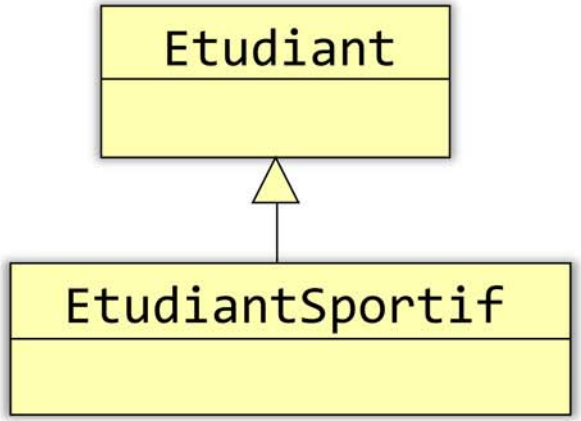
Heritage et Polymorphisme

dernière mise à jour : 4/01/2015

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.

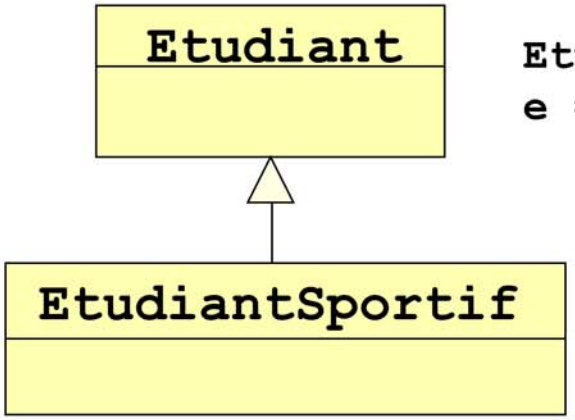


Un `EtudiantSportif` est un `Etudiant`

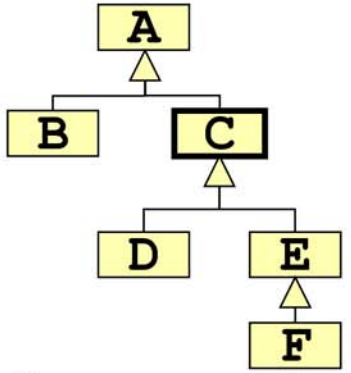
L'ensemble des étudiants sportifs est inclus dans l'ensemble des étudiants

Surclassement

- tout objet instance de la classe **B** peut être aussi vu comme une instance de la classe **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(...);
```



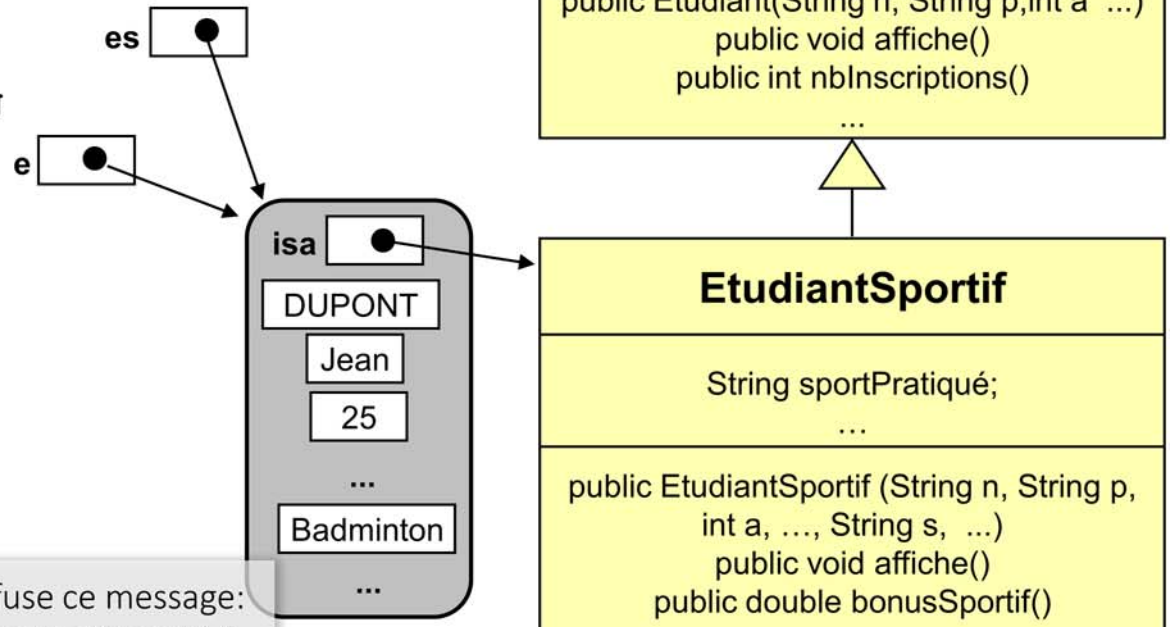

```
C c;
c = new D();
c = new E();
c = new F();
c = new A();
c = new B();
```

- *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*

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

Etudiant e;
e = es;  // upcasting

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

e.nbInscriptions();
es.nbInscriptions();

es.bonusSportif();
e.bonusSportif();
```



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

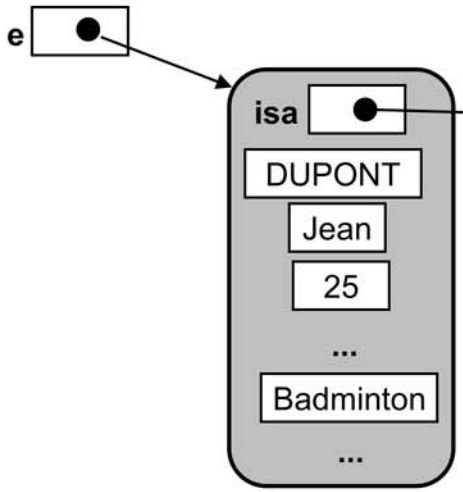
Lien dynamique

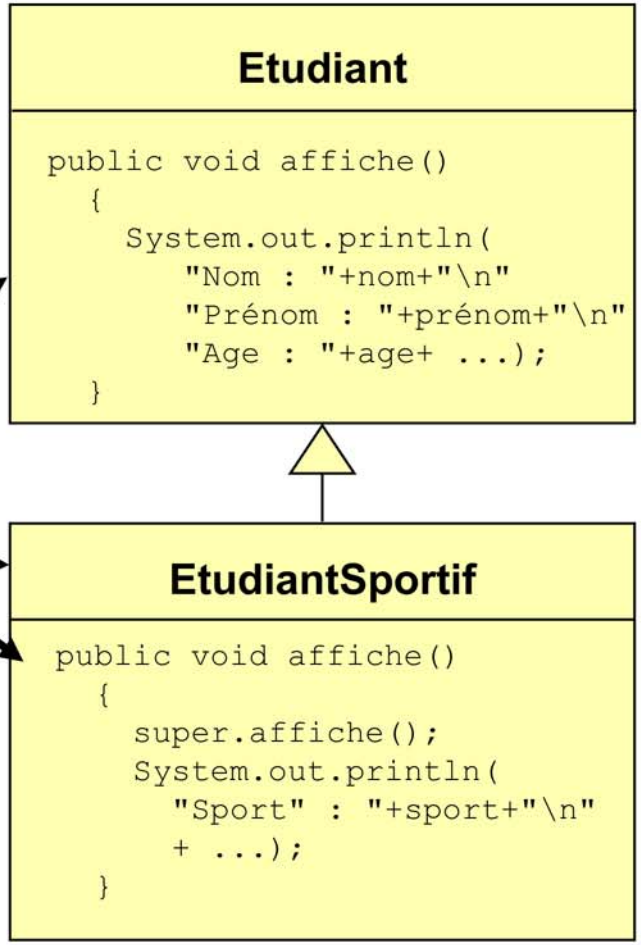
Résolution des messages

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

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

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





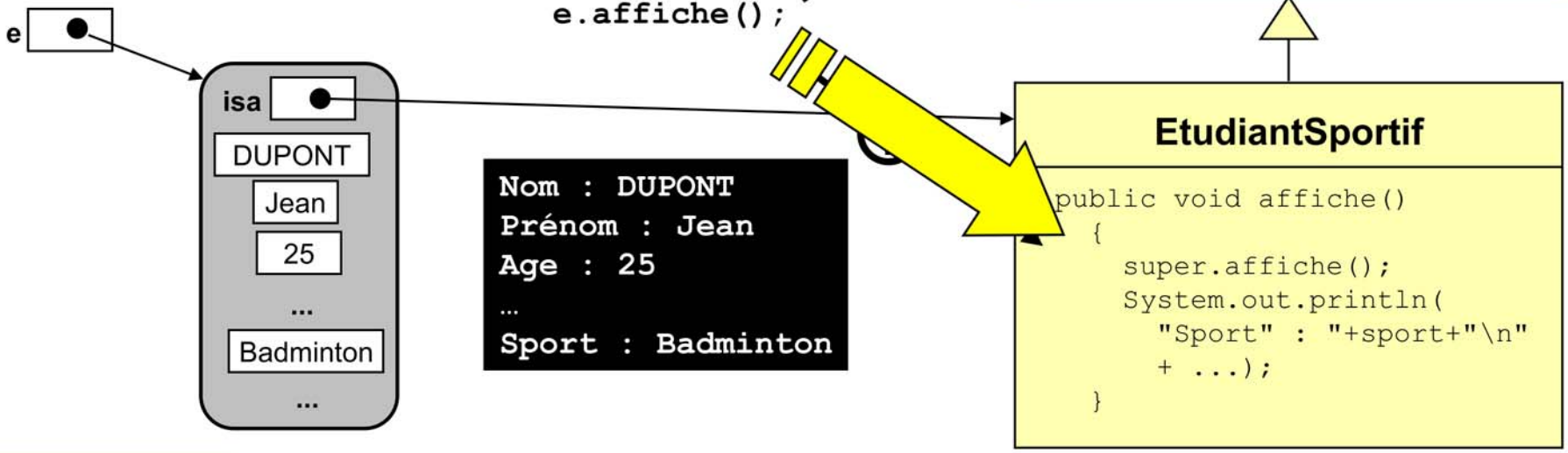
Lien dynamique

Résolution des messages

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

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

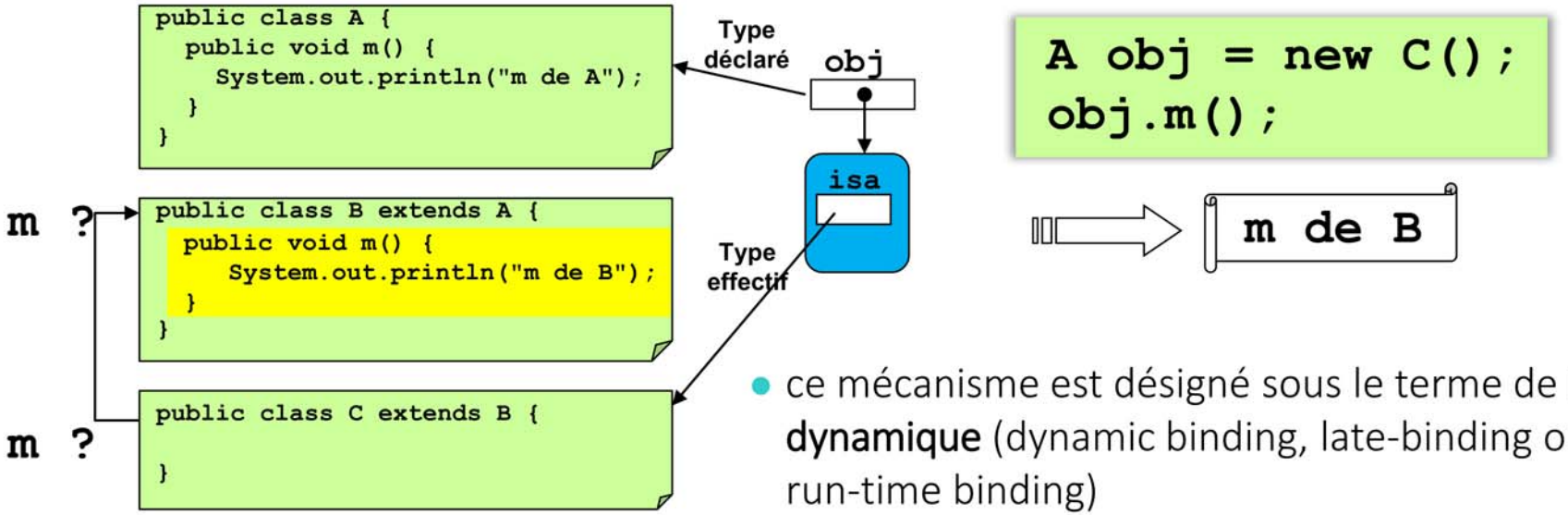
```
e.affiche();
```



Lien dynamique

Mécanisme de résolution des messages

- 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).*

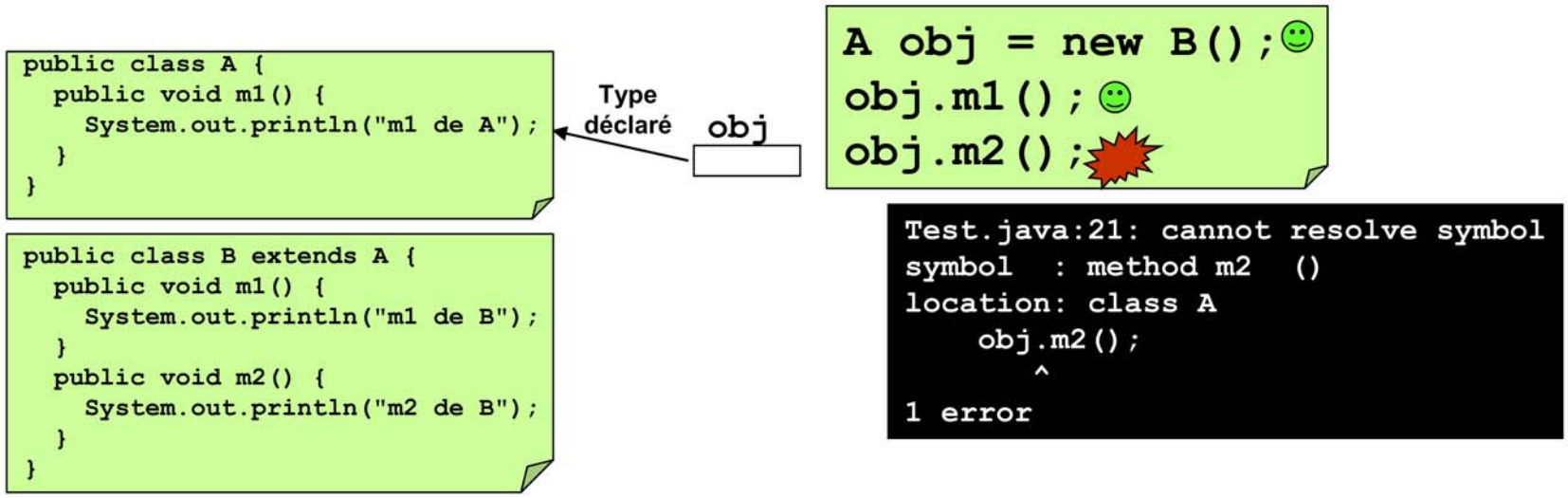


- ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamic binding, late-binding ou run-time binding)

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.*

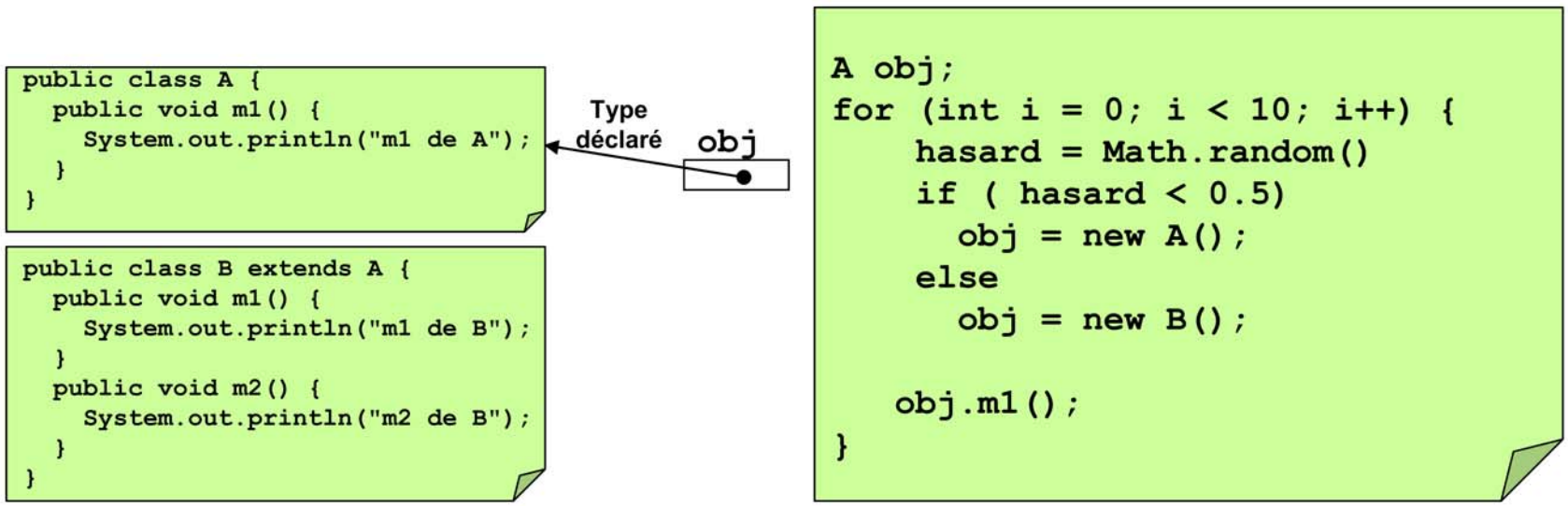


- garantir dès la compilation que les messages pourront être résolus au moment de l'exécution → robustesse du code

Lien dynamique

Vérifications statiques

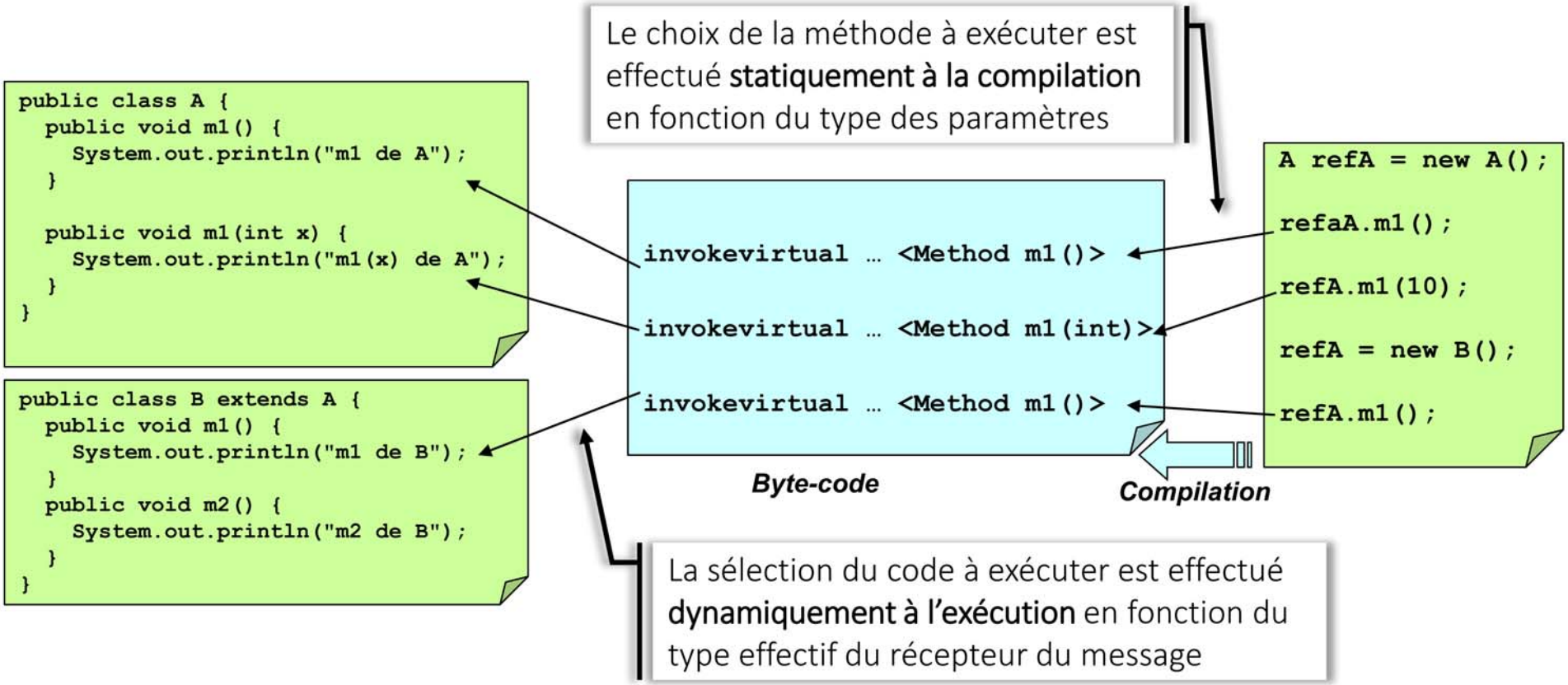
- à la compilation il n'est pas possible de déterminer le type exact de l'objet récepteur d'un message



- vérification statique: garantit dès la compilation que les messages pourront être résolus au moment de l'exécution

Lien dynamique

Choix des methodes, sélection du code



Polymorphisme

A quoi servent l'upcasting et le lien dynamique ?

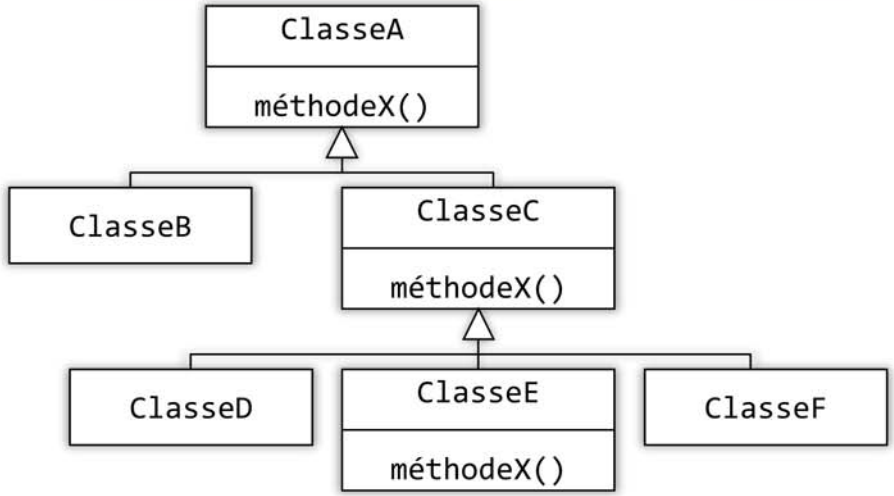
A la mise en œuvre du polymorphisme

- Le terme polymorphisme 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"*

Polymorphisme




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

Surclassement
la référence peut désigner des objets de classe différente (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)

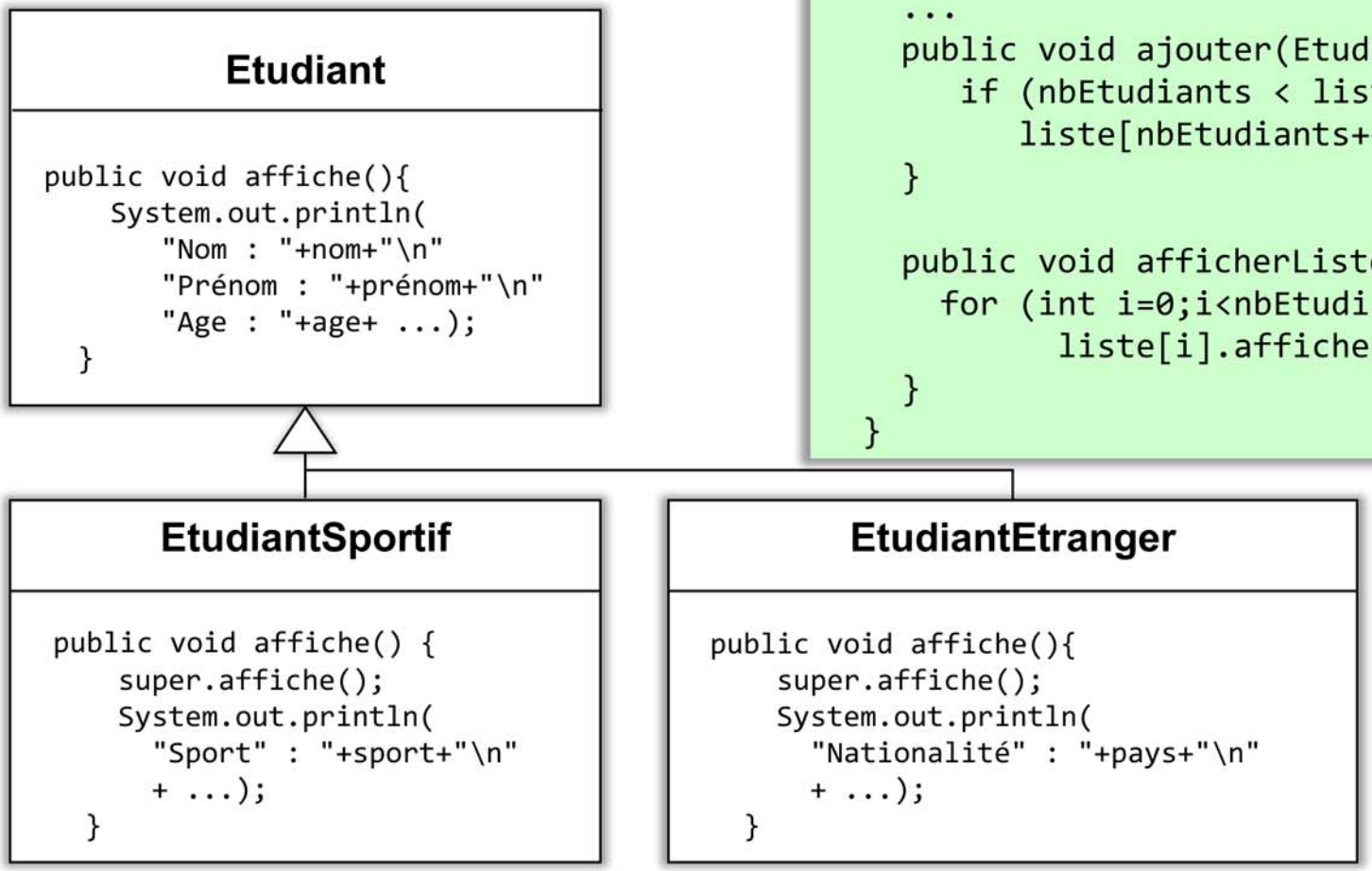
manipulation uniforme des objets de plusieurs classes par l'intermédiaire d'une classe de base commune

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");
```

```
public class GroupeTD{

   Etudiant[] liste = new Etudiant[30];
   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**

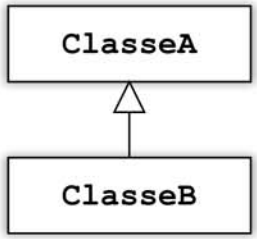
Polymorphisme

Pour conclure

« 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

Surcharge et polymorphisme




```
public class ClasseC {

   public static void methodeX(ClasseA a){
      System.out.println("param typeA");
   }

   public static void methodeX(ClasseB b){
      System.out.println("param typeB");
   }

}
```

surcharge

```
ClasseA refA = new ClasseA();
ClasseC.methodX(refA);          ——→  param TypeA
ClasseB refB = new ClasseB();
ClasseC.methodX(refB);          ——→  param TypeB
refA = refB; // upCasting
ClasseC.methodX(refA);          ——→  param TypeA
```

```
invokestatic … <Method void methodX( ClasseA )>
```

Byte-code

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

Downcasting

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

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

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

```
public class Point {

      private double x;
      private double y;

      ...

}
```

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

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

surcharge (overloads) la méthode `equals(Object o)` héritée de `Object`

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2);   --> true
Object o = p2;
p1.equals(o)     --> false
o.equals(p1)     --> false
```

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 du type déclaré du (des) paramètre(s)

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
 }
 ...
}
```

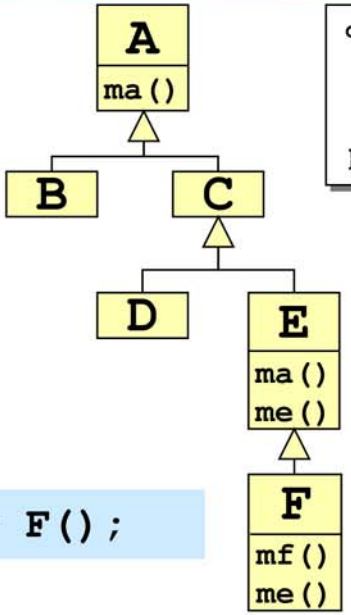
```
public class Point {

     private double x;
     private double y;

     ...

}
```

```
@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;
}
```

redéfinir (overrides) la méthode equals(Object o) héritée de Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2)        --> true
Object o = p2;
p1.equals(o)         --> true
o.equals(p1)         --> true
```

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 définie 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 définie dans F
`D d = (D) c;`	Transtypage (Dowcasting), le compilateur ne fait pas de vérification	*ClassCastException* Un F n'est pas un D

Héritage et abstraction
classes abstraites