(
      "Sport" : "+sport+"\n"
      + ...);
}
```

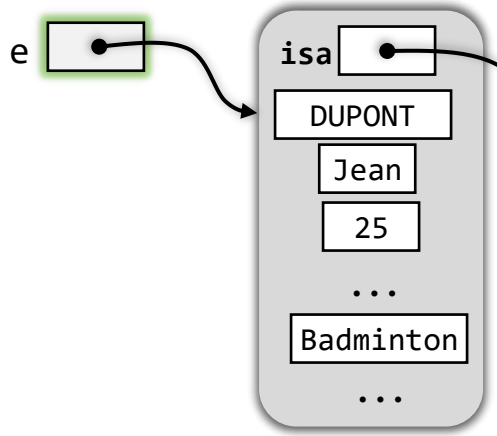
Que va donner **`e.affiche()`** ?

```
e.affiche();
```

Lien dynamique

Résolution des messages

```
Etudiant e = new EtudiantSportif(
    "DUPONT","Jean",25,..,
    "Badminton",..);
```



Etudiant

```
public void affiche() {
    System.out.println(
       "Nom : "+nom+"\n"
       "Prénom : "+prénom+"\n"
       "Age : "+age+ ...);
}
```

EtudiantSportif

```
public void affiche() {
    super.affiche();
    System.out.println(
      "Sport" : "+sport+"\n"
      + ...);
}
```

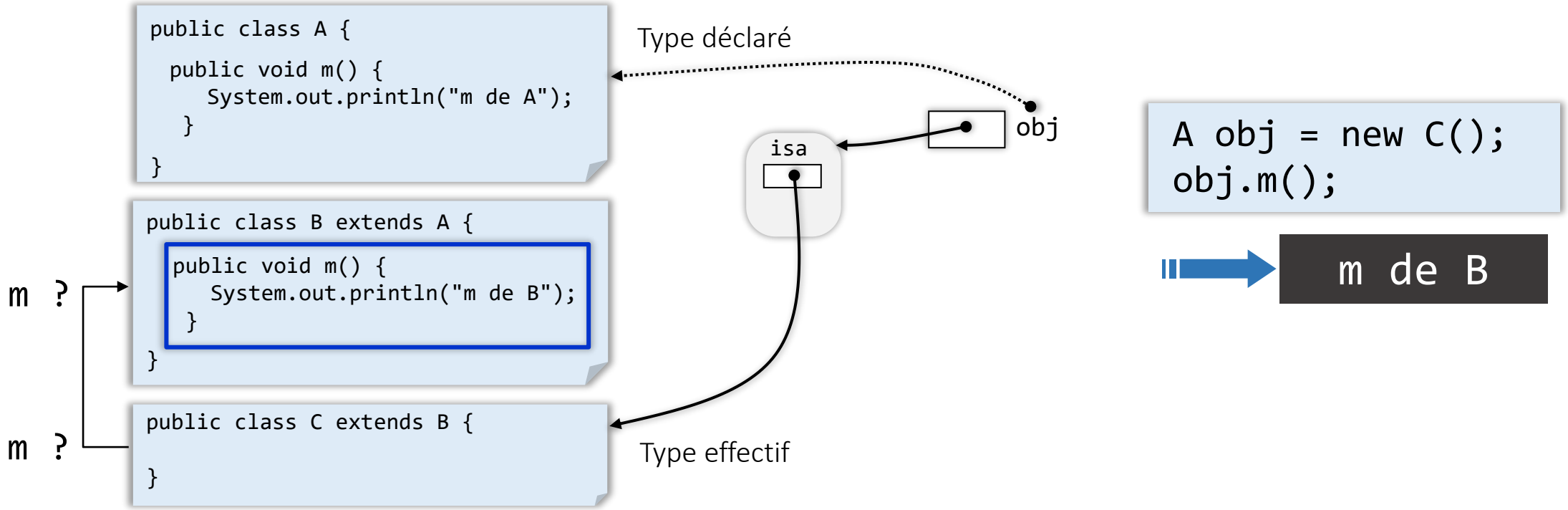
Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée", c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est en fait invoquée et exécutée

```
e.affiche();
```

```
Nom : DUPONT
Prénom : Jean
Age : 25
…
Sport : Badminton
```

Lien dynamique

- Les messages sont résolus **à l'exécution**
  - la méthode exécutée est déterminée à l'exécution (run-time) et non pas à la compilation
  - à cet instant le type exact de l'objet qui reçoit le message est connu
    - la méthode définie pour le type réel de l'objet recevant le message est appelée (et non pas celle définie pour son type déclaré).



- Ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamic binding, late-binding ou run-time binding)

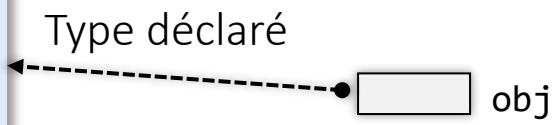
Lien dynamique

Vérifications statiques

- **A la compilation**: seules des **vérifications statiques** qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées
  - la classe déclarée de l'objet recevant un message doit posséder une méthode dont la signature correspond à la méthode appelée.

```
public class A {
  public void m1() {
    System.out.println("m1 de A");
  }
}
```

```
public class B extends A {
  public void m1() {
    System.out.println("m1 de B");
  }
  public void m2() {
    System.out.println("m2 de B");
  }
}
```




```
A obj = new B();
obj.m1();
obj.m2();
```

B est un sous type de A (upcasting)

m1 est bien définie pour le type **A**

m2 n'est définie pour le type **A**

javac

```
Test.java:21: cannot resolve symbol
symbol  : method m2  ()
location: class A
    obj.m2();
       ^
1 error
```

- garantit dès la compilation que les messages pourront être résolus au moment de l'exécution
  → robustesse du code

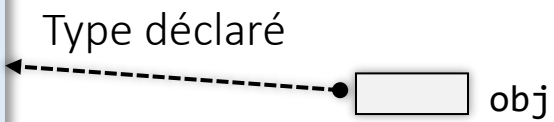
Lien dynamique

Vérifications statiques

- vérifications statiques: garantissent dès la compilation que les messages pourront être résolus au moment de l'exécution

```
public class A {
  public void m1() {
    System.out.println("m1 de A");
  }
}
```

```
public class B extends A {
  public void m1() {
    System.out.println("m1 de B");
  }
  public void m2() {
    System.out.println("m2 de B");
  }
}
```




```
A obj;
for (int i = 0; i < 10; i++) {
   double hasard = Math.random();
   if (hasard < 0.5) {
      obj = new A();
   }
   else {
      obj = new B();
   }
   obj.m1();
}
```

- à la compilation il n'est pas toujours possible de déterminer le type exact de l'objet récepteur d'un message

Lien dynamique

Choix des méthodes, sélection du code

```
A refA = new A();
1 refA.m1();
2 refA.m1(10);
  refA = new B();
3 refA.m1();
```

Compilation → javac

Byte-code

```
a invokevirtual ... <Method m1()>
b invokevirtual ... <Method m1(int)>
c invokevirtual ... <Method m1()>
```

Exécution → java

```
m1() de A
m1(int) de A
m1() de B
```

Le choix de la méthode à exécuter est effectué **statiquement à la compilation** en fonction du type déclaré de l'objet récepteur du message et des paramètres de la méthode

`refA` est de type **A** existe-t-il une méthode `m1()` dans la classe **A** ?

`refA` est de type **A** existe-t-il une méthode `m1(int)` dans la classe **A** ?

`refA` est de type **A** existe-t-il une méthode `m1()` dans la classe **A** ?

`refA` désigne un objet de type **A** exécuter la méthode `m1()` de la classe **A**

`refA` désigne un objet de type **A** exécuter la méthode `m1(int)` de la classe **A**

`refA` désigne un objet de type **B** exécuter la méthode `m1(int)` de la classe **B**

```
public class A {
  public void m1() {
    System.out.println("m1() de A");
  }

  public void m1(int x) {
    System.out.println("m1(int) de A");
  }
}
```

```
public class B extends A {
  public void m1() {
    System.out.println("m1() de B");
  }
  public void m2() {
    System.out.println("m2() de B");
  }
}
```

La sélection du code à exécuter est effectuée **dynamiquement à l'exécution** en fonction du type effectif du récepteur du message

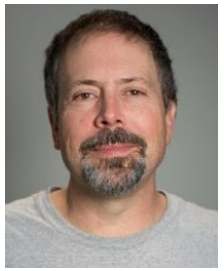
Polymorphisme

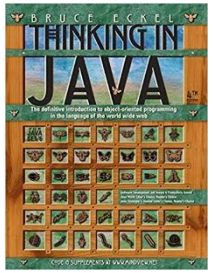
A quoi servent l'upcasting et le lien dynamique ?
A la mise en œuvre du polymorphisme

- Le terme polymorphisme (du grec πολύμορφος , « multiforme ») décrit la caractéristique d'un élément qui peut se présenter sous différentes formes.

- En programmation par objets, on appelle polymorphisme
  - le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe,
  - le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.

- *"Le **polymorphisme** constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage"*

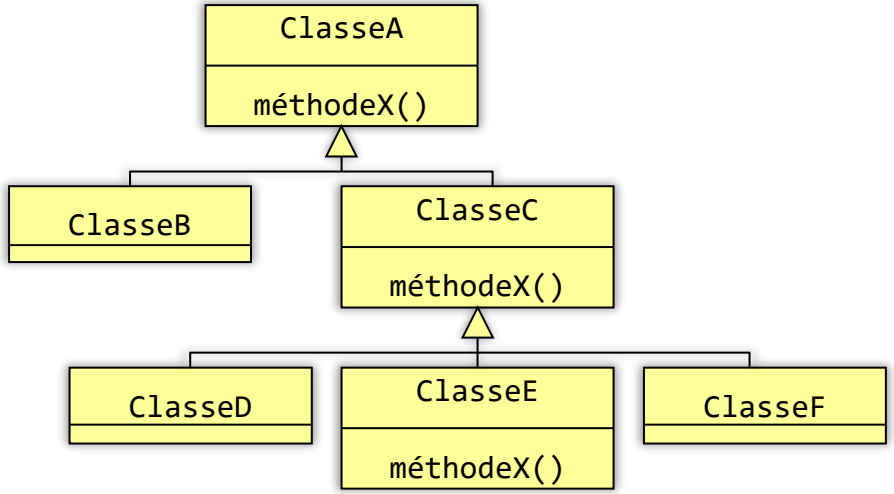




Bruce Eckel - "Thinking in JAVA"
4th Edition – 2006

https://archive.org/details/TIJ4CcR1/page/n3

Polymorphisme




```
ClasseA objA;
objA = ...
objA.methodeX();
```

Surclassement
la référence peut désigner des objets de classes différentes (n'importe quelle sous classe de `ClasseA`)

+

Lien dynamique
Le comportement est différent selon la classe effective de l'objet

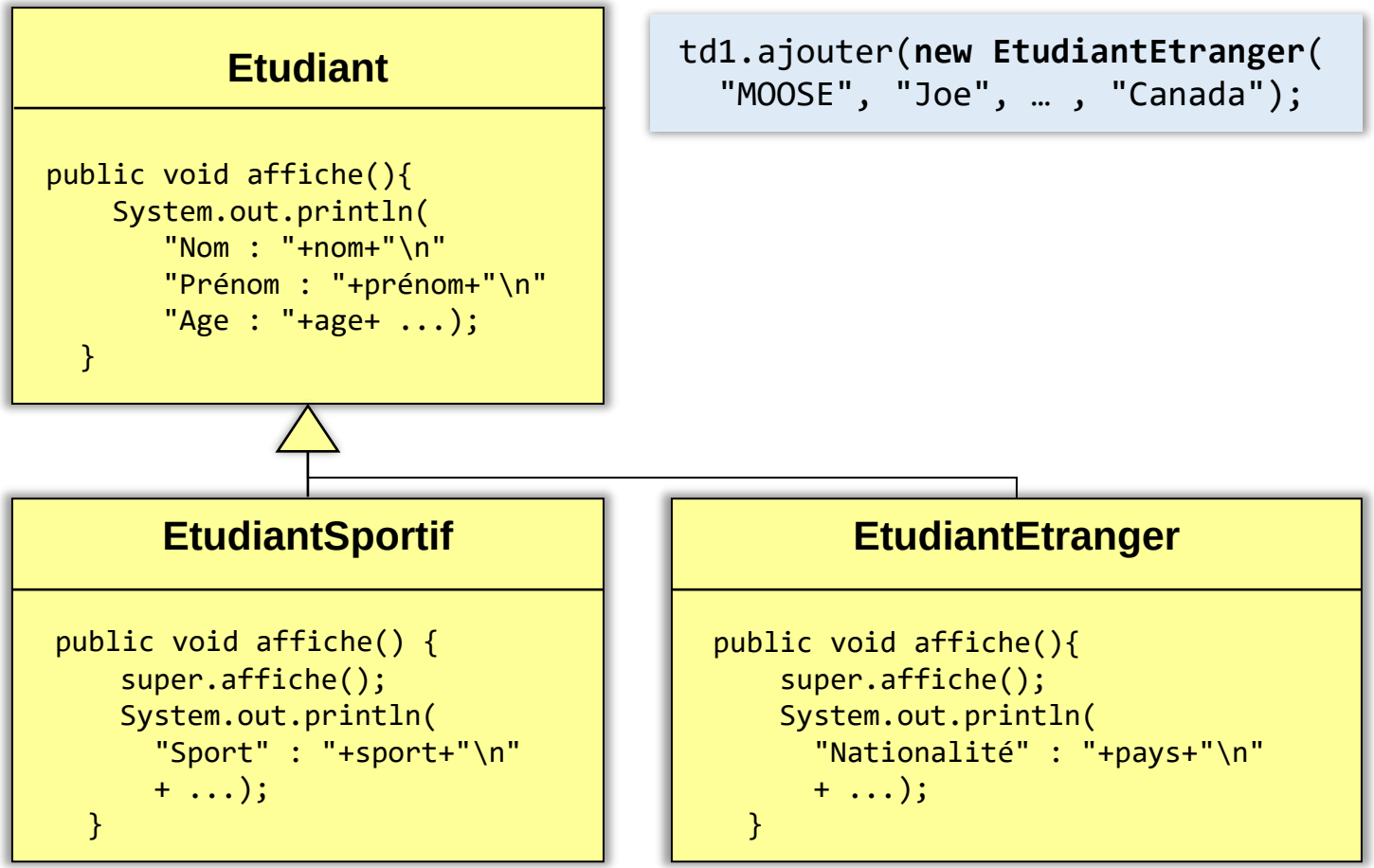
- un cas particulier de polymorphisme (polymorphisme par sous-typage)
- Permet la manipulation **uniforme** des objets de plusieurs classes par l'intermédiaire d'une classe de base commune

Polymorphisme

`liste` peut contenir des étudiants de n'importe quel type

```
GroupeTD td1 = new GroupeTD();
td1.ajouter(new Etudiant("DUPONT", …));
td1.ajouter(new EtudiantSportif("BIDULE", "Louis", … , "ski alpin");
```

```
td1.ajouter(new EtudiantEtranger(
  "MOOSE", "Joe", … , "Canada");
```




```
public class GroupeTD{

  private Etudiant[] liste = new Etudiant[30];
  private int nbEtudiants = 0;
  ...

  public void ajouter(Etudiant e) {
     if (nbEtudiants < liste.lenght) {
        liste[nbEtudiants++] = e;
     }
  }

  public void afficherListe(){
      for (int i=0;i<nbEtudiants; i++) {
         liste[i].affiche();
      }
  }

}
```

Si un nouveau type d'étudiant est défini, le code de **GroupeTD** reste inchangé

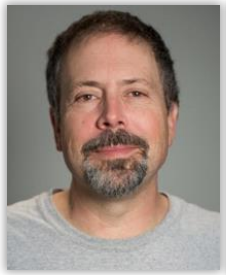
Polymorphisme

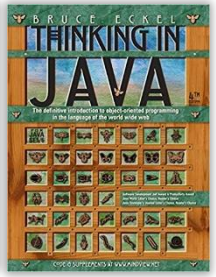
- En utilisant le polymorphisme en association à la liaison dynamique
  - plus besoin de distinguer différents cas en fonction de la classe des objets
  - possible de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule l'interface de la classe de base

- Développement **plus rapide**
- Plus grande **simplicité** et **meilleure organisation** du code
- Programmes plus facilement **extensibles**
- Maintenance du code **plus aisée**

« Once you know that all method binding in Java
happens polymorphically via late binding,
you can always write your code to talk to the base class,
and know that all the derived-class cases will work
correctly using the same code.
Or put it another way, you send a message to an object
and let the object figure out the right thing to do »

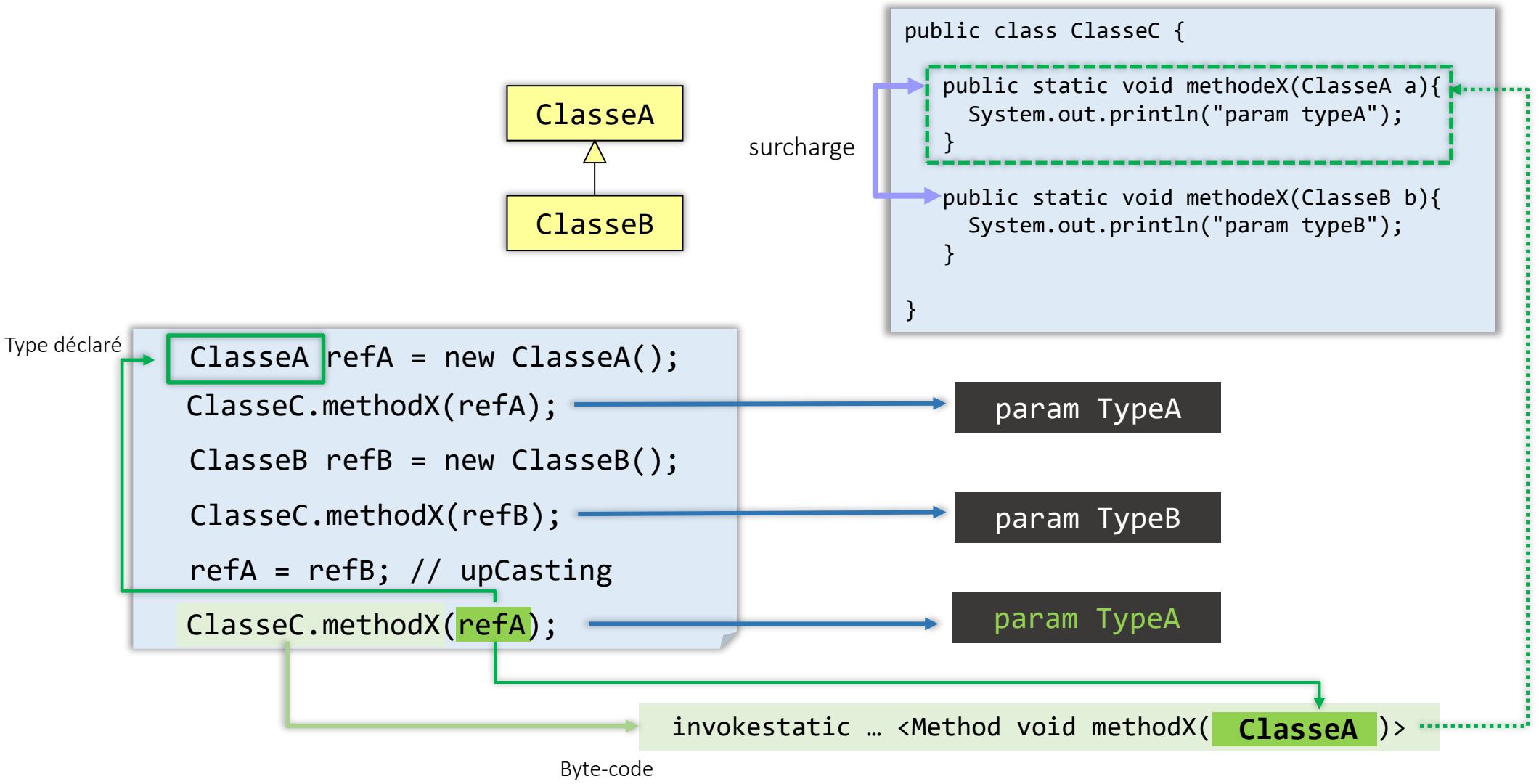
Bruce Eckel - "Thinking in JAVA"
4th Edition – 2006





https://archive.org/details/TIJ4CcR1/page/n3

Surcharge et polymorphisme



Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : **Sélection statique**

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

```
public class ClasseC {
   public void m1(ClasseA a){
     System.out.println("m1(A) de C : " + a);
   }

   public void m1(ClasseB b){
     System.out.println("m1(B) de C : " + b);
   }
}
```

surcharge

```
public class ClasseD extends ClasseC{
   public void m1(ClasseB b){
     System.out.println("m1(B) de D : " + b);
   }
}
```

```
ClasseC c = new ClasseD();
```

```
ClasseB refB  = new ClasseB();
```

```
c.m1(refB);
```

→ ???

```
ClasseA refA = refB;
```

```
c.m1(refA);
```

→ ???

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

surcharge

```
public class ClasseC {
   public void m1(ClasseA a){
     System.out.println("m1(A) de C : " + a);
   }

   public void m1(ClasseB b){
     System.out.println("m1(B) de C : " + b);
   }
}
```

```
public class ClasseD extends ClasseC{
   public void m1(ClasseB b){
     System.out.println("m1(B) de D : " + b);
   }
}
```

```
ClasseC c = new ClasseD();
```

```
ClasseB refB  = new ClasseB();
```

```
c.m1(refB);
```

→ `m1(B) de D : je suis un B`

```
ClasseA refA = refB;
```

```
c.m1(refA);
```

→ ???

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

surcharge

```
public class ClasseC {
   public void m1(ClasseA a){
     System.out.println("m1(A) de C :" + a);
   }

   public void m1(ClasseB b){
     System.out.println("m1(B) de C : " + b);
   }
}
```

```
public class ClasseD extends ClasseC{
   public void m1(ClasseB b){
     System.out.println("m1(B) de D : " + b);
   }
}
```

```
ClasseC c = new ClasseD();
```

```
ClasseB refB  = new ClasseB();
```

```
c.m1(refB);
```

→ `m1(B) de D : je suis un B`

```
ClasseA refA = refB;
```

```
c.m1(refA);
```

→ `m1(A) de C : je suis un B`

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

surcharge

```
public class ClasseC {
   public void m1(ClasseA a){
     System.out.println("m1(A) de C : " + a);
   }

   public void m1(ClasseB b){
     System.out.println("m1(B) de C" + b);
   }
}
```

```
public class ClasseD extends ClasseC{
   public void m1(ClasseB b){
     System.out.println("m1(B) de D : " + b);
   }
}
```

```
ClasseC c = new ClasseD();

ClasseB refB  = new ClasseB();

1 c.m1(refB);

ClasseA refA = refB;

c.m1(refA);
```

javac

Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : **Sélection statique**

Byte-code

1 Type déclaré : récepteur **ClasseC**, paramètre **ClasseB**
Recherche dans **ClasseC** d'une méthode **m1(ClasseB)** → `Invokevirtual m1(ClasseB)` a

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

surcharge

```
public class ClasseC {
    public void m1(ClasseA a){
      System.out.println("m1(A) de C" + a);
    }

    public void m1(ClasseB b){
      System.out.println("m1(B) de C" + b);
    }
}
```

```
public class ClasseD extends ClasseC{
   public void m1(ClasseB b){
     System.out.println("m1(B) de D : " + b);
   }
}
```

```
ClasseC c = new ClasseD();

ClasseB refB  = new ClasseB();

1 c.m1(refB);

ClasseA refA = refB;

2 c.m1(refA);
```

javac

Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : **Sélection statique**

Byte-code

1. Type déclaré : récepteur **ClasseC**, paramètre **ClasseB**
   Recherche dans **ClasseC** d'une méthode **m1(ClasseB)** → `Invokevirtual m1(ClasseB)` **a**
2. Type déclaré : récepteur **ClasseC**, paramètre **ClasseA**
   Recherche dans **ClasseC** d'une méthode **m1(ClasseA)** → `Invokevirtual m1(ClasseA)` **b**

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

surcharge

```
public class ClasseC {
    public void m1(ClasseA a){
      System.out.println("m1(A) de C" + a);
    }

    public void m1(ClasseB b){
      System.out.println("m1(B) de C" + b);
    }
}
```

```
public class ClasseD extends ClasseC{
    public void m1(ClasseB b){
      System.out.println ("m1(B) de D" + b);
    }
}
```

```
ClasseC c = new ClasseD();

ClasseB refB  = new ClasseB();

1 c.m1(refB);        → m1(B) de D : je suis un B

ClasseA refA = refB;

2 c.m1(refA);
```

javac

Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : **Sélection statique**

Byte-code

1. Type déclaré : récepteur **ClasseC**, paramètre **ClasseB**
   Recherche dans **ClasseC** d'une méthode **m1(ClasseB)** → `Invokevirtual m1(ClasseB)` **a**
2. Type déclaré : récepteur **ClasseC**, paramètre **ClasseA**
   Recherche dans **ClasseC** d'une méthode **m1(ClasseA)** → `Invokevirtual m1(ClasseA)` **b**

java

La sélection du code à exécuter est effectuée **dynamiquement à l'exécution** en fonction du type effectif du récepteur du message

a Recherche d'une méthode **m1(ClasseB)** en partant du type effectif du récepteur **ClasseD** ,exécution du code

Surcharge et polymorphisme

```
public class ClasseA {
    @Override
    public String toString() {
        return "je suis un A";
    }
}
```

```
public class ClasseB extends ClasseA{
    @Override
    public String toString() {
        return "je suis un B";
    }
}
```

```
public class ClasseC {
    public void m1(ClasseA a){
      System.out.println("m1(A) de C" + a);
    }

    public void m1(ClasseB b){
      System.out.println("m1(B) de C" + b);
    }
}
```

surcharge

```
public class ClasseD extends ClasseC{
    public void m1(ClasseB b){
      System.out.println ("m1(B) de D" + b);
    }
}
```

```
ClasseC c = new ClasseD();

ClasseB refB  = new ClasseB();

1 c.m1(refB);        → m1(B) de D : je suis un B

ClasseA refA = refB;

2 c.m1(refA);        → m1(A) de C : je suis un B
```

javac

Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : **Sélection statique**

Byte-code

1. Type déclaré : récepteur **ClasseC**, paramètre **ClasseB**
   Recherche dans **ClasseC** d'une méthode **m1(ClasseB)** → `Invokevirtual m1(ClasseB)` **a**
2. Type déclaré : récepteur **ClasseC**, paramètre **ClasseA**
   Recherche dans **ClasseC** d'une méthode **m1(ClasseA)** → `Invokevirtual m1(ClasseA)` **b**

java

La sélection du code à exécuter est effectuée **dynamiquement à l'exécution** en fonction du type effectif du récepteur du message

- **a** Recherche d'une méthode **m1(ClasseB)** en partant du type effectif du récepteur **ClasseD** ,exécution du code
- **b** Recherche d'une méthode **m1(ClasseA)** en partant du type effectif du récepteur **ClasseD** ,exécution du code

Downcasting (transtypage)

- Le **upcasting** permet d'utiliser à la compilation un type (classe) plus général pour accéder à un objet d'un type donné

```
public class Etudiant {
    ...
    public void afficherResultats() {
        ...
    }
}
```

```
public class EtudiantSportif extends Etudiant {
    ...
    public void afficherResultatsSportifs() {
       ...
    }

    public void afficherResultats() {
        super.afficherResultats();
        afficherResultatsSportifs();
    }
}
```

```
EtudiantSportif es =
   new EtudiantSportif(...);
Etudiant e = es;
```

```
e.afficherResultats();
e.afficherResultatsSportifs();
```

- Par contre le compilateur limite l'utilisation d'un objet 'upcasté' à l'interface du type de la référence

```
COMPILATION ERROR :
-------------------------------------------------------------
fr/im2ag/mcci/exemplescoursheritage/upcasting/TestEtudiants.java:[17,10] cannot find
symbol
  symbol:   method afficherResultatsSportifs()
  location: variable e of type fr.im2ag.mcci.exemplescoursheritage.upcasting.Etudiant
1 error
```

- Le **downcasting** (ou transtypage) permet de « forcer un type » de référence à la compilation

```
ClasseX obj = …
ClasseA  a = (ClasseA) obj;
```

  - C'est une « promesse » que l'on fait au moment de la compilation.

- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de `obj` soit « compatible » avec le type `ClasseA`
  - Compatible ⇔ la même classe ou n'importe quelle sous classe de `ClasseA` (`obj instanceof ClasseA`)

- Si la promesse n'est pas tenue une **erreur d'exécution** se produit.
  - `ClassCastException` est levée et arrêt de l'exécution

```
java.lang.ClassCastException: ClasseX
        at Test.main(Test.java:52)
```

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);

p1.equals(p2);
```


```
public class Point {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
      return this.x == pt.x && this.y == pt.y;
   }
}
```

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Point {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
      return this.x == pt.x && this.y == pt.y;
   }
}
```

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
```

```
p1.equals(p2);
```
→ **true**

```
Object o = p2;
p1.equals(o);
```
? → Erreur compilation, pas de méthode equals(Object) dans Point

? → **true**

? → **false**

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Point {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
      return this.x == pt.x && this.y == pt.y;
   }
}
```

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2);
```

→ **true**

```
Object o = p2;
p1.equals(o);
```

→ Erreur compilation, pas de méthode equals(Object) dans Point

→ **true**

→ **false**

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o;
 }
 ...
}
```

```
public class Point extends Object {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
      return this.x == pt.x && this.y == pt.y;
   }
}
```

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2);
```

→ **true**

```
Object o = p2;
1 p1.equals(o);
```

? Erreur compilation, pas de méthode equals(Object) dans Point

? **true**

javac

Existe-t-il une méthode `equals(Object)` dans la classe `Point` ?

Oui : cette méthode est héritée de **Object**

```
a Invokevirtual equals(Object)
```

Compilation : **analyse statique** du code

choix de la méthode à exécuter basé sur les types déclarés (de l'objet récepteur du message et des éventuels arguments passé en paramètres)

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o;
 }
 ...
}
```

```
public class Point extends Object {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
      return this.x == pt.x && this.y == pt.y;
   }
}
```

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);

p1.equals(p2);
```

→ **true**

```
Object o = p2;
1 p1.equals(o);
```

- ? Erreur compilation, pas de méthode equals(Object) dans Point
- ? **true**
- **false**

javac

Existe-t-il une méthode `equals(Object)` dans la classe `Point` ?

Oui : cette méthode est héritée de **Object**

Compilation : **analyse statique** du code

choix de la méthode à exécuter basé sur les types déclarés (de l'objet récepteur du message et des éventuels arguments passé en paramètres)

a `Invokevirtual equals(Object)`

java

Exécution : **lien dynamique**

La sélection du code à exécuter est effectuée en fonction du type effectif de l'objet récepteur du message

a Recherche d'une méthode `equals(Object)` en partant de `Point` (classe effective de l'objet récepteur du message)
Exécution du code

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o;
 }
 ...
}
```

```
public class Point extends Object {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
     return this.x == pt.x && this.y == pt.y;
   }
}
```

surcharge (*overload*) de la méthode `equals(Object o)` héritée de `Object`

De manière générale, il vaut mieux éviter de surcharger des méthodes en spécialisant les arguments

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
Object o = p2;
```

Exécution : lien dynamique

java

```
invokevirtual ... <Method equals(Point)>
invokevirtual ... <Method equals(Object)>
invokevirtual ... <Method equals(Object)>
```

La sélection du code à exécuter est effectuée en fonction du **type effectif** de l'objet récepteur du message

Compilation : **analyse statique** du code

javac

```
p1.equals(p2)
p1.equals(o)
o.equals(p1)
```

choix de la méthode à exécuter basé sur les **types déclarés** (de l'objet récepteur du message et des éventuels arguments passé en paramètres)

```
true
false
false
```

comportements incohérents

A propos de equals

- Tester l'égalité de deux objets de la même classe

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o;
 }
 ...
}
```

```
public class Point extends Object {

   private double x;
   private double y;

   ...

   public boolean equals(Point pt) {
     return this.x == pt.x && this.y == pt.y;
   }
}
```

redéfinition (*overriding*) de la méthode `equals(Object o)` héritée de `Object`

```
@Override
public boolean equals(Object o) {
    if (this == o)
         return true;

    if (! (o instanceof Point))
         return false;

    Point pt = (Point) o; // downcasting
    return this.x == pt.x && this.y == pt.y;
}
```

surcharge (*overload*) de la méthode `equals(Object o)` héritée de `Object`

De manière générale, il vaut mieux éviter de surcharger des méthodes en spécialisant les arguments

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
Object o = p2;
```

Exécution : lien dynamique

java

```
invokevirtual ... <Method equals(Point)>
invokevirtual ... <Method equals(Object)>
invokevirtual ... <Method equals(Object)>
```

La sélection du code à exécuter est effectuée en fonction du **type effectif** de l'objet récepteur du message

Compilation : **analyse statique** du code

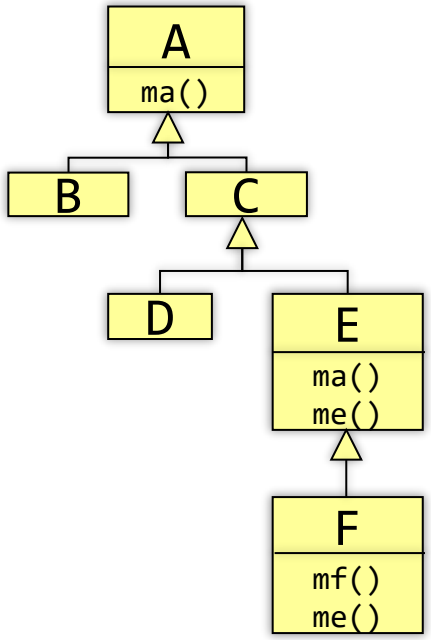
javac

`p1.equals(p2)`	true
`p1.equals(o)`	true
`o.equals(p1)`	true

comportements identiques

choix de la méthode à exécuter basé sur les **types déclarés** (de l'objet récepteur du message et des éventuels arguments passé en paramètres)

Upcasting/Downcasting




```
class A {
    public void ma() {
        System.out.println("methode ma définie dans A");
    }
}
```

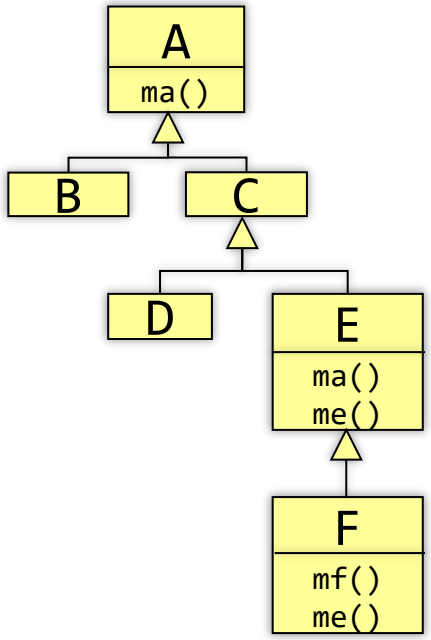
```
class E extends C {
    public void ma() {
        System.out.println("methode ma redéfinie dans E");
    }
    public void me() {
        System.out.println("methode me définie dans E");
    }
}
```

```
class F extends E {
    public void mf() {
        System.out.println("methode mf définie dans f");
    }
    public void me() {
        System.out.println("methode me redéfinie dans F");
    }
}
```

```
C c = new F();
```

	compilation	exécution
`c.ma();`		
`c.mf();`		
`B b = c;`		
`E e = c;`		
`E e = (E)c;` `e.me();`		
`D d = (D) c;`		

Upcasting/Downcasting




```
class A {
    public void ma() {
        System.out.println("methode ma définie dans A");
    }
}
```

```
class E extends C {
    public void ma() {
        System.out.println("methode ma redéfinie dans E");
    }
    public void me() {
        System.out.println("methode me définie dans E");
    }
}
```

```
class F extends E {
    public void mf() {
        System.out.println("methode mf définie dans f");
    }
    public void me() {
        System.out.println("methode me redéfinie dans F");
    }
}
```

```
C c = new F();
```

	compilation	exécution
`c.ma();`	La classe `C` hérite d'une méthode `ma`	→ `méthode ma redéfinie dans E`
`c.mf();`	*Cannot find symbol : metod mf()* Pas de méthode `mf()` définie au niveau de la classe `C`	
`B b = c;`	*Incompatible types* Un `C` n'est pas un `B`	
`E e = c;`	*Incompatible types* Un C n'est pas forcément un E	
`E e = (E)c;` `e.me();`	Transtypage (Dowcasting), le compilateur ne fait pas de vérification La classe E définit bien une méthode me	→ `méthode me redéfinie dans F`
`D d = (D) c;`	Transtypage (Dowcasting), le compilateur ne fait pas de vérification	`ClassCastException` Un `F` n'est pas un `D`