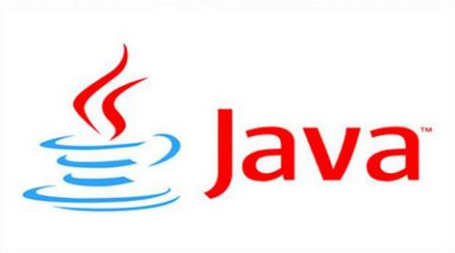
Faculté des Sciences
كلية العلوم
جامعة ابن طفيل Université Ibn Tofaïl

Département d'Informatique
Filière SMI - Semestre 6
2020-2021

Programmation Orientée Objet

Application avec Java

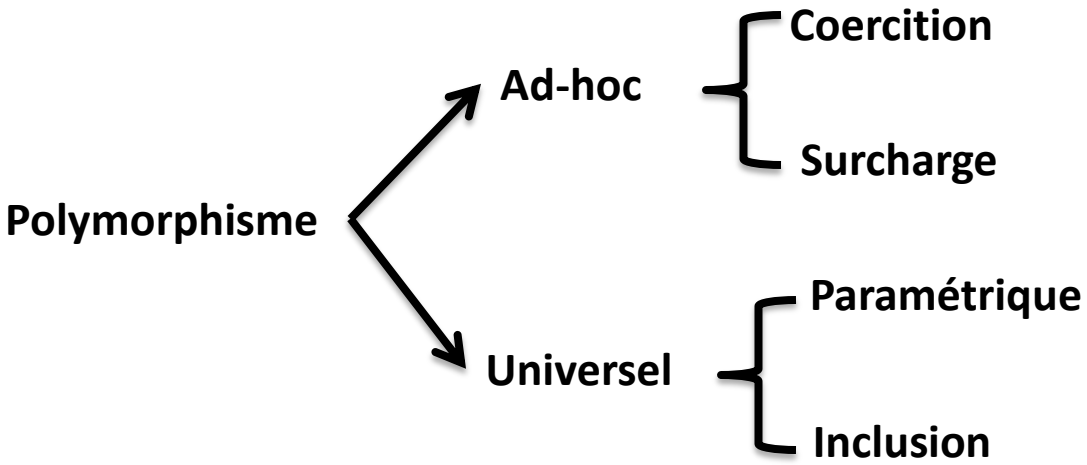


Pr. Khadija Louzaoui

Chapitre 4
Polymorphisme et Héritage

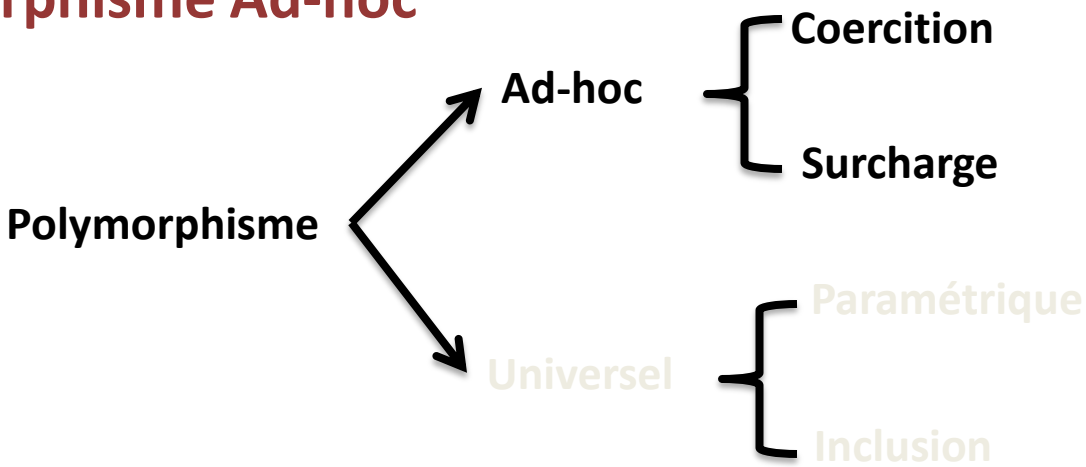
Qu'est-ce que le polymorphisme ?

- Le polymorphisme est un concept fondamental de la programmation orientée objet.
- Dans la langue grec, il signifie « **prendre plusieurs formes** ».
- le polymorphisme représente la capacité d'une entité à posséder plusieurs formes. Ici, "l'entité" en question est le **type**.
- On trouve différents types de polymorphismes :



- Ad-hoc: fonctionne sur un nombre limité de types qui n'ont pas de lien entre eux ;
- Universel: fonctionne sur un ensemble défini de types qui ont une structure commune ;

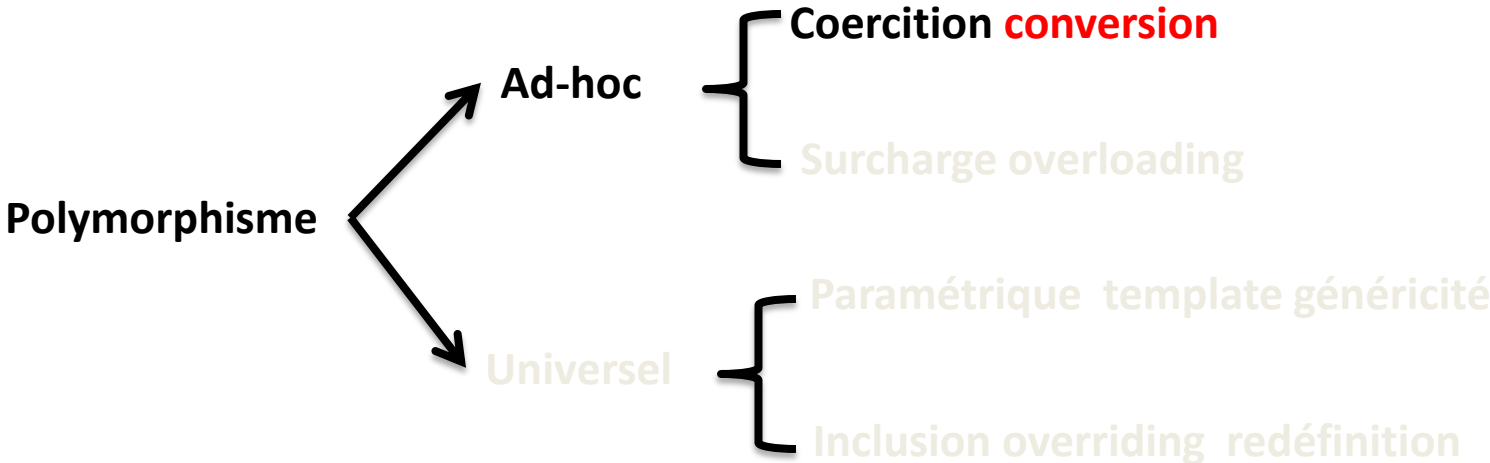
le polymorphisme Ad-hoc



- Deux mécanismes permettent de fonctionner "sur un nombre limité de type non liés entre eux" :

  - ✓ l'un la conversion de type ;
  - ✓l'autre utilise la surcharge;

Coercition

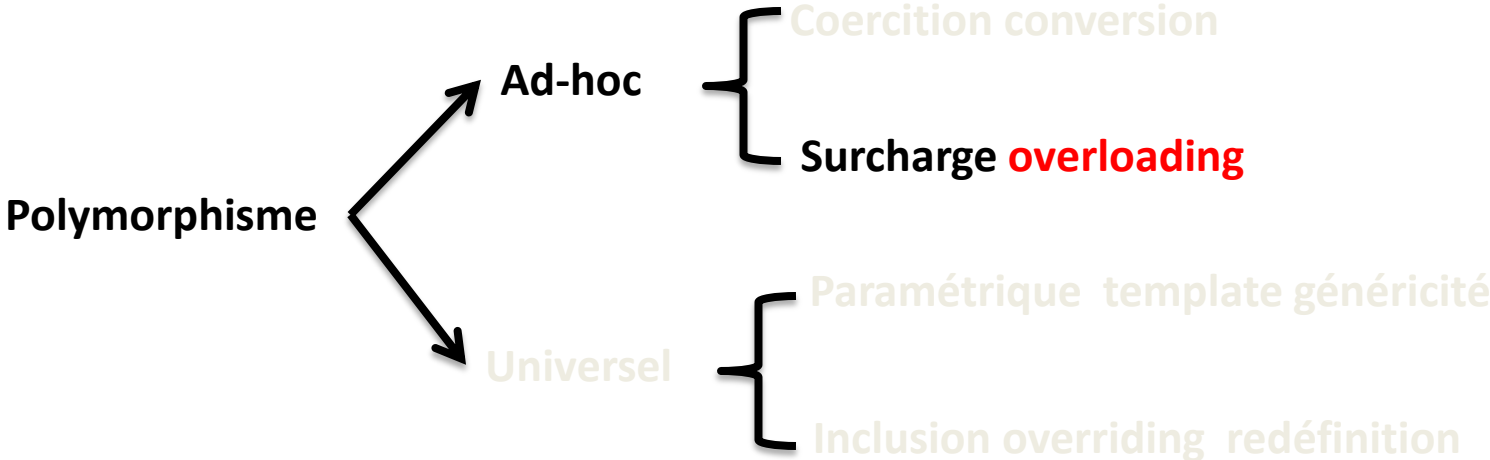


"**conversion**" d'un type vers celui attendu par un opérateur ou une fonction, ce qui permet d'éviter les erreurs de compilation. Son expression la plus connue est le casting.
Ce casting est soit implicite, soit explicite.

- Il est fait automatiquement par le compilateur pour les types de base (int en double, etc.)
- Il est fait explicitement par le programmeur, pour peu que le type casté fasse bien parti de la hiérarchie du type attendu

```
int valInt = 2;
double valDouble = 2.2;
double resultat = valInt + valDouble;
// Conversion implicite de valInt en double.
```

Surcharge



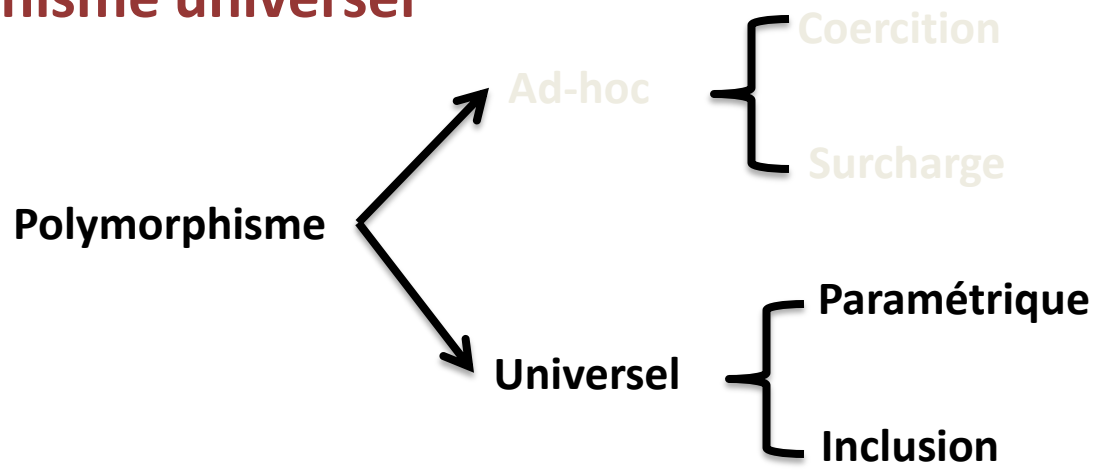
"**overloading**" : permet l'usage du même opérateur ou du même nom de méthode pour plusieurs significations distinctes de programmation.

Java supporte certains opérateurs surchargés, mais ne permet pas la définition de ses propres opérateurs.

Java supporte la surcharge des noms de méthodes, pour peu que les signatures soient différentes.

```
void mafonction (int arg);
int mafonction (double arg);
char mafonction (char arg, char arg2);
```

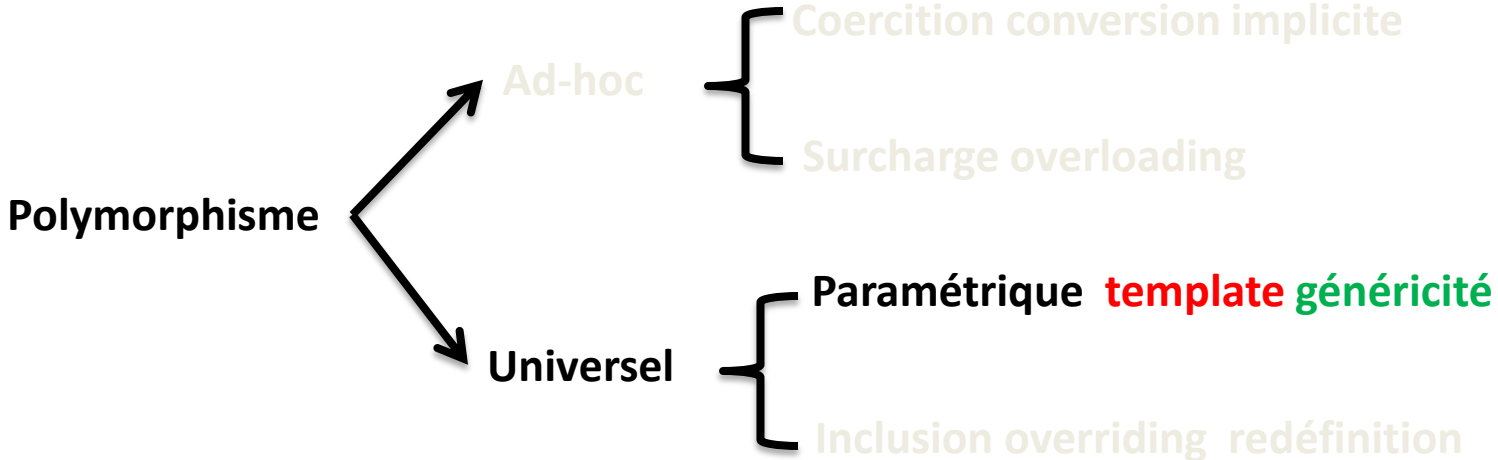
Polymorphisme universel



- Deux mécanismes permettent de définir un "ensemble défini de type possédant une structure commune".

  ✓ l'un est à chercher dans le paradigme de programmation générique;
  ✓ l'autre dans l'héritage et le binding de type.

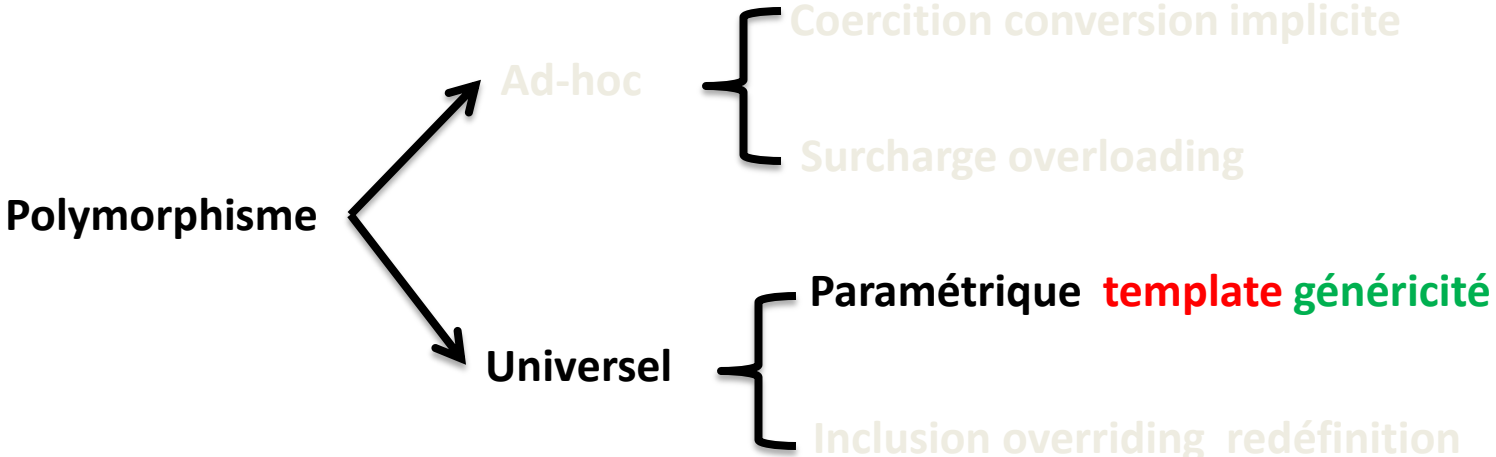
Généricité



Le polymorphisme paramétrique passe par l'utilisation des techniques génériques pour offrir un même service pour tout un ensemble de types .
une seule abstraction s'applique à un ensemble de type.
Typiquement, List<T> peut s'appliquer à tout type. Il suffit "d'instancier le template" pour une classe donné.

```
public class Exemple <T> {
    private T var;
    public T getVar() {
        return var;
    }
    public void setVar(T var) {
        this.var = var;
    }
}
```

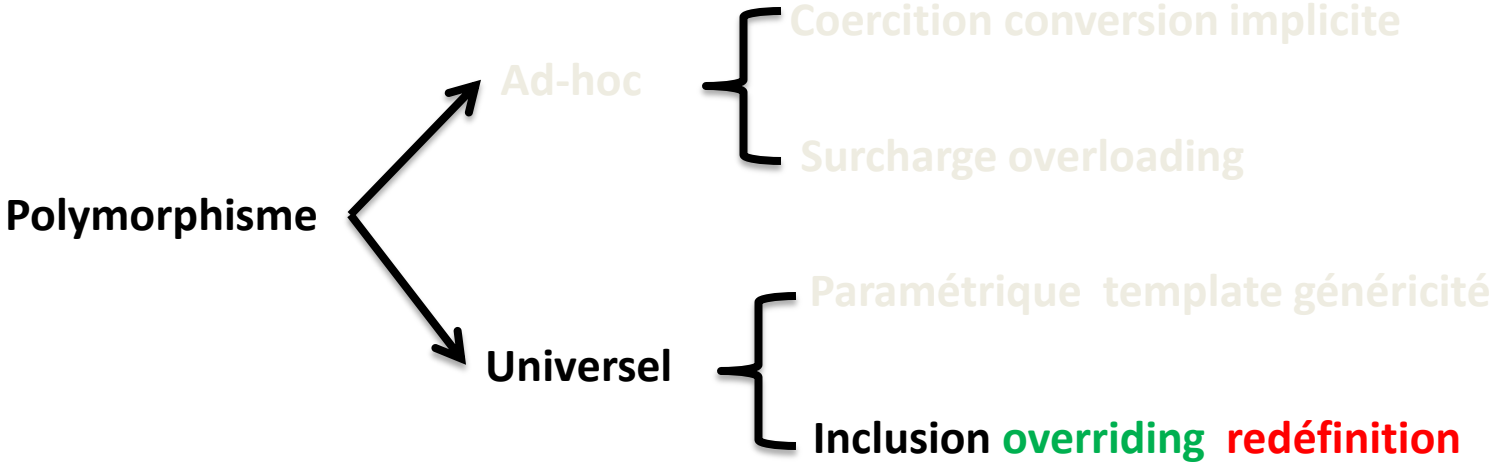
Généricité



Le polymorphisme paramétrique est le type de polymorphisme mis en œuvre dans les collections de Java et qui vous permet, par exemple, de spécifier les types des éléments d'un ArrayList.
En Java, on parle de « types génériques ».

```
ArrayList<Integer> liste = new ArrayList();
    liste.add(0);
    liste.add(1);
    liste.add(2);
    liste.add(3);
```

Polymorphisme d'inclusion



- Le polymorphisme par sous-typage est une forme de polymorphisme spécifique aux langages de programmation orientés objet puisqu'elle repose sur l'héritage et la redéfinition (overriding) de méthodes.
- Selon le principe de substitution de Liskov (LSP), il est permis d'invoquer une méthode avec des paramètres dont les types sont des sous-types des types attendus.
- Cela vaut pour les paramètres qui sont explicitement définis, mais cela vaut également pour l'objet sur lequel la méthode est invoquée.

Polymorphisme d'héritage

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

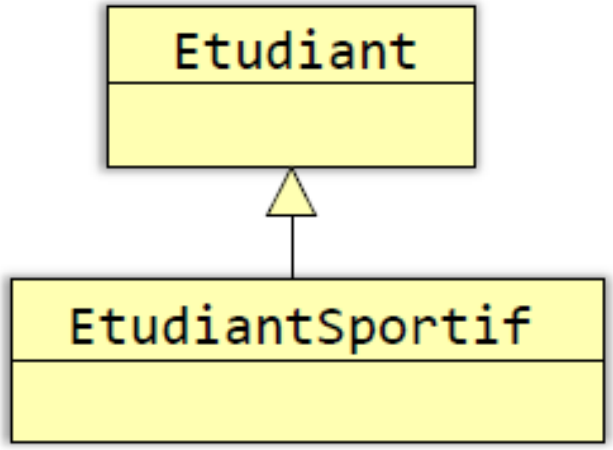
- En programmation par objets, on appelle polymorphisme
  - *le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.*
  - *le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.*

la mise en œuvre du polymorphisme

Surclassement et le liaison dynamique

Surclassement (Upcasting)

- 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 superclasse.
- 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.

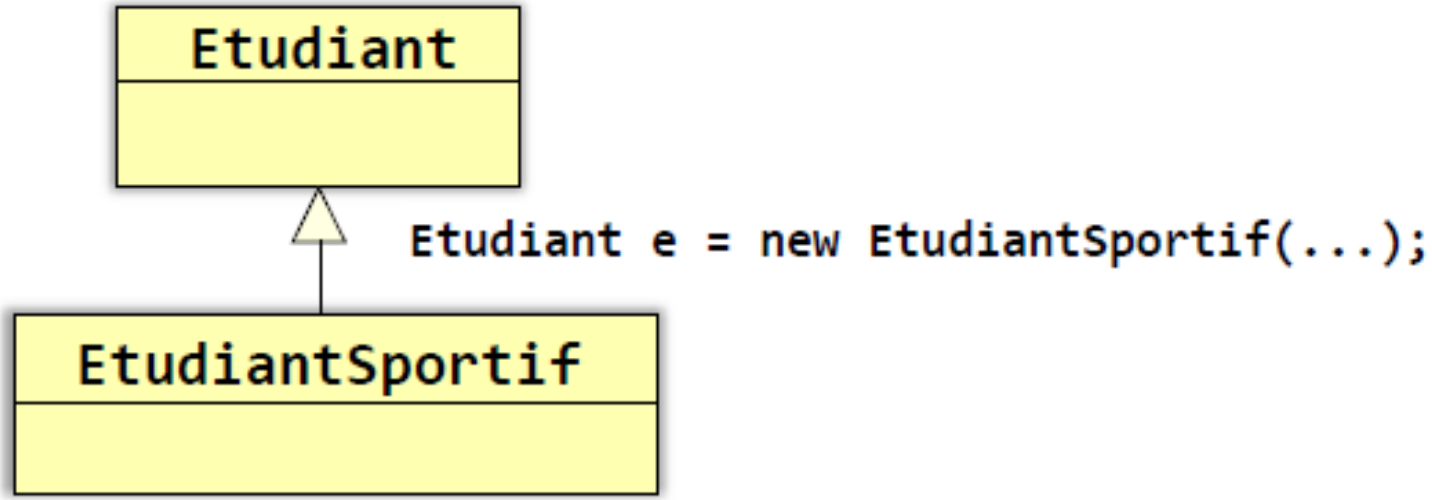


Un `EtudiantSportif` est un `Etudiant`

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

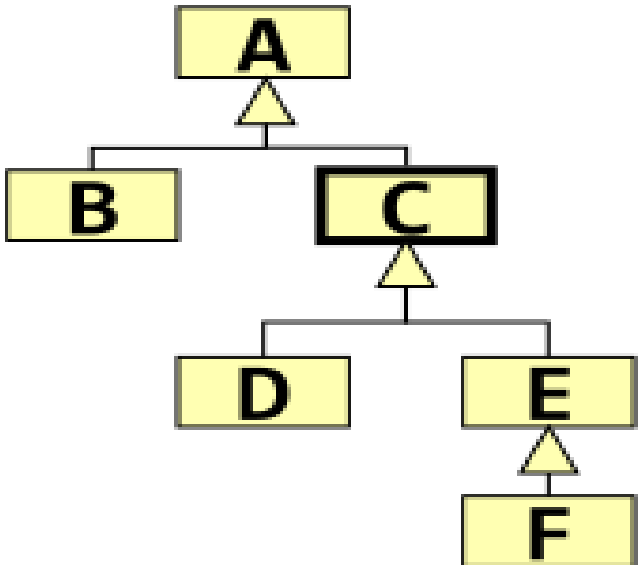
Surclassement (Upcasting)

- 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 (Upcasting)

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.

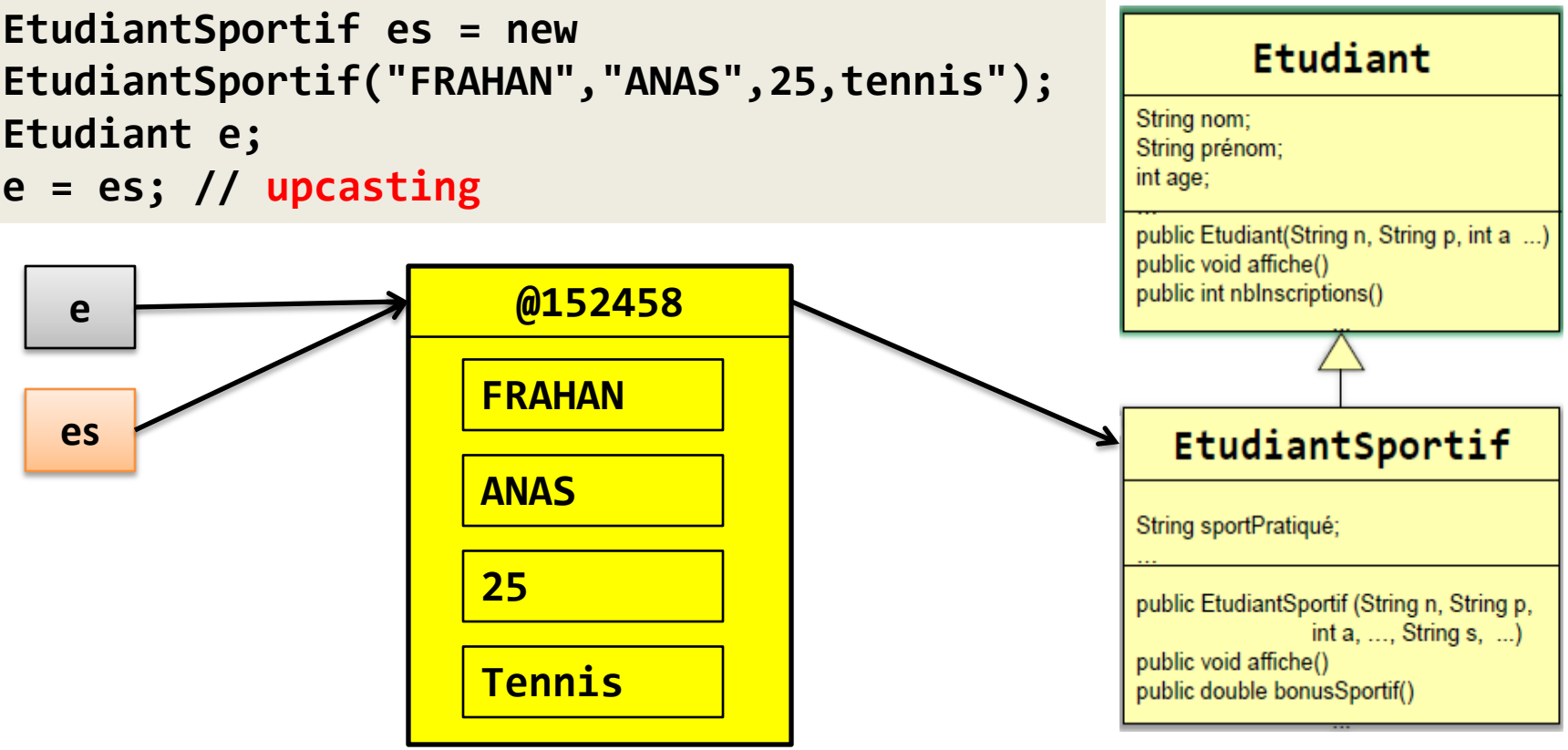



```
C c;
c = new D();
c = new E();
c = new F();
c = new A();
c = new B();
```

Surclassement (Upcasting)

- 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*

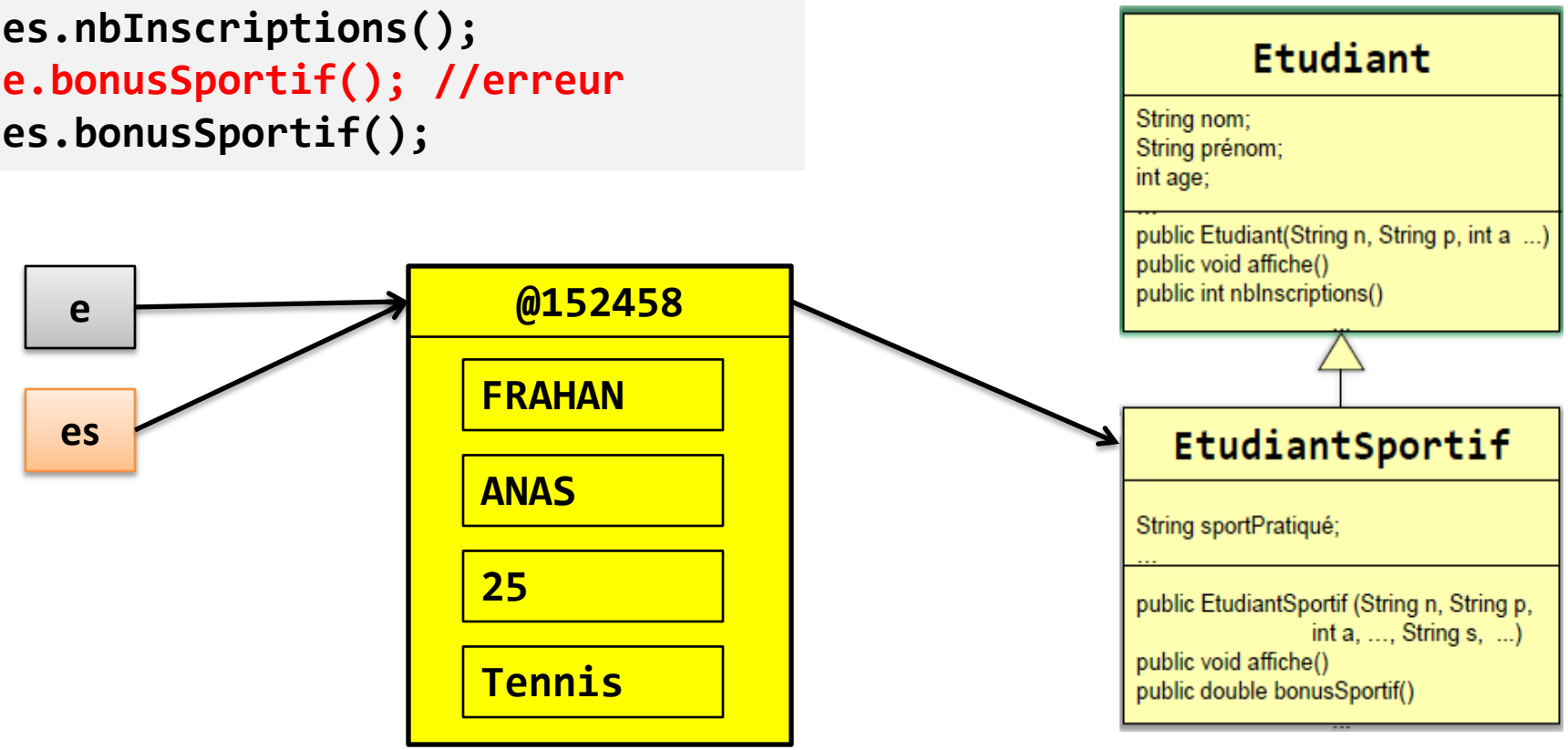
```
EtudiantSportif es = new
EtudiantSportif("FRAHAN","ANAS",25,tennis");
Etudiant e;
e = es; // upcasting
```



Surclassement (Upcasting)

```
e.affiche();
es.affiche();
e.nbInscriptions();
es.nbInscriptions();
e.bonusSportif(); //erreur
es.bonusSportif();
```

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



Upcasting : classe fille → classe mère

Définition

*On appelle **surclassement** ou **upcasting** le fait d'enregistrer une référence d'une instance d'une classe B héritant d'une classe A dans une variable de type A. En java, cette opération est implicite et constitue la base du polymorphisme.*

```
public class A { ... }

public class B extends A { ... }

A a = new B() // C'est de l'upcasting (surclassement).
```

On dit que a1 est une référence **surclassée** (elle est du type A et contient l'adresse d'une instance d'une sous-classe de A).

Downcasting : classe mère → classe fille

Définition

*On appelle **déclassement** ou **downcasting** le fait de convertir une référence « surclassée » pour « libérer » certaines fonctionnalités cachées par le surclassement. En java, cette conversion n'est pas implicite, elle doit être forcée par l'oppérateur de **cast** :* ***(<nomClasse>)***.

```
public class A { ... }

public class B extends A { ... }

A a = new B(); // surclassement, upcasting
B b = (B) a; // downcasting
```

Pour que la conversion fonctionne, il faut qu'à l'exécution le type réel de la référence à convertir soit B ou une des sous-classe de B !

Downcasting : classe mère → classe fille

Attention

Le downcasting ne permet pas de convertir une instance d'une classe donnée en une instance d'une sous-classe !

```
class A { A() {}}

class B extends A { B() {}}

A a = new A();
B b = (B) a;
```

Ça compile, mais ça plantera à l'exécution :

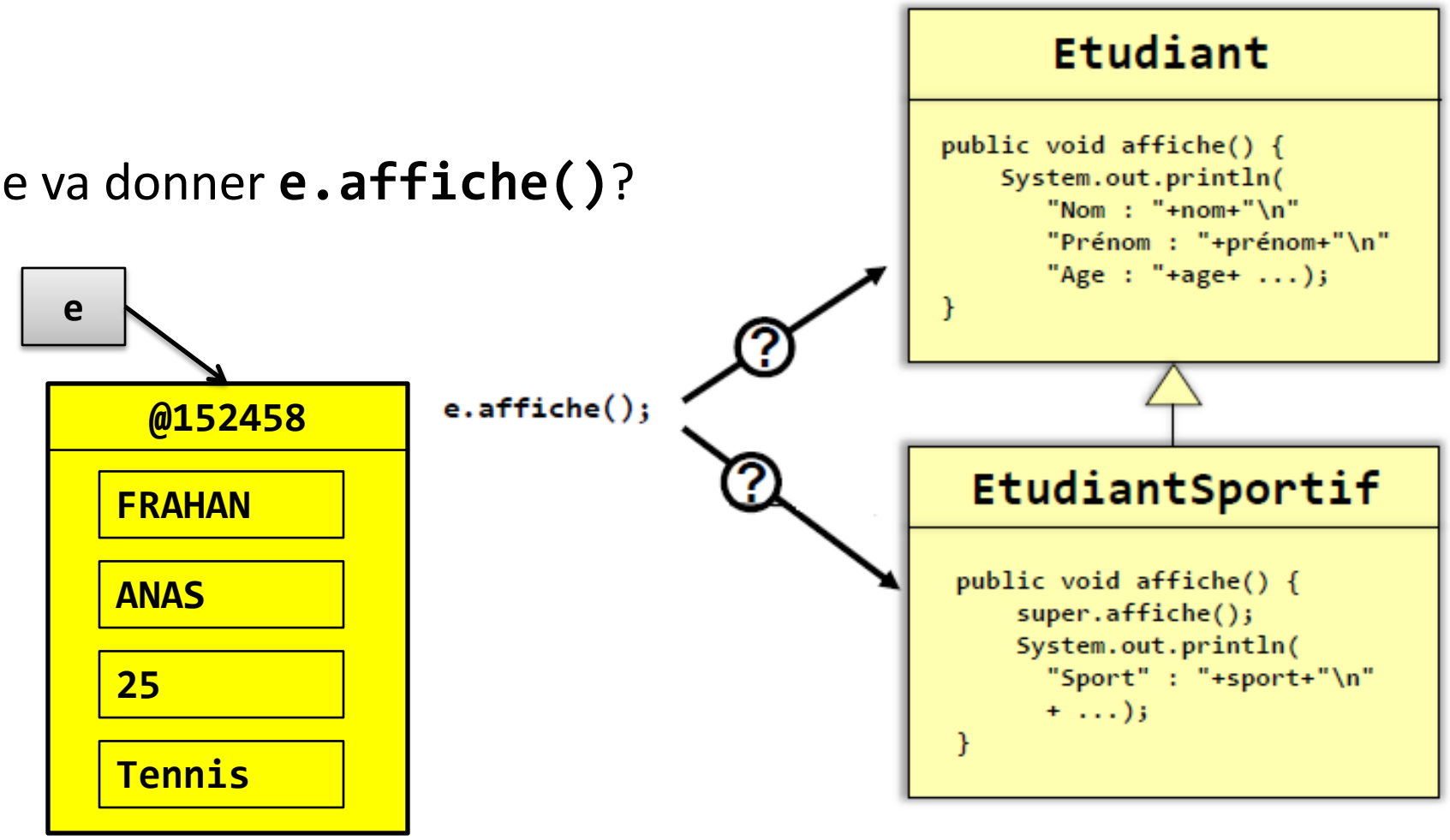
```
Exception in thread "main" java.lang.ClassCastException:
      A cannot be cast to B
```

Liaison dynamique

Résolution des messages

```
Etudiant e = new EtudiantSportif("FRAHAN","ANAS",25,"tennis");
```

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

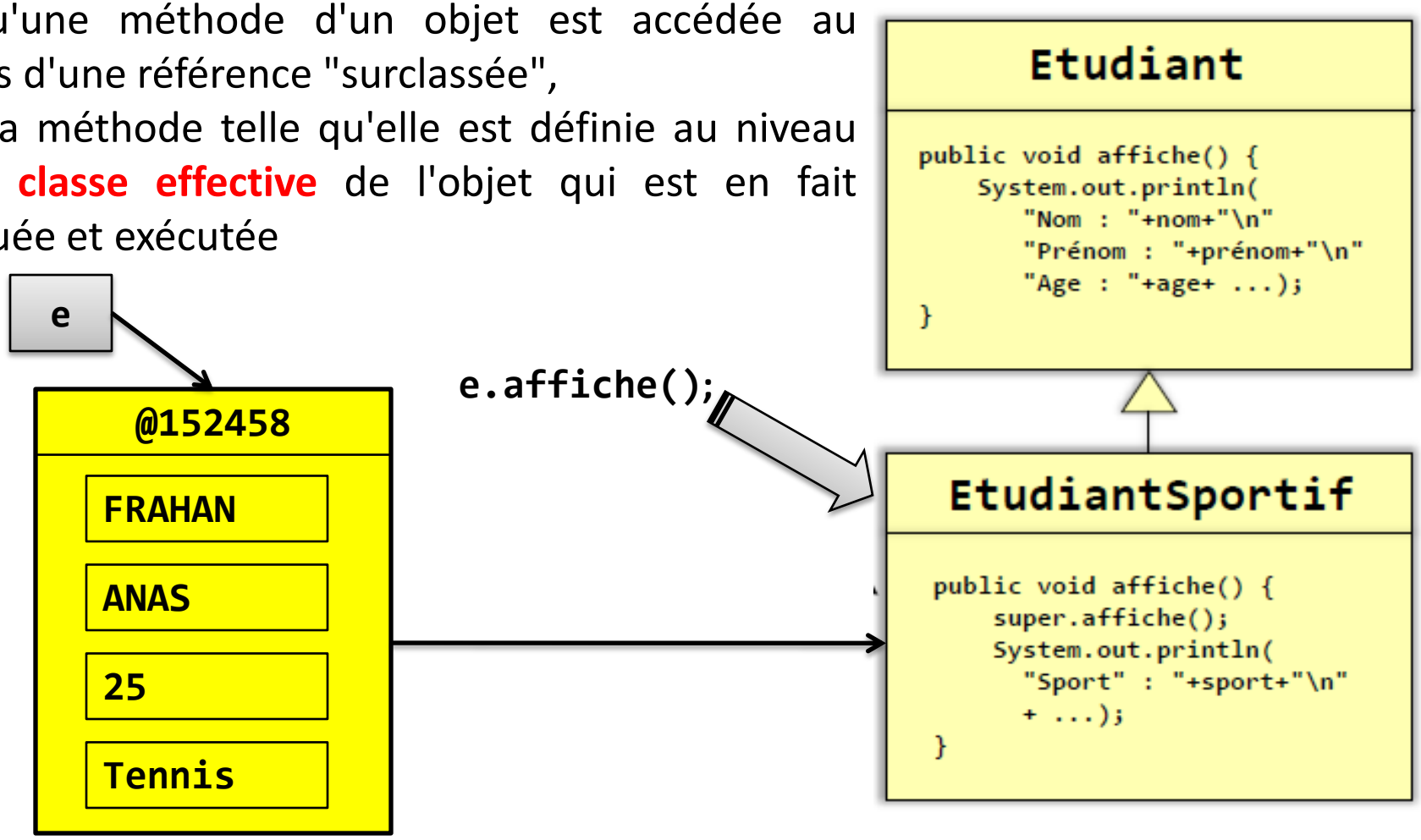


Liaison dynamique

Résolution des messages

```
Etudiant e = new EtudiantSportif("FRAHAN","ANAS",25,"tennis");
```

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

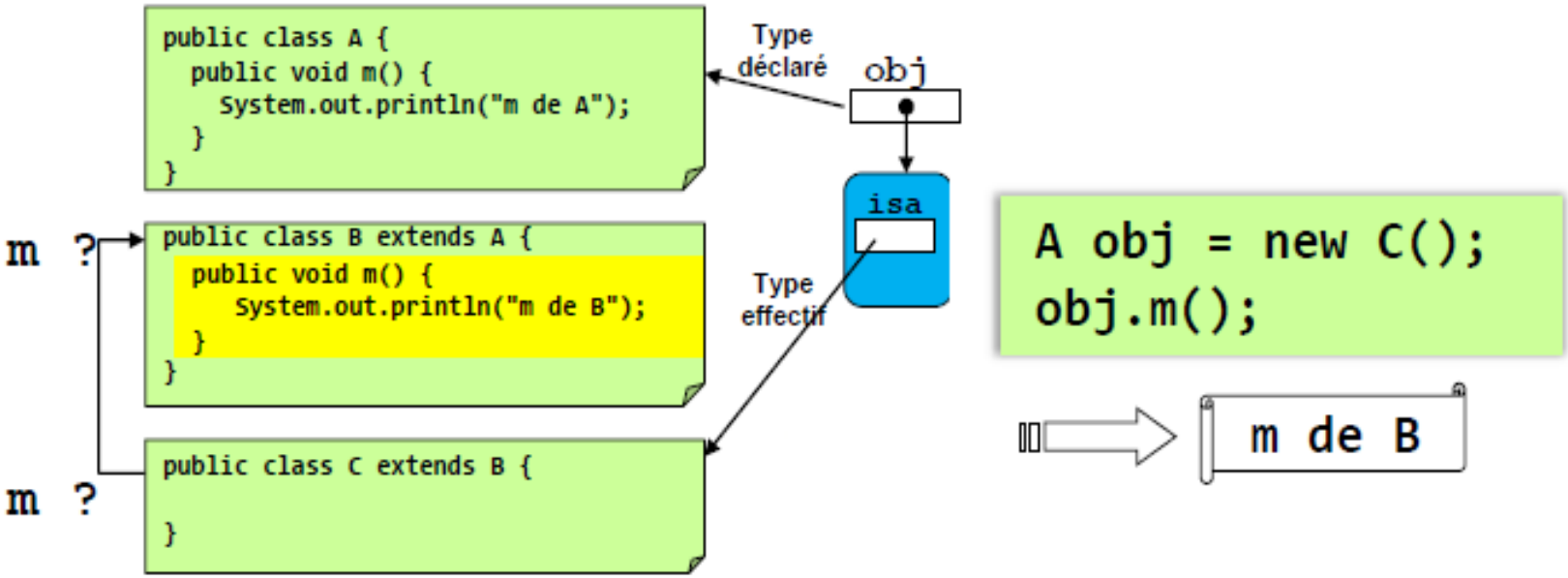


Liaison dynamique

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)

Liaison dynamique

Vérification statique

- **A la compilation:** seules des **vérifications statiques** qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées
- *l*a 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");
  }
}
```

Type déclaré `obj`

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

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

- à la compilation il n'est pas possible de déterminer le type exact de l'objet récepteur d'un message
- garantir dès la compilation que les messages pourront être résolus au moment de l'exécution --> **robustesse du code**

Liaison dynamique

Choix des méthodes, sélection du code

Le choix de la méthode à exécuter est effectué **statiquement à la compilation** en fonction du type des paramètres

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

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

```
invokevirtual … <Method m1()>
invokevirtual … <Method m1(int)>
invokevirtual … <Method m1()>
```

Byte-code

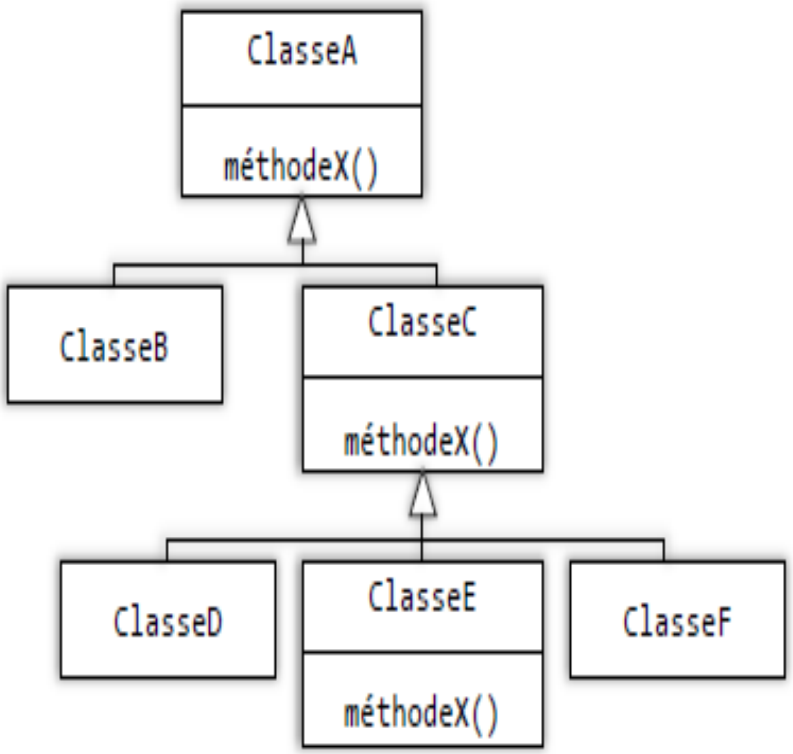
Compilation

```
A refA = new A();
refaA.m1();
refA.m1(10);
refA = new B();
refA.m1();
```

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

Polymorphisme d'inclusion

Upcasting + Liasion dynamique




```
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

Polymorphisme d'inclusion

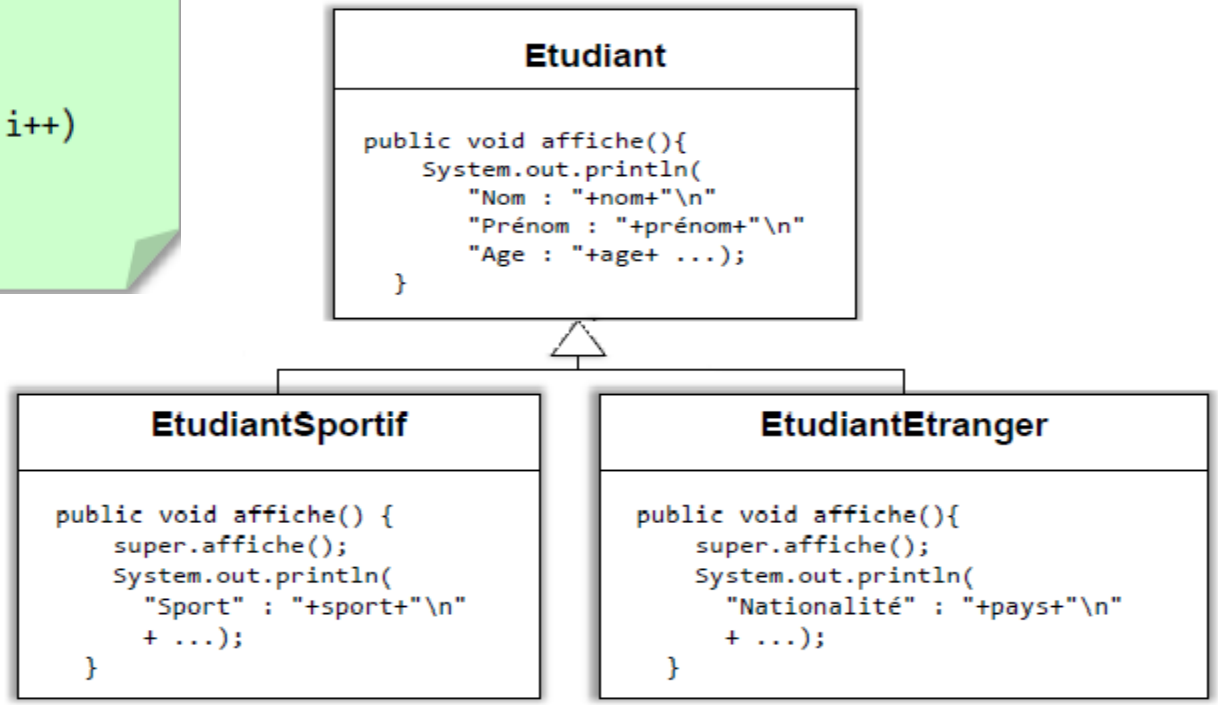
```
public class GroupeTD{

   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();
   }
 }
```

Liste peut contenir des étudiants de n'importe quel type

```
GroupeTD td1 = new GroupeTD();
td1.ajouter(new Etudiant("FARHAN", …));
td1.ajouter(new EtudiantSportif("BILAL",
"Ahmed", … , "ski");
```



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

Polymorphisme d'inclusion

- 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**

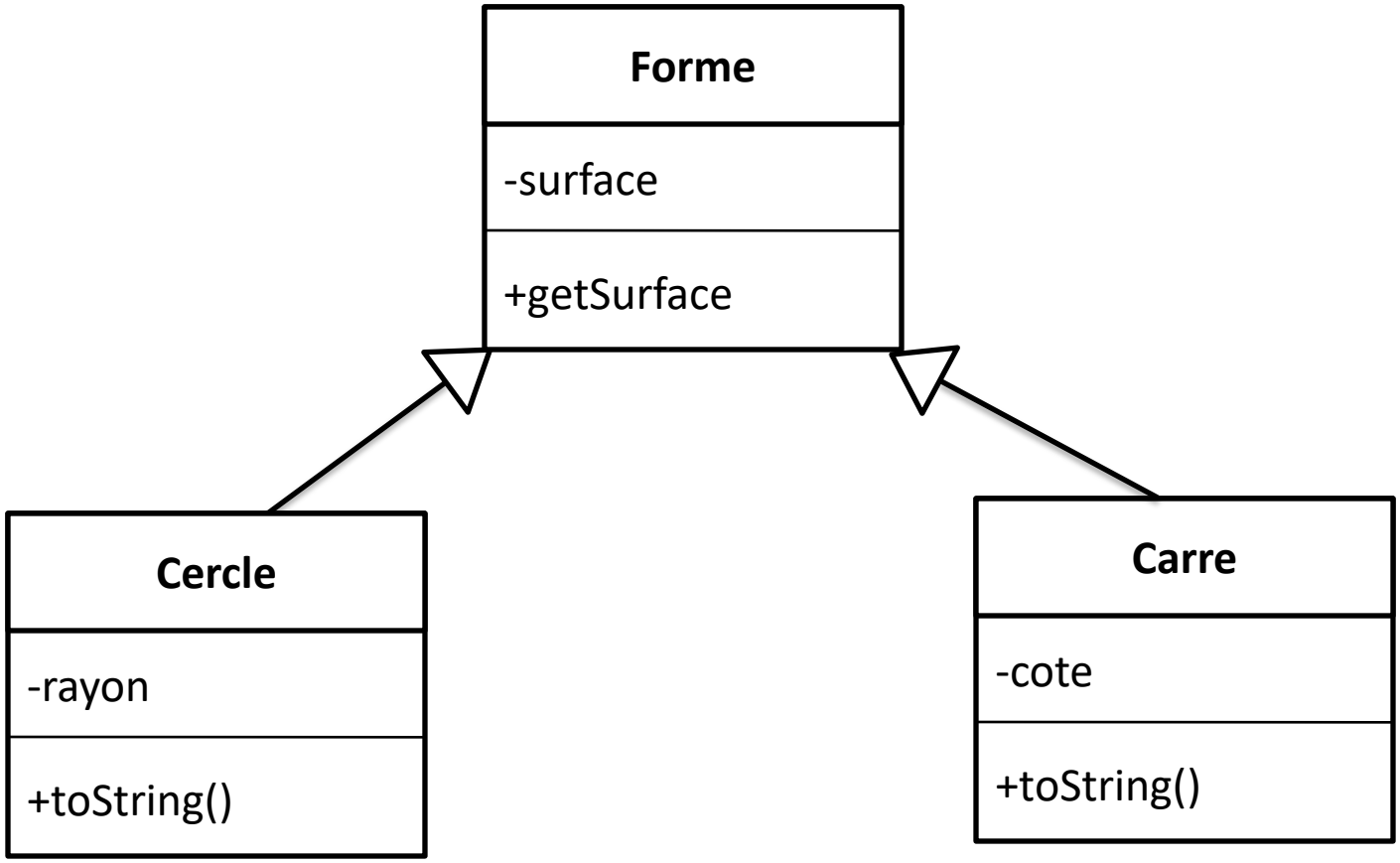
Exemple

```
public class TestFormes {

public static void main(String[] argv) {
    Forme[ ] figures = new Forme[3] ;
    figures [ 0 ] = new Carre( 2 ) ;    // Création d'un carré de 2 cm de coté
    figures [ 1 ] = new Cercle( 3 ) ;    // Création d'un cercle de 3 cm de rayon
    figures [ 2 ] = new Carre( 5.2 ) ;   // Création d'un carré de 5,2 cm de coté
    for( int i=0 ; i< figures.length ; i++ )
  System.out.println(
       figures[i] +" : surface = "+figures[i].getSurface()+"cm2") ;
  }

}
```

Exemple


Forme
-surface
+getSurface
Cercle
-rayon
+toString()
Carre
-cote
+toString()

Exemple

```
public class Forme {
    private double surface;
    public Forme(double surface) {
    this.surface=surface;
    }
    public double getSurface() {
    return surface;
    }
}
```

```
public class Carre extends Forme {
    private double cote;
    public Carre(double cote) {
    super(cote*cote);
    this.cote=cote;
    }
    @Override
    public String toString() {
    return "Carré(coté
    "+this.cote+"cm)";
    }}
```

```
public class Cercle extends
Forme {
    private double rayon;
    public Cercle(double
    rayon) {
    super(Math.PI *
    Math.pow(rayon, 2));
    this.rayon=rayon;
    }
    @Override
    public String toString()
    {
    return "Cercle(rayon
    "+this.rayon+"cm)";
    }
}
```