Java
Héritage
Et
Polymorphisme

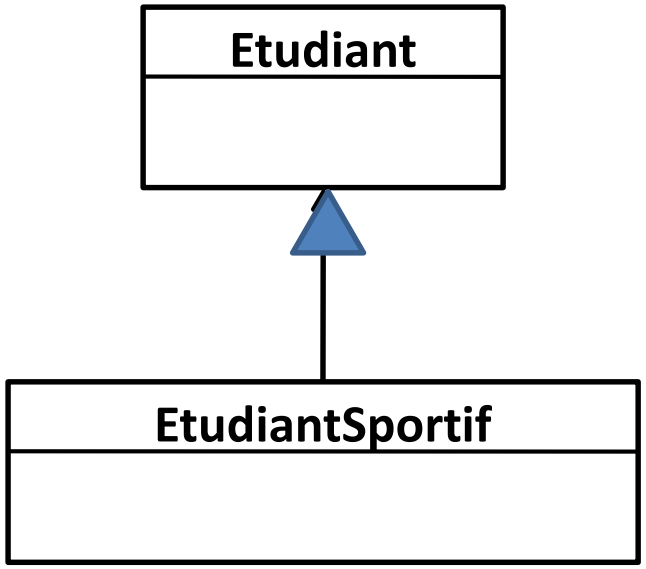
Préparé par Larbi Hassouni

Upcasting ou 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 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.



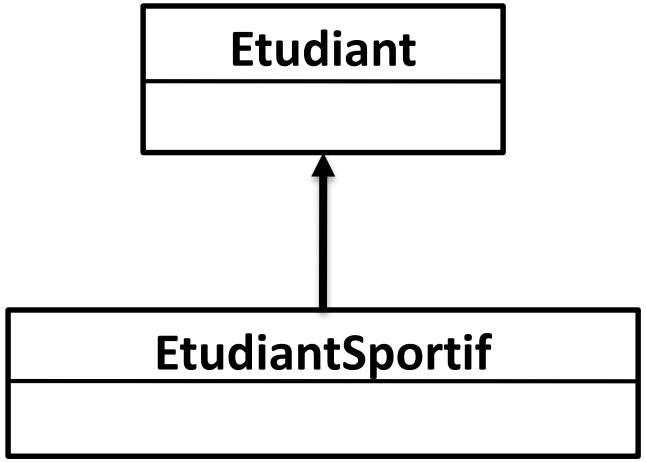
Un EtudiantSportif est un Etudiant

Un etudiant peut être un étudiant sportif

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

Upcasting ou Surclassement

- tout objet instance de la sous-classe B peut être aussi vu comme une instance de sa super-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* ***(upcasting)***



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

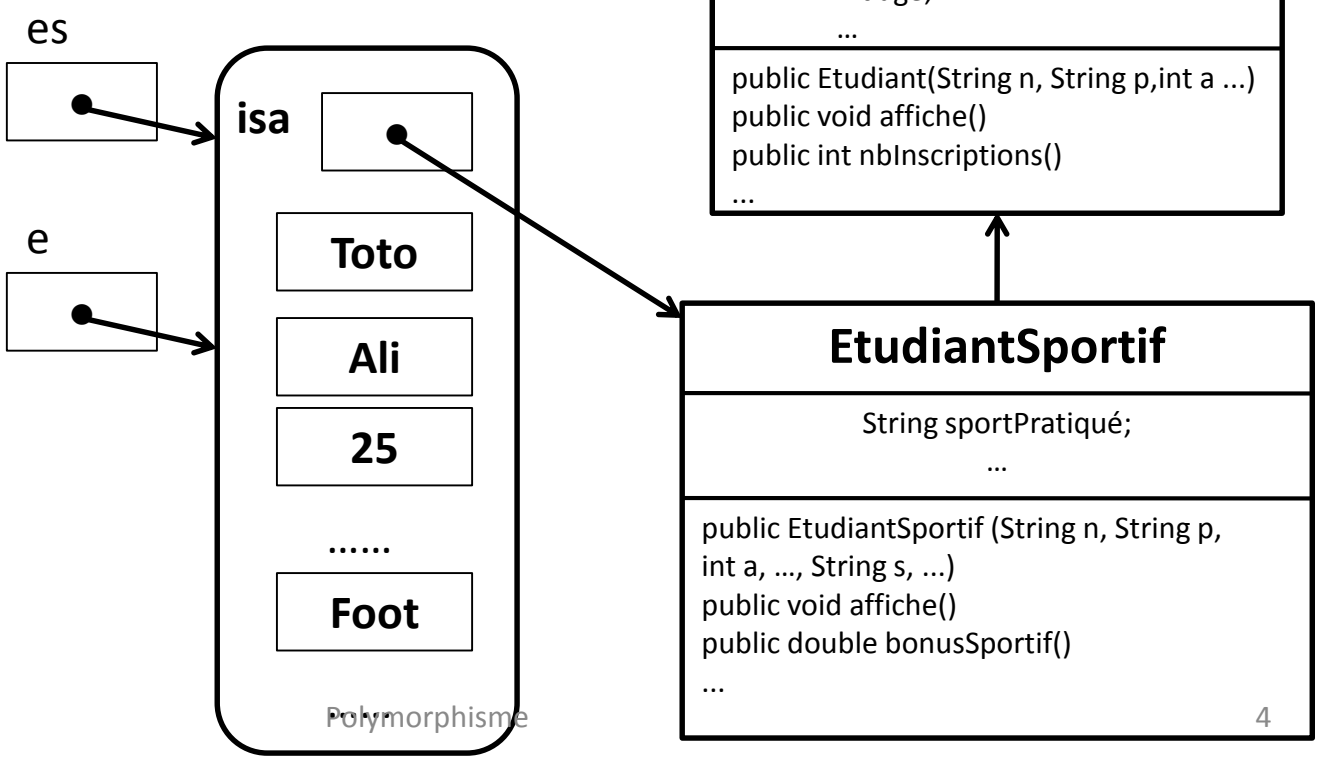
plus généralement à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte du type de la référence

Upcasting ou Surclassement

♦ **Lorsqu'un objet est "sur-classé" il est vu 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*

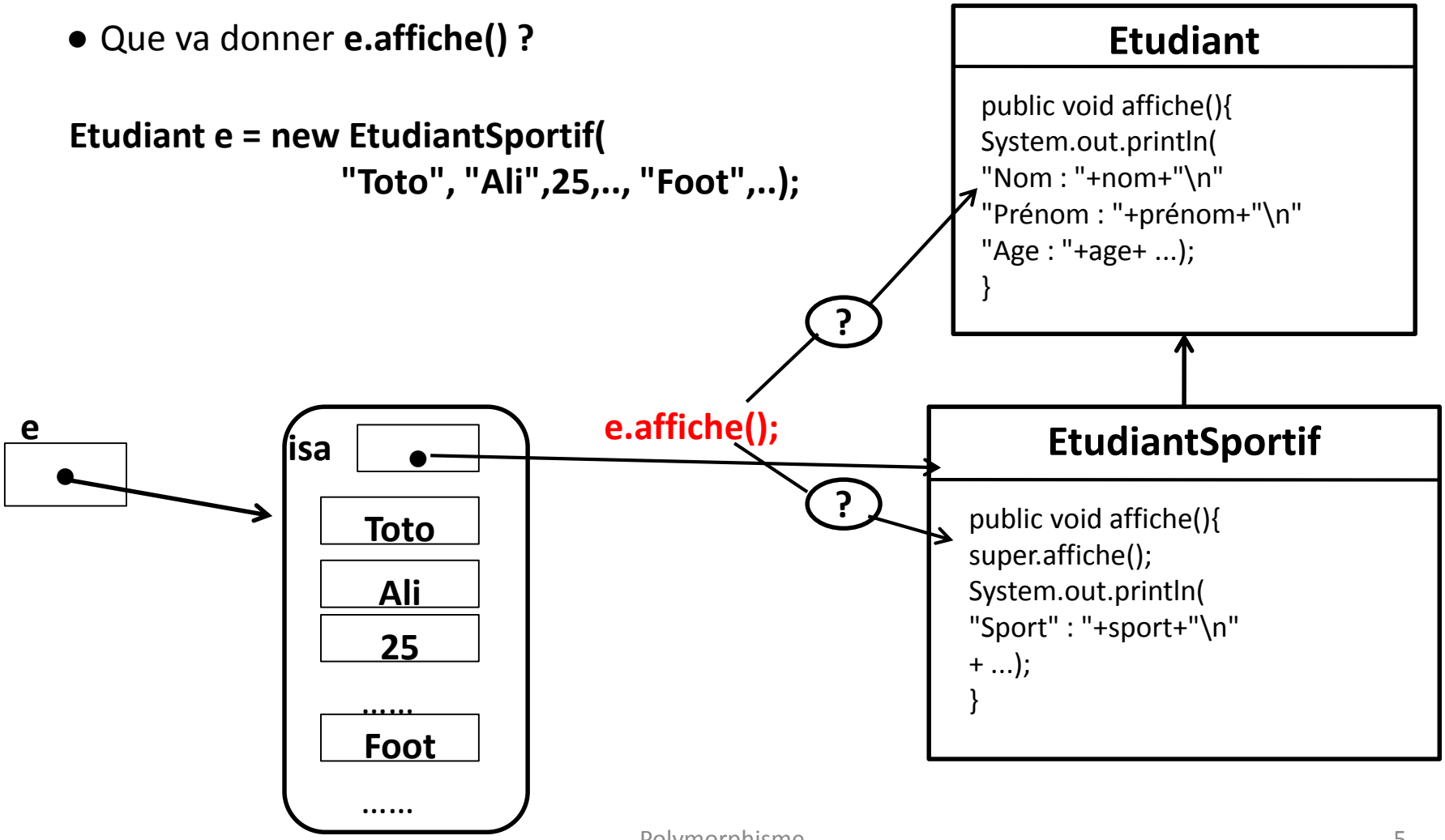
```
EtudiantSportif es;
es = new EtudiantSportif(« Toto",  »Ali",25,..,« Foot",..);

Etudiant e;
e = es; // upcasting

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

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

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



Lien dynamique
Résolution des messages

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

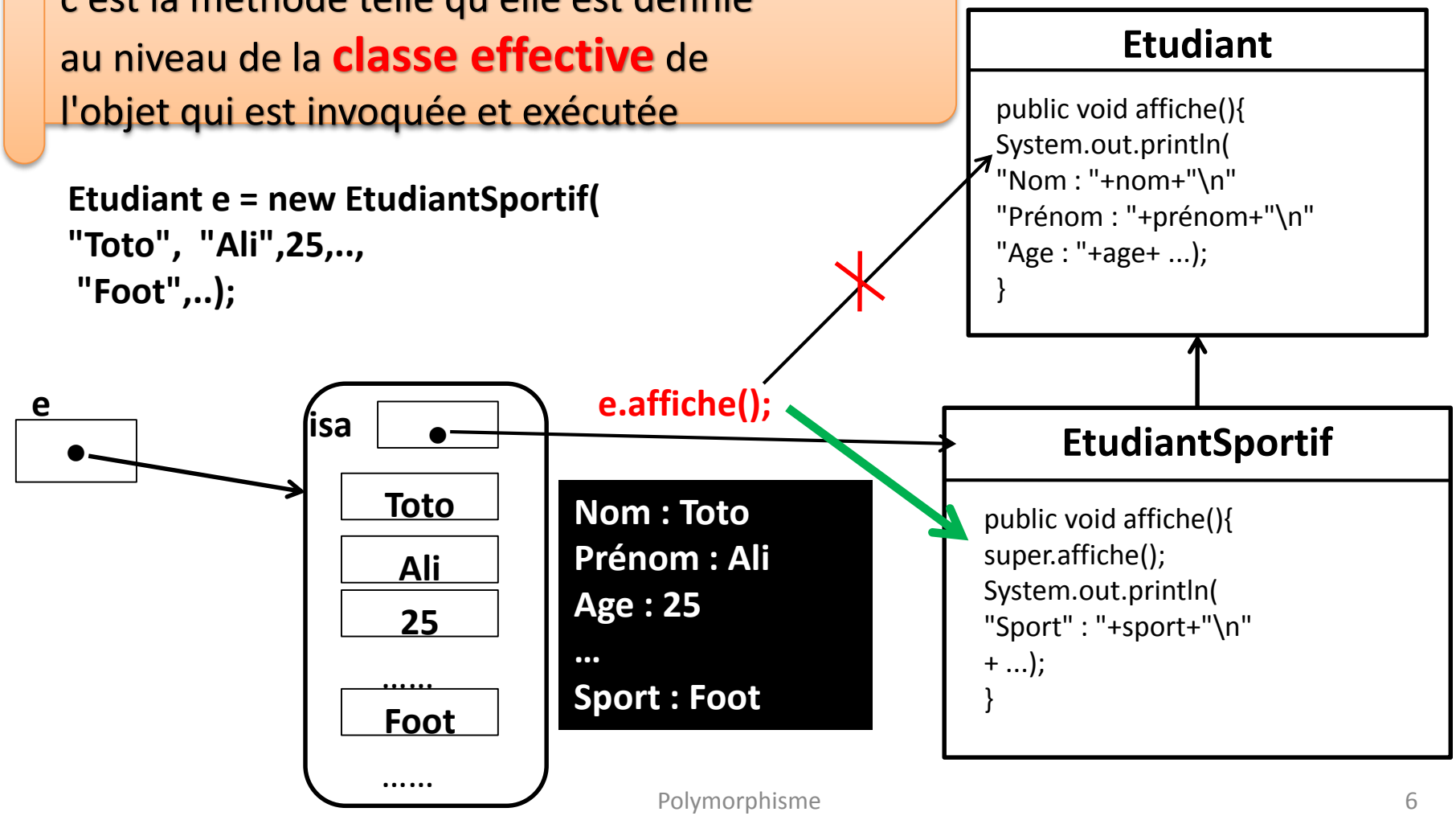
Etudiant e = new EtudiantSportif(
"Toto", "Ali",25,.., "Foot",..);



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 invoquée et exécutée

```
Etudiant e = new EtudiantSportif(
"Toto",  "Ali",25,..,
 "Foot",..);
```



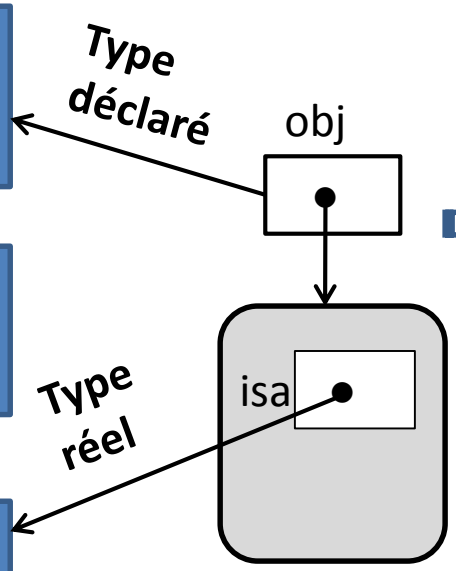
Lien dynamique
Mécanisme de résolution des messages

- 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).*

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

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

```
public class C extends B {
}
```




```
A obj = new C();
obj.m();
```

m de B

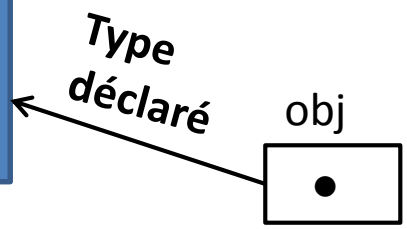
ce mécanisme est désigné sous le terme de **lien-dynamique (dynamic binding, latebinding** ou run-time binding)

Lien 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**

  ♦ *la classe déclarée de l'objet recevant le 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");
    }
}
```




```
A obj = new B();
obj.m1();
obj.m2();   //N'est pas définie
            //dans A
```

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

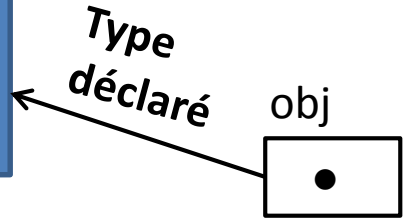
```
Test.java:21: cannot resolve symbol
symbol : method m2 ()
location: class A
obj.m2();
^
1 error
```

Lien dynamique
Vérifications statiques

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

```
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++){
   hasard = Math.random()
   if ( hasard < 0.5)
                obj = new A();
    else
                obj = new B();
    obj.m1();
}
```

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

Lien dynamique
Choix des méthodes, sélection du code

- La signature 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();
refA.m1(10);
~~refA.m2();~~
```

La sélection du code de la méthode à exécuter est effectué **dynamiquement à l'exécution en fonction du type effectif de l'objet récepteur du message**

Polymorphisme

A quoi servent l'upcasting et le lien dynamique ?

A la mise en oeuvre du polymorphisme

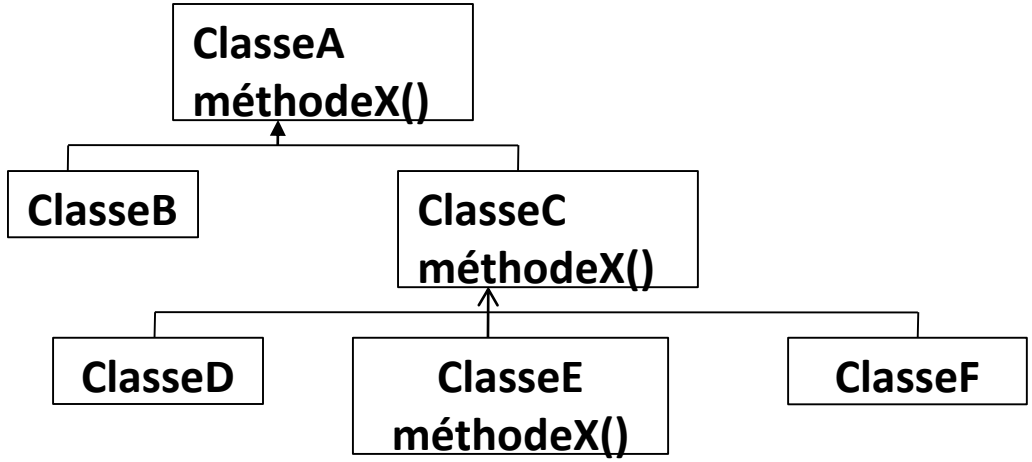
- Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

- En programmation Objet, 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();
```

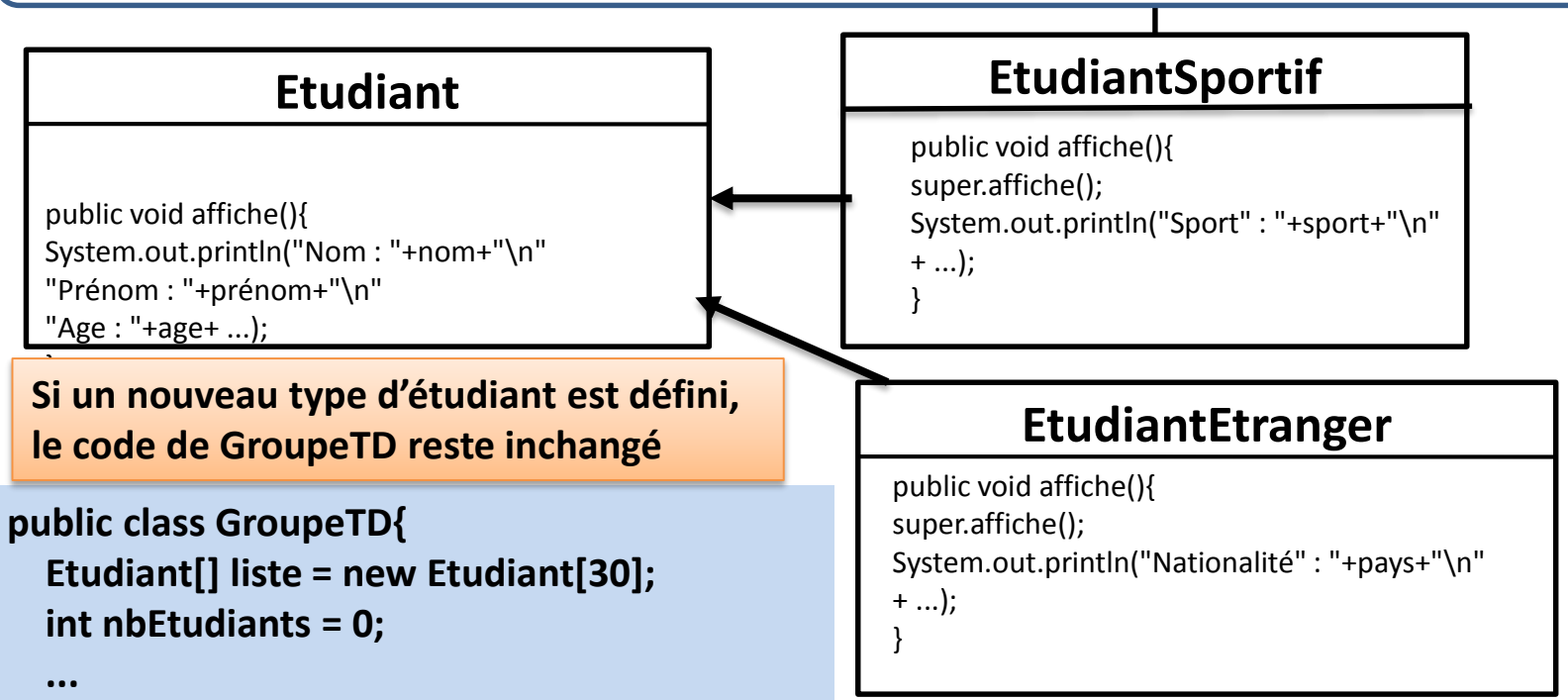
Surclassement
la référence peut désigner des objets de classe différente (n'importe quelle sous classe de ClasseA)

+

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

un cas particulier d'utilsation de Polymorphisme
est
manipulation uniforme des objets de plusieurs classes par l'intermédiaire d'une classe de base commune

Polymorphisme



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

```
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();
      }
}
```

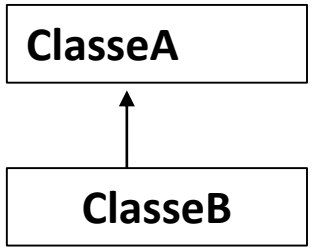
```
GroupeTD td1 = new GroupeTD();
td1.ajouter(new Etudiant(« MOHA", …));
td1.ajouter(new EtudiantSportif(« TOTO",
« ALI", … , « FOOT");
Td1.ajouter(new EtudiantEtranger("FOFO",
"NAJIB", …, "Algérie")
td1.afficherListe();
```

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 définissant 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**

Surcharge et Polymorphisme



Surcharge

```
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");
}
}
```

```
ClasseA refA = new ClasseA();
ClasseC.methodeX(refA);
ClasseB refB = new ClasseB();
ClasseC.methodeX(refB);
refA = refB; // upCasting
```

param TypeA

param TypeB

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

Redéfinition de equals

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

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

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

```
public class Point {
private double x;
private double y;
...
}
```

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

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

invokevirtual … <Method equals(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)

Redéfinition de equals

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

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

```
public boolean equals(Object o) {
   if (! (O instanceof Point))
     return false;
   Point pt = (Point) o; // Downcasting
   return (thix.x == pt.x  && this.y == pt.y)
```

```
public class Point {
private double x;
private double y;
...
}
```

redéfinit (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
```

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*

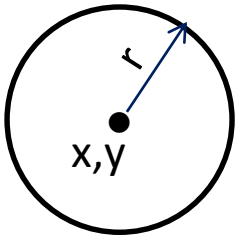
```
java.lang.ClassCastException: ClasseX
at Test.main(Test.java:52)
```

HERITAGE ET ABSTRACTION

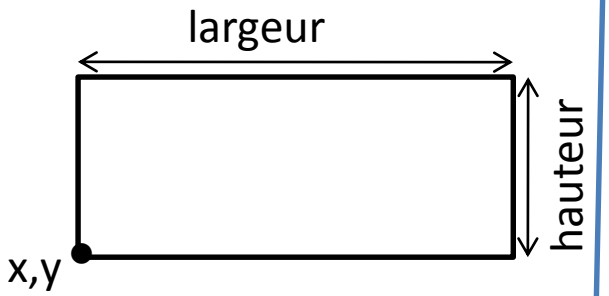
Classes abstraites

Classes abstraites : Exemple introductif

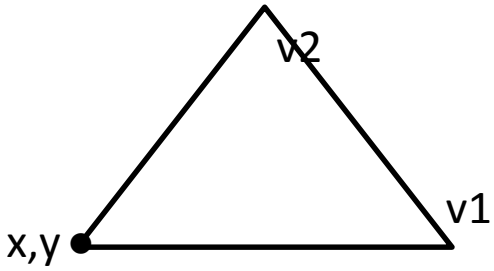
- **Formes géométriques**
  - ▲ *on veut définir une application permettant de manipuler des formes géométriques (triangles, rectangles, cercles…).*
  - ▲ *chaque forme est définie par sa position dans le plan*
  - ▲ *chaque forme peut être déplacée (modification de sa position), peut calculer son périmètre, sa surface*



Attributs :
double x,y; //centre du cercle
double r; // rayon
Méthodes :
deplacer(double dx, double dy)
double surface()
double périmètre()



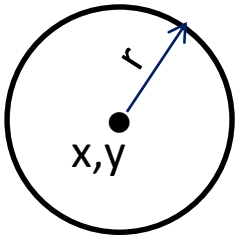
Attributs :
double x,y; //coin inférieur gauche
double largeur, hauteur;
Méthodes :
deplacer(double dx, double dy)
double surface()
double périmètre();

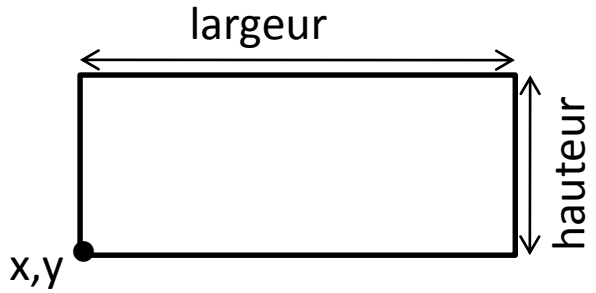


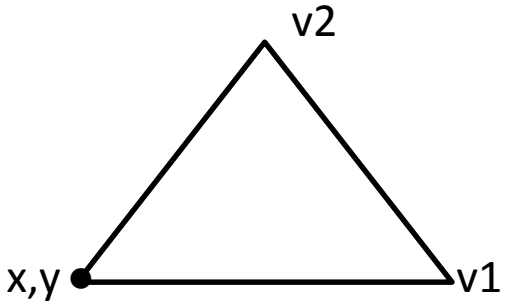
Attributs :
double x,y; //1 des sommets
double x1,y1; // v1
double x2,y2; // v2
Méthodes :
deplacer(double dx, double dy)
double surface()
double périmètre();

Comment factoriser le code ?

Classes abstraites : Exemple introductif

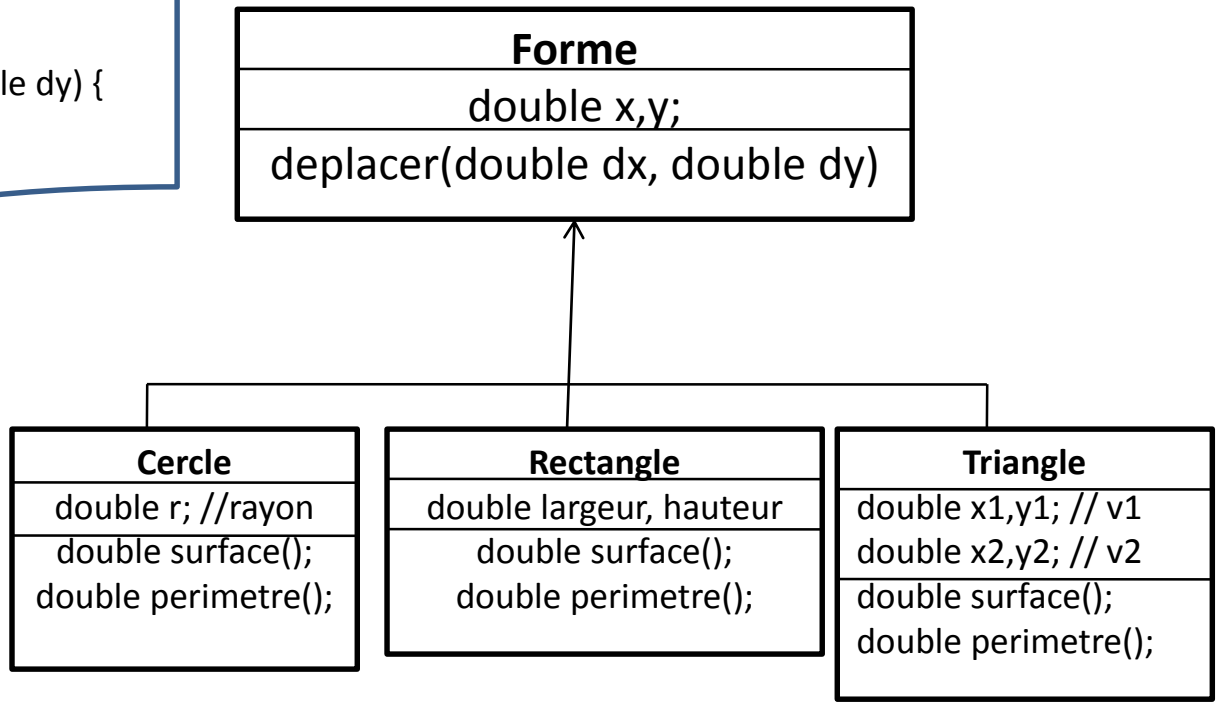







```
class Forme {
   protected double x,y;
   public void deplacer(double dx, double dy) {
      x += dx; y += dy;}}
```

```
class Cercle extends Forme {
   protected double r;
   public double surface(){
      return Math.PI * r * r;}
   protected double périmetre(){
      return 2 * Math.PI * r;}
}
```



Classes abstraites : Exemple introductif

```
public class ListeDeFormes {
  public static final int NB_MAX = 30;
  private Forme[] tabForme = new Forme[NB_MAX];
  private int nbFormes = 0;
  public void ajouter(Forme f){
    if (nbFormes < NB_MAX)
      tabForme[nbFormes++] =f;
  }
  public void toutDeplacer(double dx,double dy){
    for (int i=0; i < NbFormes; i++)
      tabFrome[i].deplace(dx,dy);
  }
  public double périmetreTotal(){
    double pt = 0.0;
    for (int i=0; i < NbFormes++; i++)
      pt += tabForme[i].périmetre();
    return pt;
  }
}
```

On veut pouvoir gérer des listes de formes

On exploite le **polymorphisme** la prise en compte de nouveaux types de forme ne modifie pas le code

Appel non valide car la méthode périmètre n 'est pas implémentée au niveau de la classe Forme

Définir une méthode périmètre dans Forme ? Comment?

```
public double périmetre(){
      return 0.0; // ou -1. ??
}
```

Une solution propre et élégante : les classes abstraites

Classes abstraites

- **Utilité :**
  - ♦ *définir des concepts incomplets qui devront être implémentés dans les sous classes*
  - ♦ *factoriser le code*

Classe abstraite

méthodes abstraites

```
public abstract class Forme{
   protected double x,y;
   public void deplacer(double dx, double dy) {
      x += dx; y += dy;
   }

   public abstract double périmètre() ;

   public abstract double surface();
}
```

Classes abstraites

- ***classe abstraite :*** classe non instanciable,
  c'est à dire qu'elle n'admet pas d'instances directes.
  - ♦ *Impossible de faire* ***new ClasseAbstraite(…);***

- ***méthode abstraite :*** méthode n'admettant pas d'implémentation
  - ♦ *au niveau de la classe dans laquelle elle est déclarée, on ne peut pas dire comment la réaliser.*

- Une classe pour laquelle au moins une méthode abstraite est déclarée est une classe abstraite (l'inverse n'est pas vrai).

- Les méthodes abstraites sont particulièrement utiles pour mettre en oeuvre le polymorphisme.

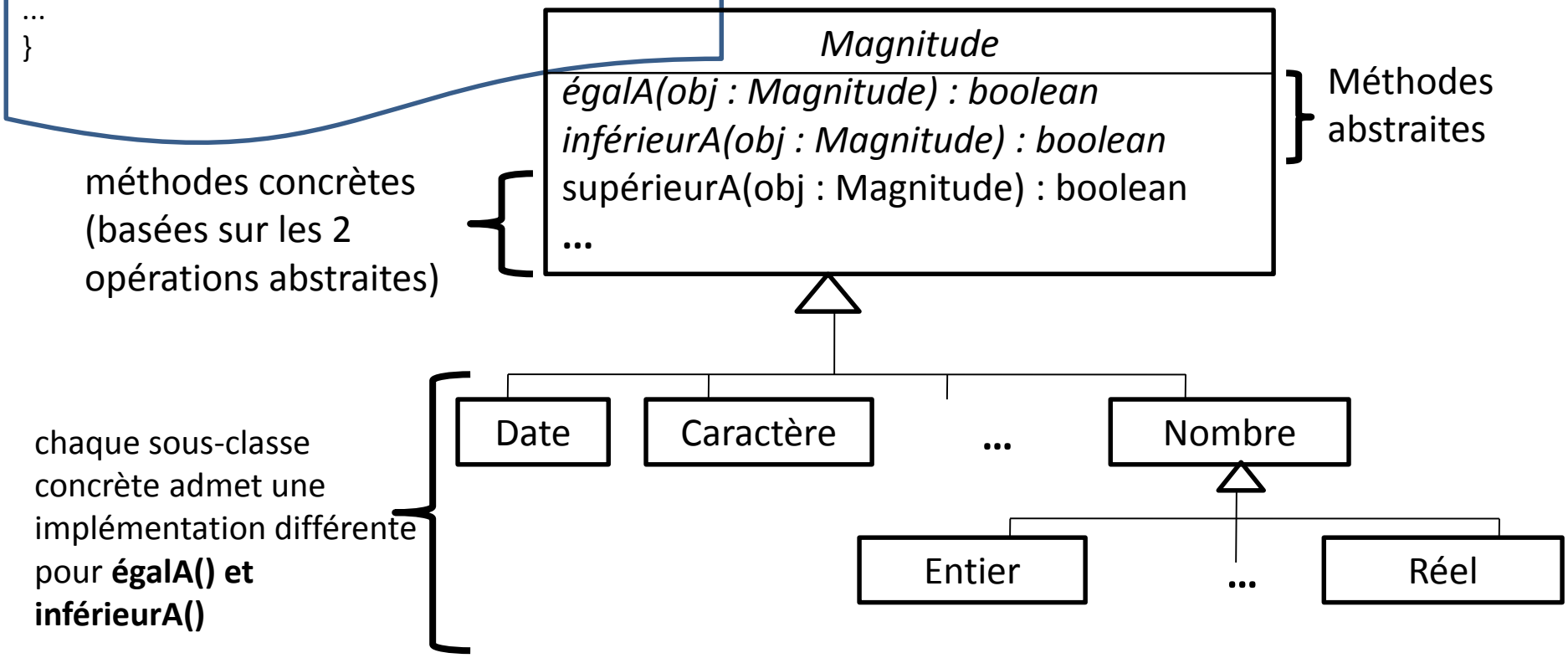
  - ♦ *l'utilisation du nom d'une classe abstraite comme type pour une (des) référence(s) est toujours possible (et souvent souhaitable !!!)*

Classes abstraites

- Une classe abstraite est une description d'objets destinée à être héritée par des classes plus spécialisées.
- Pour être utile, une classe abstraite doit admettre des classes descendantes ***concrètes.***
- Toute classe **concrète; sous-classe d'une classe abstraite doit** “concrétiser” toutes les opérations abstraites de cette dernière.
- Une classe abstraite permet de regrouper certaines caractéristiques communes à ses sous-classes et définit un comportement minimal commun.
- La factorisation optimale des propriétés communes à plusieurs classes par généralisation nécessite le plus souvent l'utilisation de classes abstraites.

Classes abstraites

```
public abstract class Magnitude {

  public abstract boolean egalA(Magnitude m) ;
  public abstract boolean inferieurA(Magnitude m) ;

  public boolean superieurA(Magnitude m) {
    return !egalA(m) && !inferieurA(m);
  }
...
}
```



HERITAGE ET ABSTRACTION

Interfaces

Interfaces : Exemple introductif

```
abstract class Animal {
   ...
   abstract void talk();
}
```

Bill Venners ***Designing with Interfaces One Programmer's Struggle to Understand the Interfaces http://www.atrima.com/designtechniques/index.html***

```
class Dog extends Animal {
   ...
   void talk() {
     System.out.println("Woof!");
   }
}
```

```
class Bird extends Animal {
   ...
   void talk() {
     System.out.println("Tweet");
}
}
```

```
class Cat extends Animal {
   ...
   void talk() {
     System.out.println("Meow");
   }
}
```

Polymorphisme signifie qu'une référence d'un type (classe) donné peut désigner un objet de n'importe quelle sous classe et selon la nature de cet objet produire un comportement différent

```
Animal animal1 = new Dog();
...
Animal animal2 = new Cat();
```

animal **peut être un Chien, un Chat ou** n'importe quelle sous classe d 'Animal

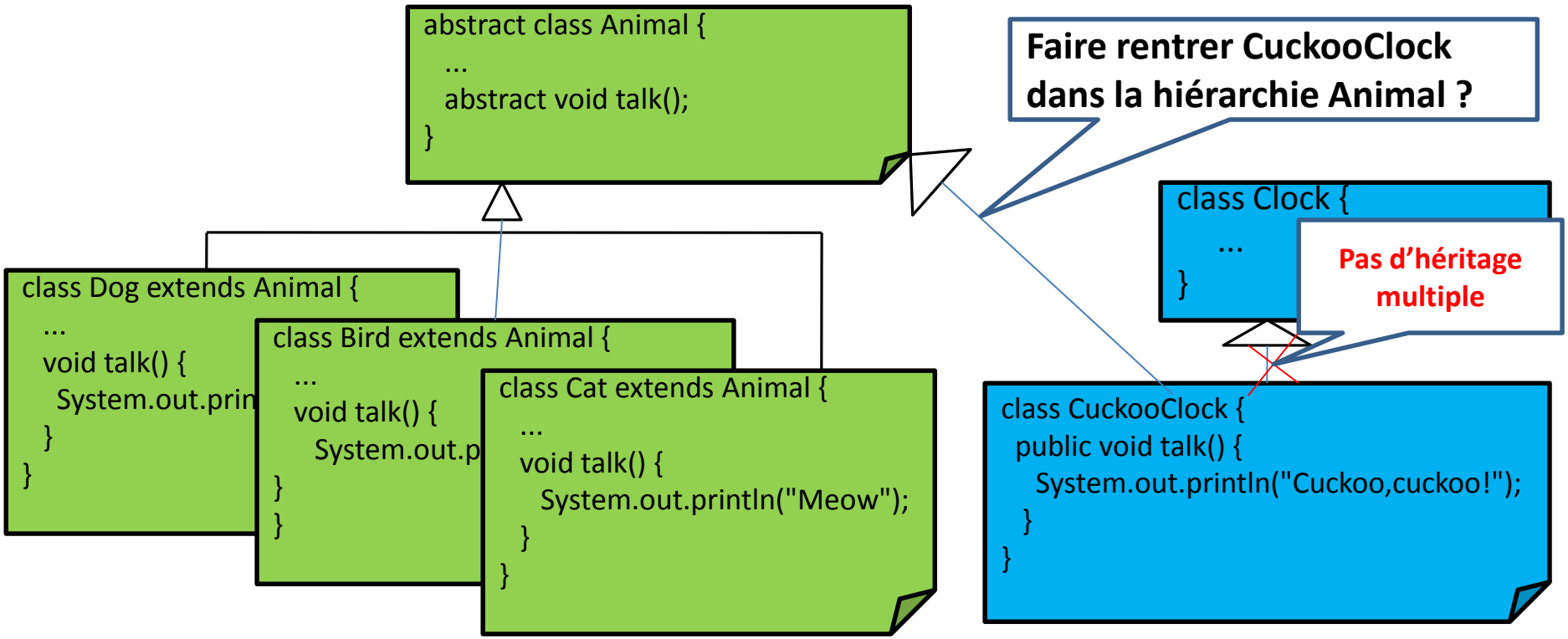
En JAVA le polymorphisme est rendu possible par la **liaison dynamique (*dynamic binding)***

```
class Interrogator {
   static void makeItTalk(Animal subject) {
     subject.talk();
   }
}
```

JVM **décide à l'exécution (*runtime) quelle méthode*** invoquer en se basant sur la classe de l'objet

Interfaces : Exemple introductif

Comment utiliser **Interrogator pour faire parler aussi un CuckooClock ?**



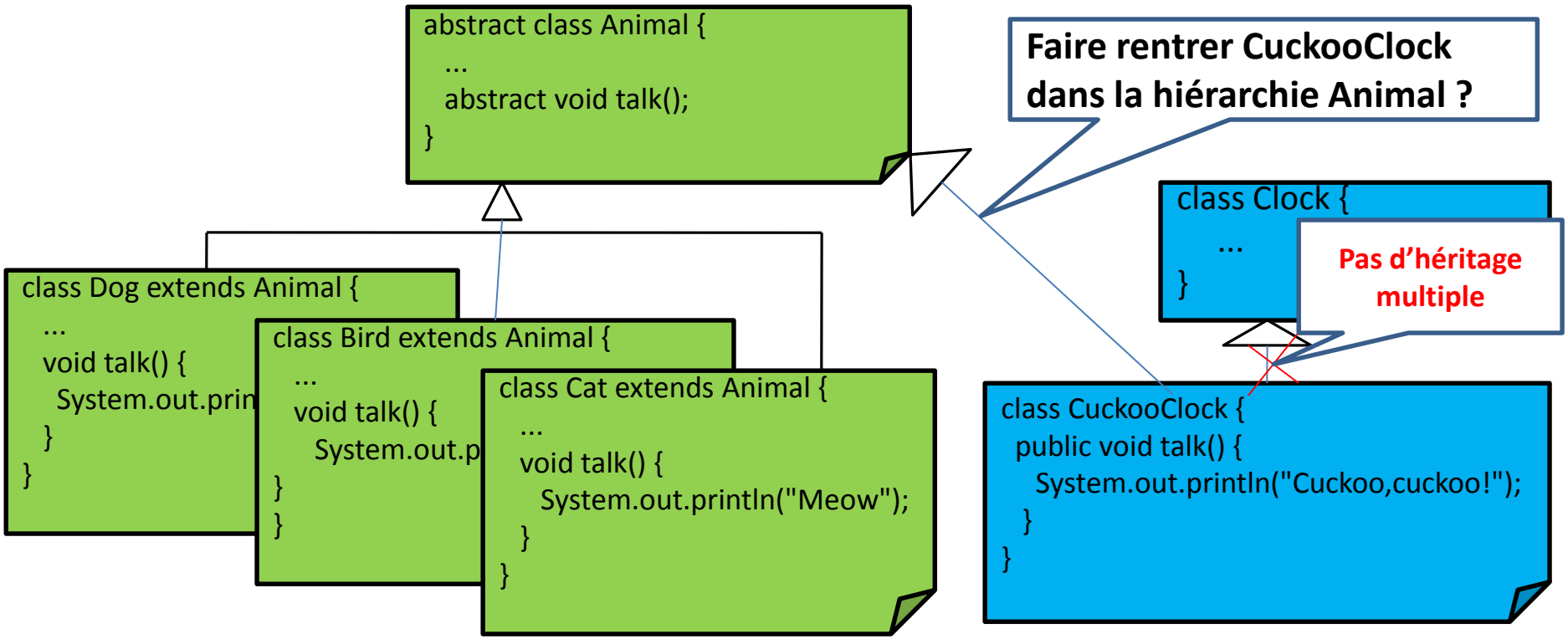

```
class Interrogator {
  static void makeItTalk(Animal subject) {
    subject.talk();
  }
}
```

```
class CuckooClockInterrogator {
  static void makeItTalk(Cuckooclock subject) {
    subject.talk();
  }
}
```

Se passer du polymorphisme ?

Interfaces : Exemple introductif

Comment utiliser **Interrogator pour faire parler aussi un CuckooClock ?**



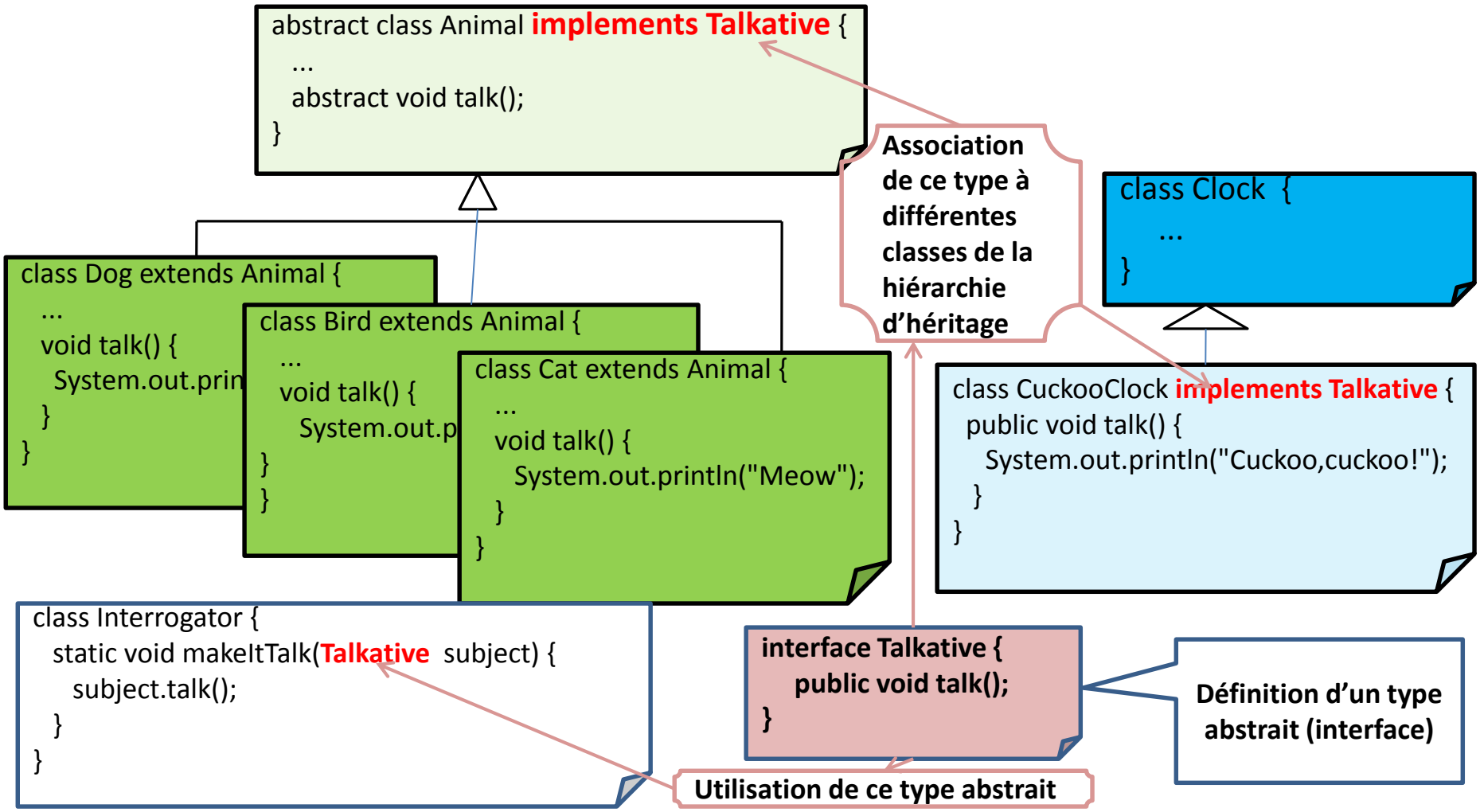

```
class Interrogator {
  static void makeItTalk(Animal subject) {
    subject.talk();
  }
}
```

```
class CuckooClockInterrogator {
  static void makeItTalk(Cuckooclock subject) {
    subject.talk();
  }
}
```

Se passer du polymorphisme ?

Interfaces : Exemple introductif

Comment utiliser **Interrogator pour faire parler aussi un CuckooClock ?**



Les interfaces permettent **plus de polymorphisme car avec les interfaces il** n'est pas nécessaire de tout faire rentrer dans une seule famille (hiérarchie) de classes

Interfaces

- Java's interface gives you more polymorphism than you can get with singly inherited families of classes, without the "burden" of multiple inheritance implementation.

Bill Venners *Designing with Interfaces - One Programmer's Struggle to Understand the Interface*
http://www.atrima.com/designtechniques/index.html

Interfaces : Déclaration d'une interface

- Une *interface est une collection de méthodes utilisée pour spécifier un* service offert par une classe.

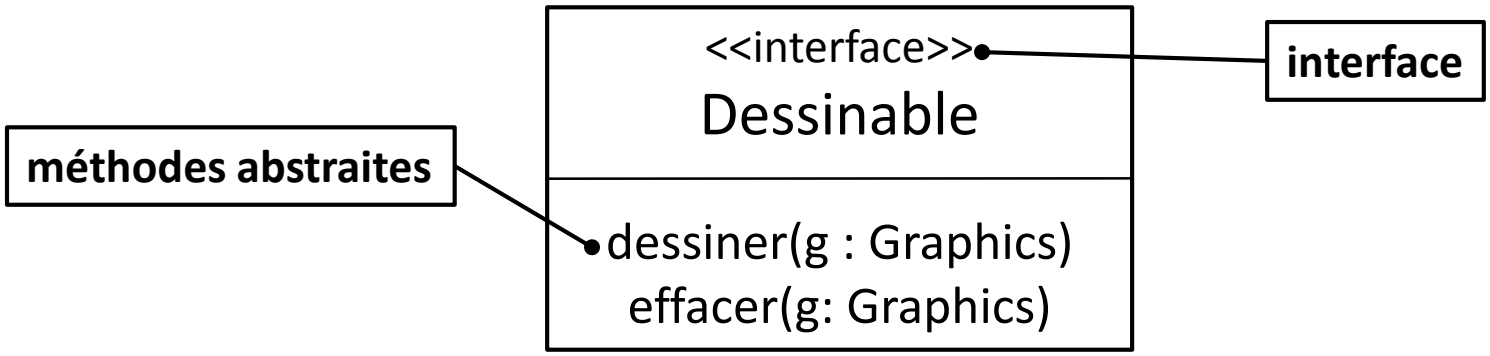
- Une interface peut être vue comme une classe abstraite sans attributs et dont toutes les méthodes sont abstraites.

```
import java.awt.*;
public interface Dessinable {
   public void dessiner(Graphics g);
   void effacer(Graphics g);
}
```

Dessinable.java

- Toutes les méthodes sont abstraites
- Elles sont implicitement publiques

- Possibilité de définir des attributs à condition qu'il s'agisse d 'attributs de type primitif
- Ces attributs sont implicitement déclarés comme **static final**

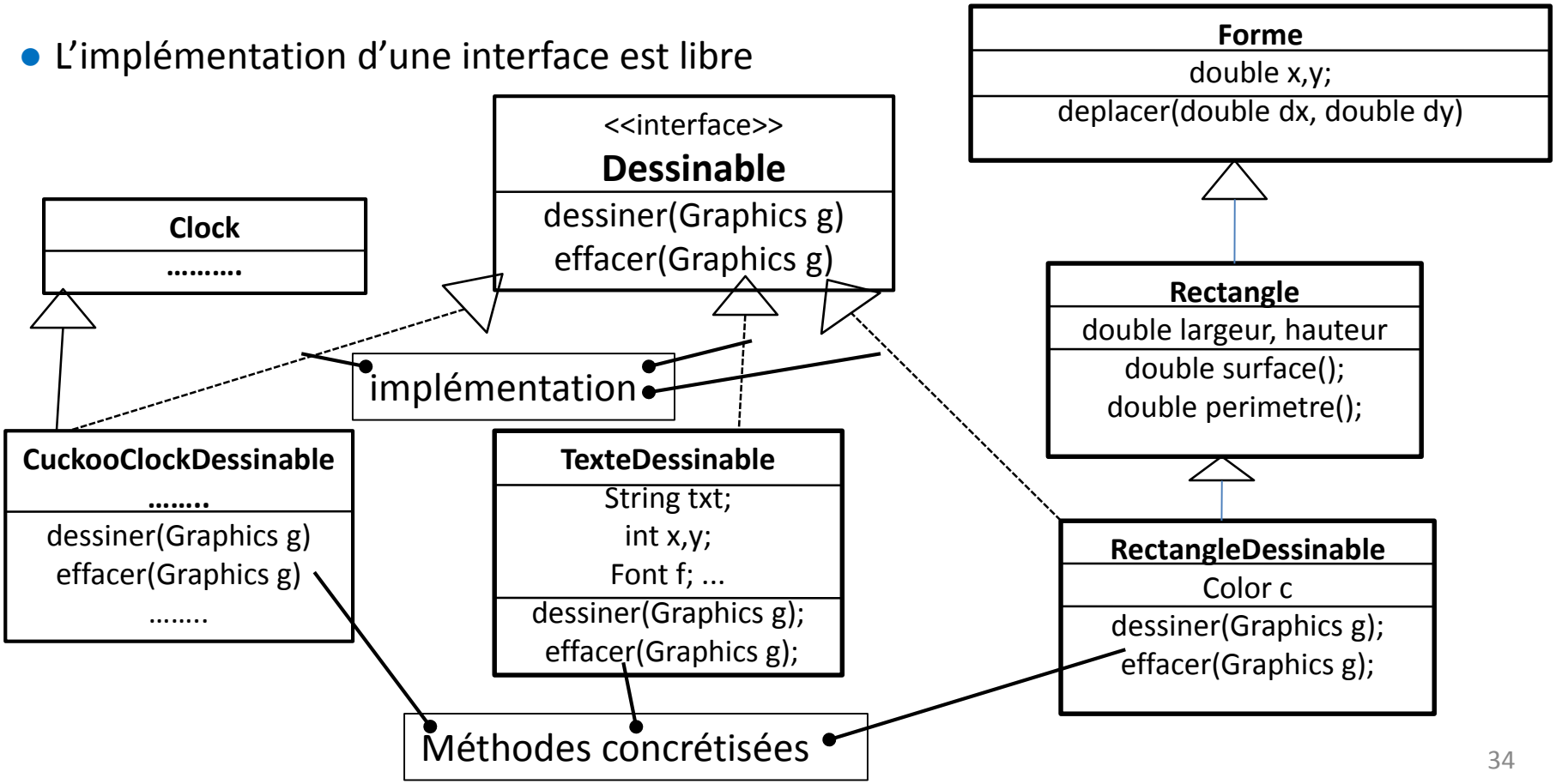


Interfaces : Implémentation d'une interface

- Une interface est destinée à être *"réalisée" (implémentée) par d'autres* classes (celles-ci en héritent toutes les descriptions et concrétisent les opérations abstraites).

- *Les classes réalisantes **s'engagent à** fournir le service spécifié par l'interface*

- L'implémentation d'une interface est libre



Interfaces : Implémentation d'une interface

- De la même manière qu'une classe étend sa super-classe elle peut de manière **optionnelle** implémenter une ou plusieurs interfaces
- *dans la définition de la classe, après la clause **extends nomSuperClasse**, faire apparaître explicitement le mot clé **implements suivi du nom de l'interface** implémentée*

```
class RectangleDessinable extends Rectangle implements Dessinable {

  public void dessiner(Graphics g){
    g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
  }

  public void effacer(Graphics g){
    g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
  }
}
```

*si la classe est une classe concrète **elle doit fournir une implémentation (un corps) à chacune** des méthodes abstraites définies dans l'interface (qui doivent être déclarées publiques)*

Interfaces : Implémentation d'une interface

- Une classe JAVA peut implémenter simultanément plusieurs interfaces
  - *Pour cela la liste des noms des interfaces à implémenter séparés par des virgules doit suivre le mot clé implements*

```
class RectangleDessinable extends Rectangle implements Dessinable,Comparable {

  public void dessiner(Graphics g){
      g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
   }
   public void effacer(Graphics g){
      g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
   }

   public int compareTo(Object o) {
      if (o instanceof Rectangle)
            ...
   }
}
```

Méthodes de l'interface Comparable

Méthodes de l'interface Dessinable

Interfaces et polymorphisme

- Une interface peut être utilisée comme un type
  - *A des variables (références) dont le type est une interface il est possible d'affecter des instances de toute classe implémentant l'interface, ou toute sous-classe d'une telle classe.*

```
public class Fenetre {
  private nbFigures;
  private Dessinable[] figures;
  ...
  public void ajouter(Dessinable d){
    ...
  }
  public void supprimer(Dessinable o){
    ...
  }
  public void dessiner() {
    for (int i = 0; i < nbFigures; i++)
      figures[i].dessiner(g);
  }
}
```

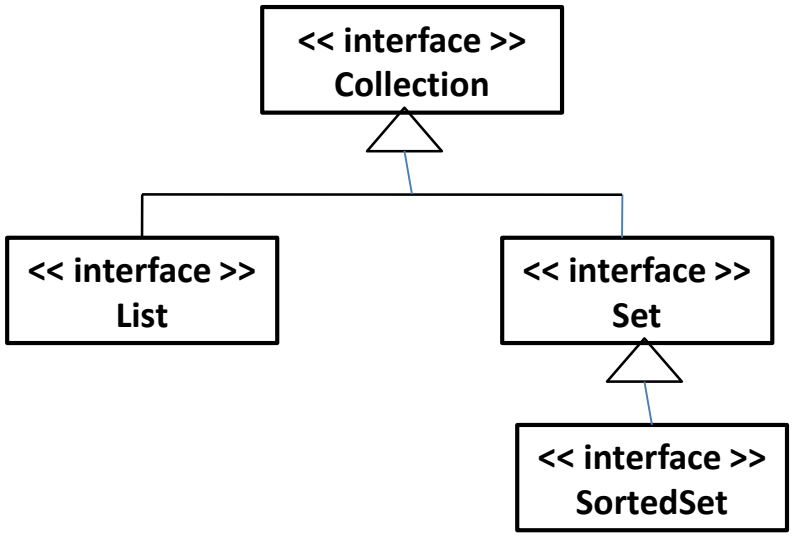
```
Dessinable d;
..
d = new RectangleDessinable(…);
...
d.dessiner(g);
d.surface();
```

permet de s'intéresser uniquement à certaines caractéristiques d'un objet

règles du polymorphisme s'appliquent de la même manière que pour les classes :
- vérification statique du code
- liaison dynamique

Heritage d'interfaces

- De la même manière qu'une classe peut avoir des sous-classes, une interface peut avoir des "sous-interfaces"
- Une sous interface
  - *hérite de toutes les méthodes abstraites et des constantes de sa "superinterface"*
  - *peut définir de nouvelles constantes et méthodes abstraites*



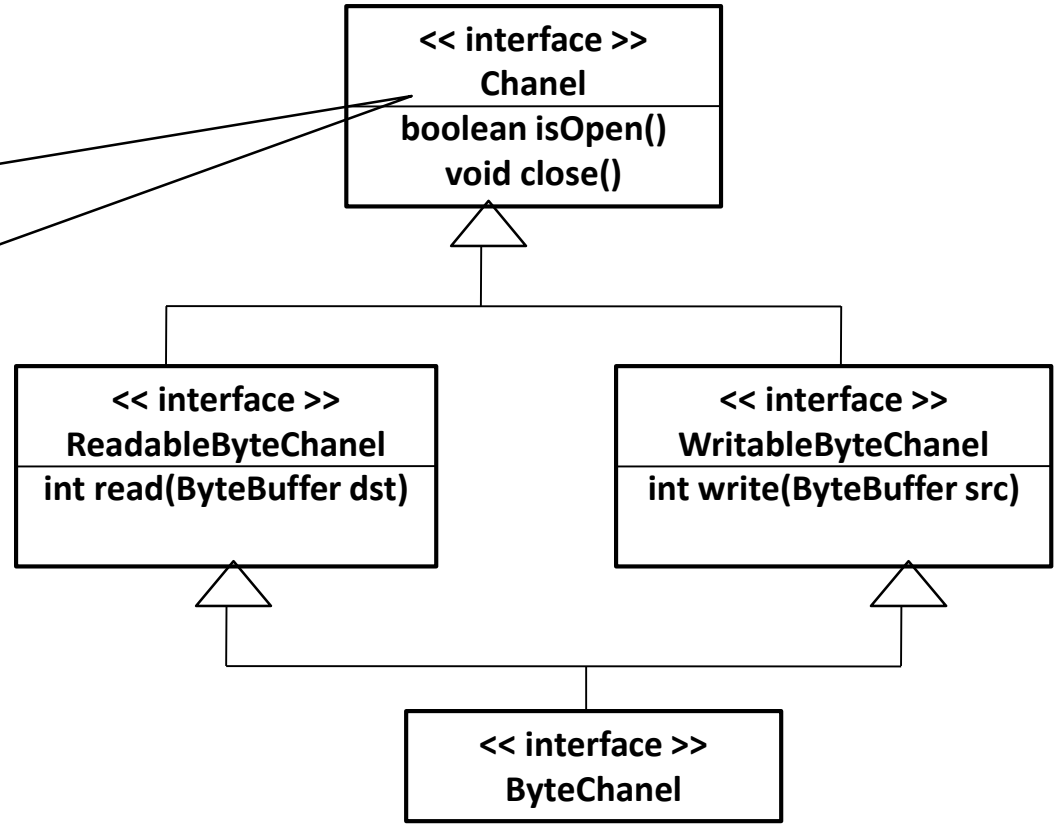

```
interface Set extends Collection{
  ...
}
```

- Une classe qui implémente une interface doit implémenter toutes les méthodes abstraites définies dans l'interface et dans les interfaces dont elle hérite.

Heritage d'interfaces

- A la différence des classes une interface peut étendre plus d'une interface à la fois

représente une connexion ouverte vers une entité telle qu'un dipositif hardware, un fichier, une "socket" réseau, ou tout composant logiciel capable de réaliser une ou plusieurs opérations d'entrée/sortie.




```
package java.nio;
interface ByteChannel extends ReadableByteChanel, WriteableByteChanel {
}
```

Interfaces : Intérêts

- Les interfaces permettent de s ’affranchir d ’éventuelles contraintes d’héritage.

♦ *Lorsqu’on examine une classe implémentant une ou plusieurs interfaces, on est sûr que le code d’implémentation est dans le corps de la classe. Excellente localisation du code (défaut de l’héritage multiple, sauf si on hérite de classes purement abstraites).*

- Permet une **grande évolutivité du modèle objet**

LES CLASSES ENVELOPPES (WRAPPERS)

LES CLASSES ENVELOPPES (WRAPPERS)

Les variables instances d'une classe (objets) et les variables de types primitifs ne se comportent pas de la même manière:

•L'affectation porte sur l'adresse d'un objet, sur la valeur d'une variable de type primitif

•Les règles de compatibilité de l'affectation se fondent sur une hiérarchie d'héritage pour les objets, sur une hiérarchie de type pour les variables.

•Le polymorphisme ne s'applique qu'aux objets

•Les collections ne sont définies que pour des éléments qui sont des objets.(collections = LinkedList, ArrayList, HashSet, TreeSet, …etc)

Les classes enveloppes permettent de manipuler les types primitifs comme des objets.

LES CLASSES ENVELOPPES (WRAPPERS)

Les classes WRAPPERS sont :

Boolean : encapsule le type primitif boolean

Character : encapsule le type char

Byte : encapsule byte

Short : encapsule short

Integer : encapsule int

Long : encapsule long

Float : encapsule float

Double : encapsule double

LES CLASSES ENVELOPPES (WRAPPERS)

Construction et accès aux valeurs

Toutes les classes enveloppes disposent d'un constructeur qui reçoit un argument d'un type primitif:

```
Character charObj = new Character('c');
Integer  intObj = new Integer(12) ;
 Double doubleObj = new Double(12.5);
```

Elles disposent toutes d'une méthode de la forme <typeprimitif>Value qui permet de retrouver la valeur du type primitif correspondant:

```
char a = charObj.charValue();
int n = intObj.intValue();
double x = doubleObj.doubleValue();
```

Ces instructions sont simplifiées dans le JDK5.0 à l'aide du principe « boxing/unboxing ».

Ces classes WRAPPERS sont finales (ne peuvent pas être dérivées), et inaltérables (Les valeurs qu'elles encapsulent ne sont pas modifiables

LES CLASSES ENVELOPPES (WRAPPERS)
Comparaison avec la méthode equals

L'opérateur = = appliqué à des objets, comparent leur référence.

Exemple : Integer intObj1 = new Integer(10);

Integer intObj2 = new Integer(10);

L'expression intObj1 == intObj2 retourne la valeur false (à moins que le compilateur ne crée qu'un seul objet de type Integer contenant la valeur 10, et référencé par les deux objets intObj1 et intObj2).

En revanche, la méthode equals a bien été redéfinie dans les classes WRAPPERS, de manière à comparer effectivement les valeurs correspondantes.
intObj1.equals(intObj2) retournera à la valeur true.

LES CLASSES ENVELOPPES (WRAPPERS)
Boxing / Unboxing dans JDK5.0

Le JDK 5.0 a introduit les techniques de boxing (emballage) et d'unboxing (déballage) qui effectuent des conversions, et mises en place automatiquement par le compilateur, entre les classes WRAPPERS et les types primitifs.

Exemple:

```
Integer  intObj = 10            // 10 est converti en  new Integer(10);
Double doubleObj = 10.5   //10.5 est converti en  new Double(10.5);

int n = intObj                  //intObj est converti en  intObj.intValue();
double x = doubleObj       //doubleObj est converti en doubleObj.doubleValue();
```

Ces convertions sont effectués également lors des opérations arithmétiques:

```
intObj2 = intObj1 + 5   //intObj2 = new Integer(intObj1.intValue() + 5);
intObj1++               // intobj = new Integer(intObj1.intValue() + 1);
```

Attention : Ces conversions ne sont possibles qu'entre un type enveloppe et son type primitif correspondant. Les instructions suivantes sont incorrects:

Double doubleObj = 10;   // 10, de type int, ne peut pas etre converti en Double.

Integer intObj; Double doubleObj = intObj //erreur de compilation

CLASSES ANONYMES

JAVA permet de définir ponctuellement une classe, sans lui donner de nom. Cette technique est très utilisée pour la gestion des événements dans les interfaces graphiques.

Une classe anonyme est soit une classe qui dérive d'une autre classe, soit elle implémente une interface.

Format d'une classe anonyme qui dérive d'une autre classe:

```
Class A{    // définition des champs et méthodes de A }
A a = new A(){
                    // Définition des champs et méthodes de la classe anonyme dérivant de A
                     };
```

Format d'une classe anonyme qui implémente une interface:

```
Interface  I{ // définition des constantes et déclaration des méthodes de I }
I a = new I(){
                     // Définition  des méthodes de la classe anonyme implémentant
l'interface I
                      };
```

CLASSE ANONYME
Dérivée d'une autre classe

Exemple :

```
Class A{
            public void affiche(){
                                    System.out.println(« Je suis un objet de la classe A »);
                                    }
        }
Public class Anonyme1{
  public static void main(String[] args){
    A a1 = new A();
    A a2 = new A(){
             public void affiche(){
              System.out.println(« Je suis un objet de la classe anonyme dérivée de
A »);
                                                    };
    a1.affiche();
    a2.affiche();
}
```

CLASSE ANONYME
Implémentant une interface

Exemple :

```
Interface I{
            public void affiche();
         }
Public class Anonyme2{
  public static void main(String[] args){
    I a = new I(){
            public void affiche(){
             System.out.println(« Je suis un objet de la classe anonyme qui implémente
                                        l'interface I »);
                                  };
    a.affiche();
}
```

Les chaînes de caractères
Classe String

Les chaînes des caractères sont des séquences de caractères.
Elles sont largement utilisées dans les programmation en Java.
En Java une chaîne de caractère est un objet.

Java fournit la classe **String** pour créer et manipuler les chaînes de caractères

Création des chaînes de caractères

La façon la plus directe de créer une chaîne est d'écrire : **String s = "Bonjour".**

"Bonjour " est une constante chaîne de caractère (litteral string).
Chaque fois que le compilateur rencontre une litteral string, il crée un objet instance de la classe String dont la valeur est la constante en question ("Bonjour " dans ce cas).

La classe String possède 11 constructeurs qui vous permettent de construire des objets String avec une valeur initiale provenant de plusieurs sources.

Ainsi, il est possible de créer un objet String à partir d’un tableau de caractères.

Exemple:

```
char[] bonjourTab = { ‘b', ‘o', ‘n', ‘j', 'o', ‘u’, ‘r’, '.'};

String bonjourString = new String(bonjourTab);

System.out.println(bonjourString); //affichera : bonjour
```

Note:

La classe String est **immuable**, par conséquent, une fois un objet String est créé, il ne peut pas être modifié. La classe String possède un ensemble de méthodes qui opèrent sur les objets comme s'ils les modifiaient, mais en réalité, puisque la classe String est immuable, ces méthodes créent et retournent de nouveaux objets qui contiennent les résultat de l'opération.

Longueur d'une chaîne

La classe String fournit une méthode d'instance length() qui permet d'obtenir la longueur d'une chaîne (Attention, ne pas confondre avec le champ public length d'un tableau).

Exemple:

```
String palindrome = "Dot saw I was Tod";

int len = palindrome.length(); // fournit 17
```

Concatenation des chaînes

La classe String inclut une méthode d'instance qui permet de concaténer deux chaînes: ch1.concat(ch2); retourne un nouveau objet String formé par les carctères de ch1 ausquels on a ajouté à la fin les caractères de ch2.

Attention, ch1 n'est pas modifié.

Il est aussi possible d'utiliser concat avec des constantes chaines comme

"Bonjour ".concat("tout le monde");

Les chaines sont souvent concaténées en utilisant l'opérateur +, qui est souvent utilisé dans les instructions print.
Exemple : « bonjour » + « tout le monde »; produit l'objet chaine « bonjour tout le monde ».
Il est possible de concaténer n'importe quel objet à une chaine. Lorsque l'objet n'est pas une chaine, sa méthode toString() est appelé pour le convertir en une chaine.

Note:
Java n'autorise pas de diviser une constante string en plusieurs lignes dans un programme source. Par conséquent, il faut avoir recours à l'opérateur + à la fin de chaque ligne dans le cas d'une constante chaîne multilignes.
Exemple :

```
String quote = "Now is the time for all good " +
               "men to come to the aid of their country.";
```

Création d'une chaine de format

La classe String possède une méthode statique format() qui permet de créer une chaine de format qui peut être réutilisée.
Par exemple, au lieu de :

System.out.printf(" La valeur de la variable float est %f, losque celle de la " + " variable entier est %d, et celle de type string est %s", floatVar, intVar, stringVar);
Vous pouvez écrire:

String fs;

fs = System.out.printf(" La valeur de la variable float est %f, celle de la " + " variable entier est %d, et celle de la variable string est %s", floatVar, intVar, stringVar);

System.out.println(fs);

Conversions entre nombre et chaines de caractères

Conversion des chaines en nombres

Chaque classe WRAPPER (Byte, Integer, Double, Float, Long, et Short) fournit une méthode nommée valueOf qui convertit une chaîne en un objet instance de la dite classe.

```
public class ValueOfDemo {
   public static void main(String[] args) {
     if (args.length == 2) {
         float a = (Float.valueOf(args[0]) ).floatValue();
         float b = (Float.valueOf(args[1]) ).floatValue();
         System.out.println("a + b = " + (a + b) );
         System.out.println("a - b = " + (a - b) );
         System.out.println("a * b = " + (a * b) );
         System.out.println("a / b = " + (a / b) );
        System.out.println("a % b = " + (a % b) ); }
   else {
     System.out.println("This program requires two command-line arguments."); } }
}
```

Note:

Chaque classe WRAPPER fournit également une méthode statique dont le nom est parseXXXX() (parseInt, parseFloat, …etc) qui convertit une chaîne en un nombre de type primitif.

Puisque cette méthode retourne un type primitif, elle est plus directe que la méthode. Ainsi dans le programme précédent, au lieu d'écrire :

```
float a = (Float.valueOf(args[0]) ).floatValue();
float b = (Float.valueOf(args[1]) ).floatValue();
```

Nous pouvons écrire directement

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Conversion des nombres en chaine de caractères

Java fournit plusieurs méthodes pour convertir un nombre en une chaine de caractères.
Méthode 1: utiliser l'opérateur + pour concatener un nombre à une chaine vide

```
int i;
String s1 = "" + i;        //la conversion est faite automatiquement.
```

Méthode 2 : Utiliser la méthode statique valueOf de la classe String.

```
String s2 = String.valueOf(i);
```

Méthode 3 : Utiliser la méthode toString() de la classe WRAPPER correspondante.

Par example:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

Manipulation des caractères dans une chaîne

La classe String fournit un ensemble de méthodes pour examiner le contenu des chaines de caractères, comme rechercher un caractère, ou une sous-chaine, changer la casse des lettres, …etc.

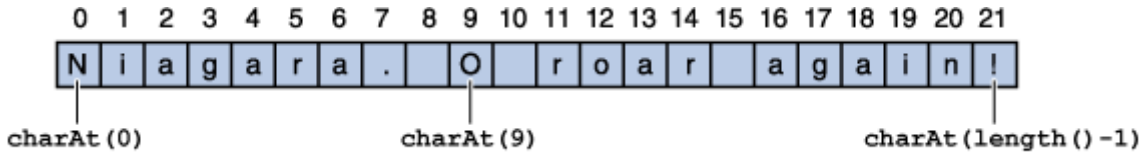
Obtention d'un caractère à partir d'une chaine

La méthode charAt() permet d'obtenir un caractère situé à un indice donné à l'intérieur d'une chaine. L'indice du 1er caractère de la chaine est 0, et celui du dernier est length()-1.

Par exemple le code suivant :

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Retourne le caractère 'O', qui se trouve à l'indice 9 (position 10) comme le montre la figure ci-dessous.



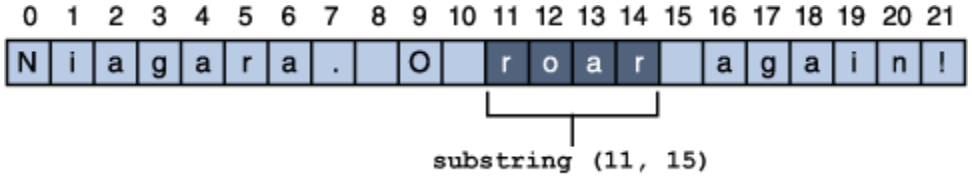
Obtention d'une sous-chaine à partir d'une chaine

La méthode substring permet d'extraire une sous chaine à partir d'une chaine de caractères. Cette méthode a deux versions:

String substring(int debut, int fin) : retourne la sous chaine située entre l'indice debut et l'indice fin-1.

String substring(int debut) : retourne la sous chaine située entre l'indice debut et la fin de la chaine.

String ch = "Niagara. O roar again!";
String sousch1 = ch.substring(11, 15); //Extrait la sous chaine "roar"



String sousch2 = ch.substring(16); //Extrait la sous chaine "again!"