CHAPITRE III:

Héritage et polymorphisme

Centre Universitaire de Mila
2ième Année Licence Informatique
Matière: Programmation Orientée Objet
Responsable de la matière: DR. SADEK BENHAMMADA

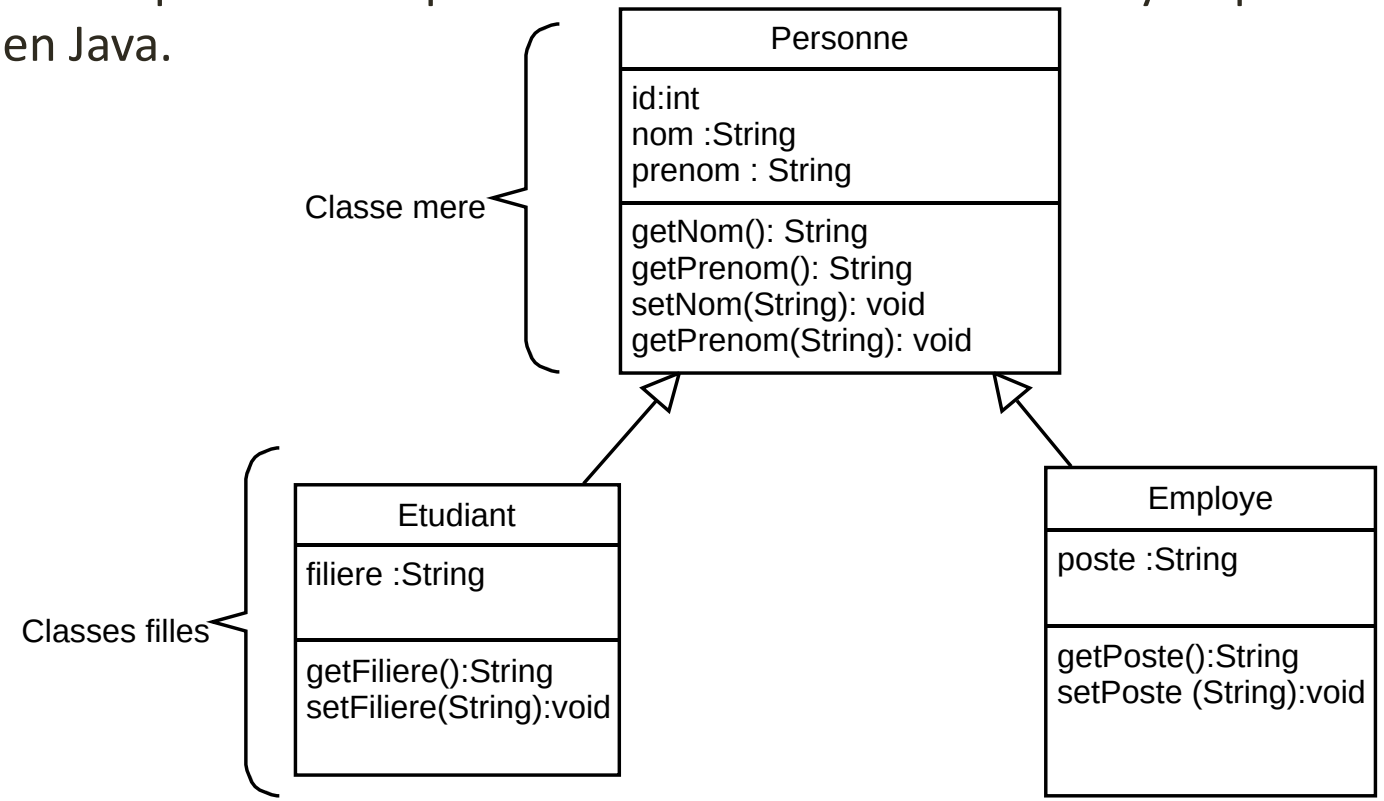
1. Généralités

1. Généralités

1.1. Définition

- L'héritage décrit une relation entre une classe générale (classe de base, classe mère ou classe parent) et une classe spécialisée (*classe fille ou* sous-classe).
- Les classes spécialisées héritent tous les attributs et toutes les méthodes de la classe générale.
- Une classe peut avoir plusieurs sous-classes.
- Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.

Exemple



2. Mise en œuvre de l’héritage

2. Mise en œuvre de l'héritage

2.1. Déclaration d'une classe fille

- Pour exprimer que la classe **ClasseB** hérite les attributs et les méthodes de la classe **ClasseA**, on écrira:

```
public class ClasseB extends ClasseA
```

- **ClasseA** est appelée la classe parent (ou mère) et **ClasseB** la classe dérivée (ou fille).

Exemple

```
class Etudiant extends Personne){...}
```

Un objet de la classes `Etudiant`:

1. **Hérite les attributs et méthodes de la classe *`Personne`*:**
   - Il doit disposer d'une valeur pour tous les attributs de la classe `Personne`
   - Il peut utiliser toutes les méthodes de la classe Personne
2. **Dispose des attributs et des méthodes supplémentaires**
3. **Peut redéfinir certaines méthodes non statiques de la classe *`Personne`***

2.2. Accès aux attributs et méthodes hérités

- Les attributs et méthodes définis avec le modificateur d'accès **`public`** sont accessible par les sous-classes et toutes les autres classes.
- Un attribut défini avec le modificateur **`private`** est hérité mais il n'est pas accessible directement par les sous-classes.
- Un attribut défini avec le modificateur **`protected`** est accessible directement dans toutes les classes filles

- **En général:**
  - **Les attributs d'une classe mère sont** déclarées **`protected`**, pour les rendre accessibles directement par les objets **de ses sous-classes;**
  - **Les méthodes** sont déclarées **`public`.**

Exemple

```
public class Personne{
        protected int id;
        protected String prenom;
        protected String nom;
        public String getNom(){...}
          ...
}
```

```
class Etudiant extends Personne{
    private String filiere;
    public String getFiliere(){...}
    public void setFiliere(String f){...}
...
}
```

```
class Employe extends Personne{
      private String poste;
      public String getPoste(){...}
      public void setPoste(String f){...}
...
}
```

2.3. Déclaration du constructeur d'une sous-classe

- Le constructeur d'une sous-classe doit toujours commencer par appeler le constructeur de sa classe mère.
- L'appel du constructeur de la classe mère se fait avec l'instruction **super(paramètres)**, avec les paramètres adéquats.

Exemple

```
public class Personne{
  protected String prenom;
  protected String nom;
  ...
  public personne(String prenom,String nom){
    this.prenom=prenom;
    this.nom=nom;
  }
}
```

```
public class Etudiant extends Personne{
    private String filiere;
    ...
    public Etudiant(String prenom,String nom, String filiere){
        super(prenom, nom);
        this.filiere=filiere;
     }
}
```

3. Redéfinition d'une méthode héritée (Overriding)

3. Redéfinition d'une méthode héritée (Overriding)

- On appelle ***redéfinition*** le fait de **réécrire** dans une classe fille une méthode de la classe mère.
- La méthode de la classe fille porte le même nom et possède les mêmes paramètres que la méthode de la classe mère.
- Si la signature de la méthode change (les paramètres), ce n'est plus une redéfinition **(Overriding)** mais une surcharge **(overloading)**.

Exemple: Redéfinition de la méthode **toString()** de la classe mère Personne, dans la classe fille Étudiant:

```
public class Personne{
        protected String nom;
        protected String prenom;

        public Personne(String nom,String prenom){
            this.nom=nom;
            this.prenom=prenom;}

        public String toString(){
           return "Nom:"+this.nom+",Prenom:"+this.prenom;}
  }
```

Redéfinition d'une méthode héritée (Overriding)

Exemple (suite)

```
public class Etudiant extends Personne{
        private String filiere;

        public Etudiant(String nom, String prenom, String filiere){
            super(nom, prenom);
            this.filiere=filiere;
         }

//Redefinition de la méthode toString()
        public String toString() {
            return "Nom:"+this.nom+",Prenom:"+this.prenom
                    + ", Filiere: "+ this.filiere;
         }
        public static void main(String arg[]){
           Etudiant e=new Etudiant("Ahmed", "Ali", "Informatique");
            System.out.println(e.toString());
        }
}
```

Resultat affiché:
Nom:Ahmed, Prenom:Ali, Filiere: Informatique

3. Redéfinition d'une méthode héritée (Overriding)

Le mot-clé `super`

- Le mot-clé *super* permet de désigner la classe mère.
- À l'aide du mot-clé *super*, il est possible de manipuler les attributs et les méthodes de la classe mère.
- Pour manipuler un attribut de la classe mère, il suffit d'utiliser la syntaxe suivante :

```
super.nomAttribut
```

- De la même façon, pour manipuler une méthode de la superclasse, il suffit d'utiliser la syntaxe suivante :

```
super.nomMethode()
```

Redéfinition d'une méthode héritée (Overriding)

Exemple: Appel de la méthode **toString()** de la classe mère **Personne**, dans celle de la classe fille **Etudiant**:

```
public class Personne{
        ...
        public String toString(){
           return "Nom:"+this.nom+",Prenom:"+this.prenom;}
  }
```

```
public class Etudiant extends Personne{
        ...
         //Redefinition de la méthode toString()
        public String toString()
         {
            return super.toString()+", Filiere: "+ this.filiere;
         }
        public static void main(String arg[]){
           Etudiant e=new Etudiant("Ahmed", "Ali", "Informatique");
            System.out.println(e.toString());}
}
```

Resultat affiché:

Nom:Ahmed, Prenom:Ali, Filiere: Informatique

4. L’abstraction

4. L'abstraction

- L'abstraction s'applique aux méthodes et aux classe
- Une méthode abstraite est une méthode déclarée avec le modificateur **`abstract`** et sans corps:

```
abstract typeDeRetour nomMethode (params)
```

- Une méthode déclarée **`abstract`** ne peut être exécutée. Sa déclaration indique simplement que les sous-classes doivent la redéfinir.
- Toute classe qui possède au moins une méthode abstraite doit être déclarée **abstraite**:

```
abstract class NomDeClasse { ... }
```

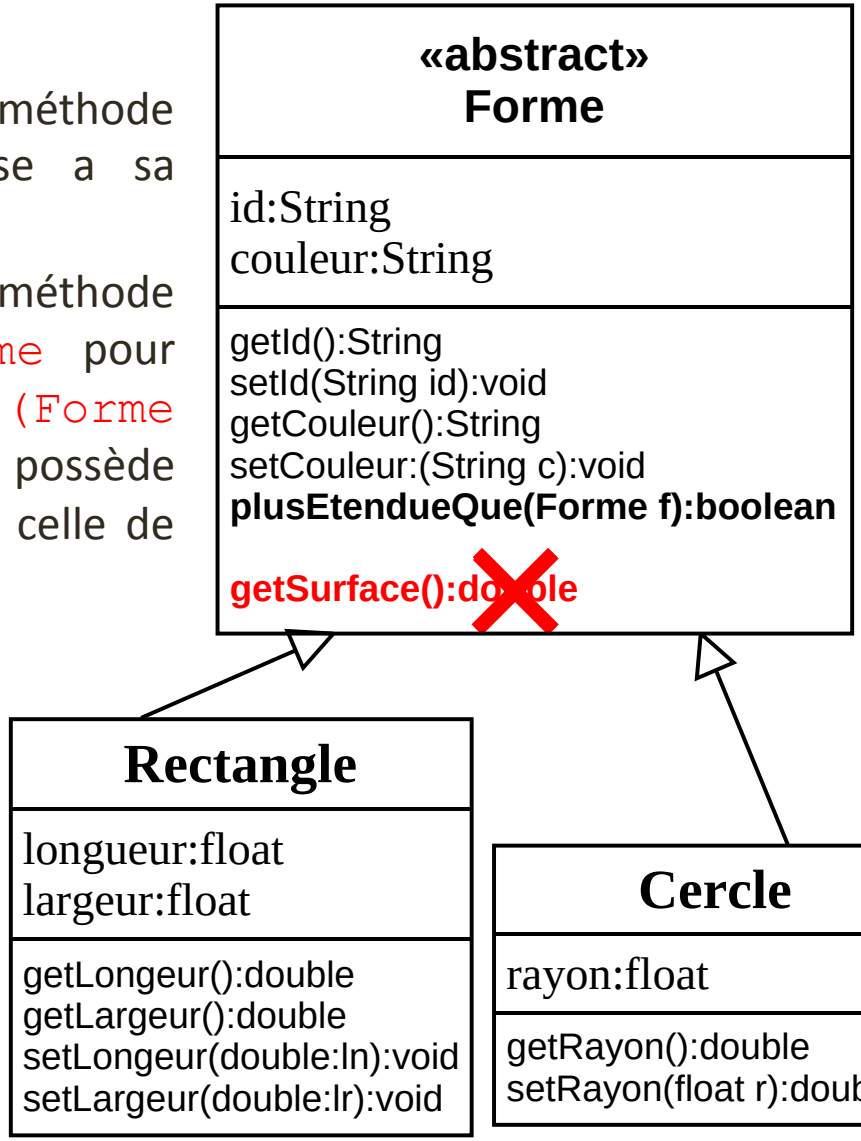
- Une classe ***abstract*** ne peut pas être instanciée (On ne peut pas créer des objets d'une classe abstraite).
- Une classe abstraite peut avoir des sous-classes qui implémentent les méthodes abstraites.
- Si une sous-classe ne redéfinie pas de façon concrète une méthode abstraite, la sous-classe doit être déclarée explicitement absraite.

4. L'abstraction

Exemple

Problème

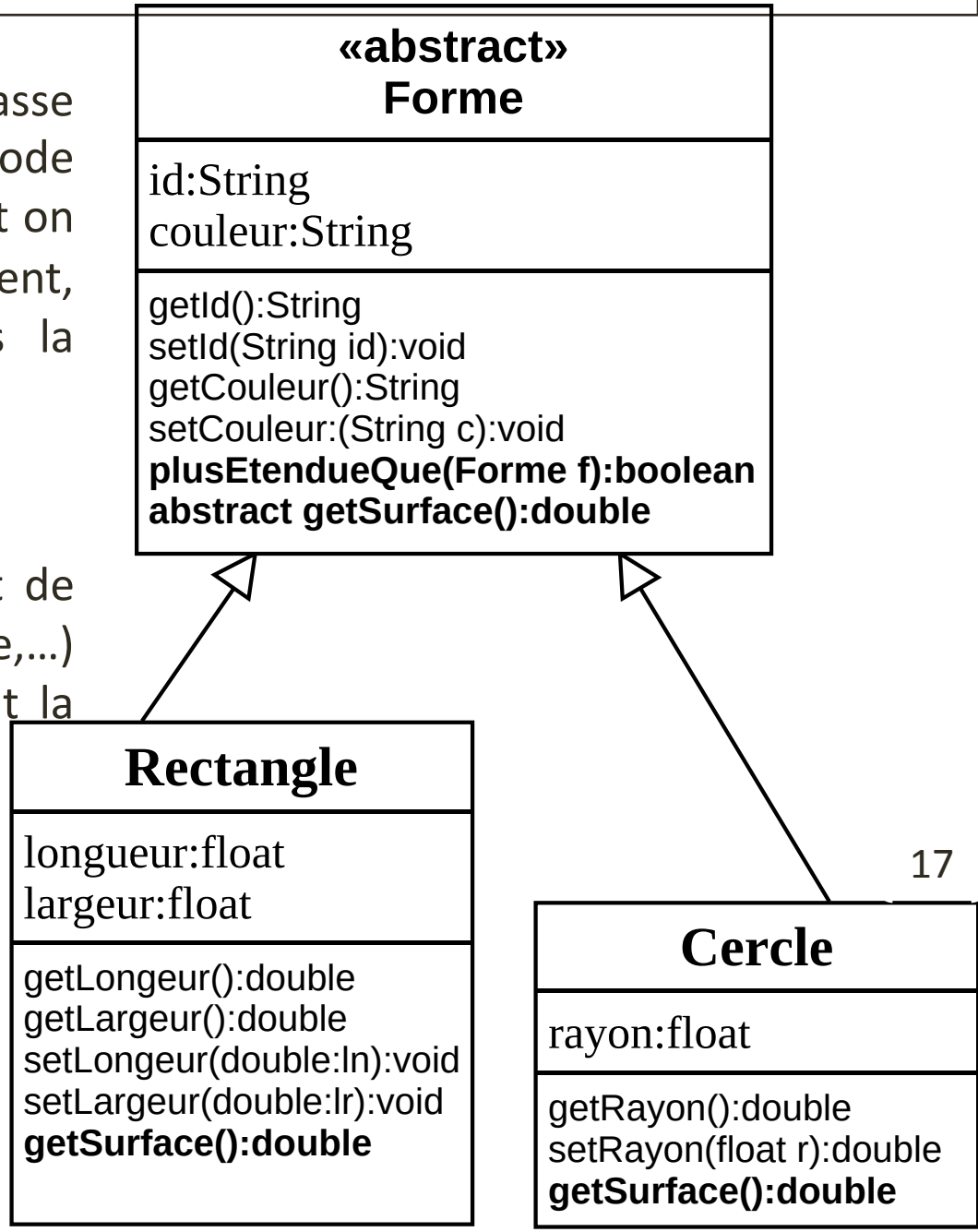
- Dans la classe mère **`Forme`**:
  1. Il n'est pas possible d'écrire une méthode `getSurface()`: Chaque sous-classe a sa propre méthode getSurface().
  2. On a besoin de la méthode `getSurface()` dans la classe `Forme` pour écrire la méthode `plusEtendueQue(Forme f)` qui détermine si une autre forme possède une surface supérieure ou inférieure à celle de l'objet appelant.



4. L'abstraction

Solution

1. Définir au niveau de la classe générique `Forme` la méthode abstraite `getSurface()`**;** dont on ne précise pas le comportement, mais qui sera appelée dans la méthode `plusEtendueQue(Forme f)`.
2. Toutes les classes qui héritent de cette classe (Rectangle, Cercle,…) devront spécifier concrètement la méthode **`getSurface().`**



4. L'abstraction

```
public abstract class Forme {
...
        public abstract double getSurface();
        public boolean plusEtendueQue(Forme f){
                double s1 = this.getSurface();
                double s2 = f.getSurface();
                return (s1>s2);
        }
}
```

```
public class  Rectangle extends Forme {
   private double longueur;
   private double largeur;
   ...
   public double getSurface() {
     return longueur*largeur;
   }
}
```

```
public class Cercle extends Forme {
   private double rayon;
...
        public double getSurface()  {
        return Math.pow(rayon,2)*Math.PI;
   }
}
```

5. Le polymorphisme

5. Le polymorphisme

- Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé.
- En JAVA, le moyen de réaliser le polymorphisme est le transtypage (*cast* en anglais).
- Le cast s’applique sur les types de base, et sur les classes.

5. Le polymorphisme

5.1. Cast des types de base

- C'est la conversion de types qui se fait entre les types de base du langage java, (byte, short, int, long, float, double, char).
- Le cast peut être implicite ou explicite:
- Un transtypage est implicite si le type cible a un domaine de valeurs plus grand que le type d'origine : byte (1 octets) → short (2) → int (4) → long (8) → float (4) →double (8).

```
//Exemple de transtypage implicite
int i =3;
double d = i; //d vaut 3.0
```

- Un transtypage est explicite si le type cible a un domaine de valeurs plus petit que le type d'origine.

```
//Exemple de transtypage explécite
double d = 3.96;
int i = (int)d;//i vaut 3
```

5.2. Cast entre classes

- Il est possible de convertir un objet d'une classe **`A`** en un objet d'une classe **`B`** si les classes **`A`** et **`B`** ont un lien d'héritage.
- Le transtypage d'un objet dans le sens **fille→mère** est toujour possible et implicite.
- Le transtypage dans le sens **mère→fille** doit être explicite et n'est pas toujours possible.

5. Le polymorphisme

5.2.1. Le cast implicite (sens fille→mère)

- Il est possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance).
- Le casr de la classe fille vers la classe mère est implicite
- Un objet a:
  - Un type déclaré dans le code source (vérifié à la compilation par Javac)
  - Un vrai type : celui qui lui est donné lors de l'appel du constructeur (vérifié à l'exécution par la JVM).

Exemples

```
public class Personne{
private String nom;
...
private void setNom(String n){this.nom=n;}
...}
```

```
public class Etudiant extends Personne{
private String filiere;
...
private void setFiliere(String f){this.filiere=f;}
}
```

5. Le polymorphisme

Exemples (suite)

```
Personne pers = new Etudiant();
```

Type déclaré — *Vrai type*

Code	Correcte/Erreur
`Etudiant e1 = new Etudiant();`	Correcte
`Personne p1 = new Etudiant ();`	Correcte: Etudiant hérite de Personne
`Etudiant e = new Personne();`	Erreur de compilation: Etudiant n'est pas une classe dérivé de Personne
`Personne p2 = e1;`	Correcte: Etudiant hérite de Personne
`Object obj=e;`	Correcte: Etudiant est une classe dérivée de Object comme toutes les classes déclarées en java
`e1= p1;`	Erreur de compilation: Personne n'est pas une classe dérivée de Personne
`p1.setNom("Ahmed");`	Correcte: setNom() est une méthode du type déclaré de p1 (Personne)
`p1.setFiliere(« Math");`	Erreur de compilation:: setFiliere() n'est pas une méthode du type déclaré de p1 (Personne)

5. Le polymorphisme

5.2.1. Le cast implicite (sens fille→mère)

- Il est aussi possible de caster implicitement des paramètres et des types de retour.

Exemple

```
Public class Personne{
private String nom;
private String prenom;
private int age;
public  boolean plusAgeQue(Personne p){return this.age>p.age;}
...
}
```

```
Public class Etudiant extends Personne{
private String filiere;
...
}
```

```
public class Test
{
        public static void main(String arg[]){
        Personne p=new Personne("Idir","Kamel",20);
        Etudiant e= new Etudiant("Mohammed","Ali",22,"Informatique");
        boolean b=p.plusAgeQue(e);//cast implécite
}
```

5. Le polymorphisme

5.2.1. Le cast explicite (sens mère →fille)

- Pour utiliser une référence de la superclasse pour désigner un objet d'une classe dérivée, il faut utiliser un transtypage explicite.
- Pour que la transtypage réussisse, le vrai type de l'objet à convertir doit être la sous-classe.

Exemple

Code	Correcte/Erreur
`Etudiant e = new Personne();`	Erreur de compilation: L'objet **e** est instance de la superclasse (Personne),
`Personne p=new Etudiant;` `Etudiant e =(Etudiant)p;`	Correct: L'objet p est instance de la sous-classe (Etudiant),
`Personne p=new Etudiant;` `p.setFiliere("Informatique ");`	Erreur de compilation: **setFiliere()** n'est pas une méthode du type déclaré de **p (Personne).**
`Personne p=new Etudiant;` `((Etudiant)p).setFiliere("Math");`	Correcte: Cast explécité sens mere→fille, et le vrai type de **p** est Etudiant.

5. Le polymorphisme

5.3. L'operateur instanceof

- Pour L'operateur ***instanceof*** est utilisé pour tester si un objet est une instance d'un type donné ou de l'une des sous-classe de ce type.

Exemple

```
public class Personne{}
```

```
public class Etudiant extends Personne{}
```

```
public class Employe extends Personne{}
```

```
public class Test {
  public static void main (String arg[]){
    Personne e = new Etudiant() ;
    System.out.println(e instanceof Etudiant); //true
    System.out.println(e instanceof Personne); //true
    System.out.println(e instanceof Employe); //false
```

5. Le polymorphisme

5.4. Utilité du polymorphisme

- Le polymorphisme est très utile pour la création d'ensembles (**tableaux ou listes**) regroupant des objets de classes différentes :

Exemple

```
public class Forme {}
```

```
public class Rectangle extends Forme {
    private double longueur;
    private double largeur;
    public Rectangle(double longueur, double largeur){
       this.longueur=longueur;
       this.largeur=largeur;}
    public double getSurface() { return longueur*largeur;}
}
```

```
public class Cercle extends Forme{
    private double rayon;
    public Cercle(double rayon)    { this.rayon=rayon;}
    public double getSurface() {
        return Math.pow(rayon,2)*Math.PI;}
}
```

5. Le polymorphisme

Exemple (Suite)

```
public class TestForme {
 public static void main(String[] args)
  {
   Forme[] formes=new Forme[4];
   formes[0]=new Rectangle(5,10); // Cast implicite
   formes[1]=new Cercle(5);//Cast implicite
   formes[2]=new Rectangle(10,20);//Cast implicite
   formes[3]=new Cercle(10);//Cast implicite
   System.out.println("Listes des formes et leurs surfaces");
   for (int i=0;i<formes.length;i++) {
    if (formes[i] instanceof Rectangle)
      System.out.println("Un rectangle de surface:"+
           ((Rectangle)formes[i]).getSurface());//Cast explicite
    if (formes[i] instanceof Cercle)
      System.out.println("Un Cercle de surface:"+
           ((Cercle)formes[i]).getSurface());//Castexplicite
   }}}
```

```
Listes des formes et leurs surfaces
Un rectangle de surface:50.0
Un Cercle de surface:78.53981633974483
Un rectangle de surface:200.0
Un Cercle de surface:314.1592653589793
```

6. Le mot clé *final*

6. Le Le mot clé *final*

- Le mot-clé *final* s'applique aux variables, méthodes et classes.
- Le mot clé *final* permet d'indiquer qu'un élément ne sera pas modifié.

6.1. Les variables final

- Une variable déclarée *final* ne peut plus voir sa valeur modifiée (une constante).
- S'il s'agit d'un attribut, les constantes sont aussi déclarées statiques, pour gagner de la place en mémoire (une seul copie pour tous les objets).

Exemple

```
public class Cercle extends Forme{
   static final double PI=3.141592653589793;
   private double rayon;
   public Cercle(double r)  {
       rayon=r;
    }
   public double getSurface()  {
        return rayon*rayon*PI;
   }
}
```

6.2. Les méthodes final

- Dans le cas d'une méthode, cela signifie que les classes qui héritent de la classe dans laquelle cette méthode est définie ne peuvent pas redéfinir cette méthode. Les sous-classes doivent utiliser cette méthode telle quelle.

Exemple

```
public class A{
    public final void methode (){…}
    ...
}
```

```
public class B extends A{…
    public void methode (){…} //Erreur
    ...
}
```

6.3. Les classes final

- Dans le cas d'une classe, **`final`** signifie qu'aucune autre classe ne peut hériter de cette classe.
- **Exemple**:

```
public final class A{…}
```

```
public class B extends A{…} //erreur
```

- Beaucoup de classes de la librairie java sont ***final***, entre autres: **`java.lang.System`**, **`java.lang.String`**, et **`java.lang.Math`**.

7. Les interfaces

7. Les interfaces

- Avec l'héritage multiple, une classe peut hériter de plusieurs super-classes. **L'héritage multiple n'existe pas en Java.**
- **Les interfaces** permettent de de remplacer l'héritage multiple,
- Une interface est une collection de méthodes abstraites, de constantes, de méthodes statiques, et des méthodes par défaut,

Déclaration d'une interface :

```
[public] interface nomInterface [extends Interface1, Interface2 ... ]
{
    // corps de l'interface
}
```

- Une classe implémente une interface, héritant les méthodes et les constantes de l'interface.
- Une classe peut implémenter plusieurs interfaces;

Implémentation d'une interface

```
[Modificateurs] class nomClasse [extends superClasse]
[implements nomInterface1, nomInterface 2, ...]
{
     //corps de la classe
}
```

7. Les interfaces

- Les méthodes d'une interface sont publique abstraites : elles sont **implicitement** déclarées avec les modificateur **`public`** et **`abstract;`**
- Les attributs d'une interface sont des constantes publiques , ils sont **implicitement** déclarées avec les modificateurs **`public`**, **`static`** et **`final`**.
- **Une méthode par défaut** une interface, est une méthode définit par sa signature son une implémentation.

- Une classe concrète (non abstraite) qui implémente une interface doit nécessairement implémenter toutes les méthodes abstraites de cette interface.

- Si une interface possède des méthodes par défaut, les instances des classes concrète qui implémente cette interface peuvent appeler ces méthodes.
- Les classes peuvent aussi redéfinir les méthodes par défaut des interfaces qu'elles implémentent.

7. Les interfaces

Exemple 1

```
public interface Individu{
    String getNom();
    String getPrenom()
}
```

```
interface AfficheType {
void afficherType();

}
```

```
public class  voiture implements AfficheType
{
     ...
 public void afficherType(){ System.out.println("Je suis une voiture");}
}
```

```
public class  Etudiant implements AfficheType, Individu
{
   ...

 public String getNom(){return this.nom;}
 public String getPrenom(){return this.prenom;}
 public void afficherType(){ System.out.println("Je suis un Etudiant");}
}
```

7. Les interfaces

Exemple 2

```
public interface Geometrie
{
    double PI=3.14;
    double getSurface();
    double getCirconference();
    default double carre(double x){return x*x;}
}
```

```
public class Rectangle implements Geometrie{
   private double longueur;
   private double largeur;
...
   public double getSurface() {return longueur*largeur;}
   public double getCirconference() {return longueur*largeur*2;}
}
```

```
public class Cercle implements Geometrie{
   private double rayon;
   public double getSurface() {return carre(rayon)*PI;}
   public double getCirconference() {return 2*PI*rayon;}
}
```

8. Les énumérations (enum)

8. Les énumérations

- Une `enum` contient la liste des valeurs possibles qu'une donnée peut prendre. Elle se déclare avec le mot-clef `enum`.

Exemple

```
Exemple:
public enum Filiere {
        informatique,mathematique,economie,lettres;
}
```

```
Exemple:
public class Etudiant extends Personne{
        // ...
private Filiere filiere;
// ...
public Etudiant(String prenom, String nom, int age,Filiere filiere){
super(prenom,nom,age);
this.filiere=filiere;
}
public  static void main(String[] args) {
Etudiant etud = new Etudiant("Mohammed","Ali",20,Filiere.informatique);
}
}
```

9. Les tableaux et les collections

9. Les tableaux et les collections

9.1. Les tableaux

- Un tableau permet de rassembler sous un même identificateur des données de même type.

9.1.1. La déclaration

- La syntaxe de déclaration d'un tableau à une dimension est la suivante :

```
<type> <nomTableau>[] =new <type> [n]
```

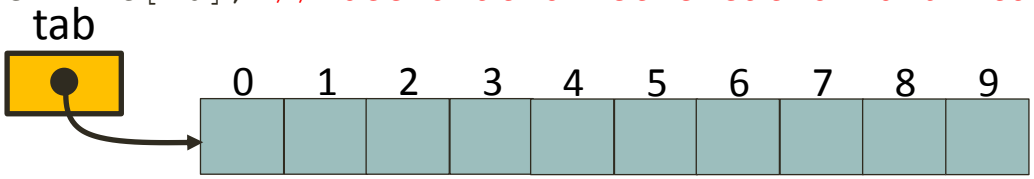
ou:

```
<type>[] <nomTableau> =new <type> [n]
```

- **n** représente le nombre d'elements du tableau;

Exemples

```
int tab [] = new int[10]; // déclaration et création d'un tableau de 10 entiers
```




```
char tabc[]; // déclaration
tabc = new char[10]; //création
Personne tabP=new Personne[10] // déclaration d'un tableau d'objets de la classe Personne
```

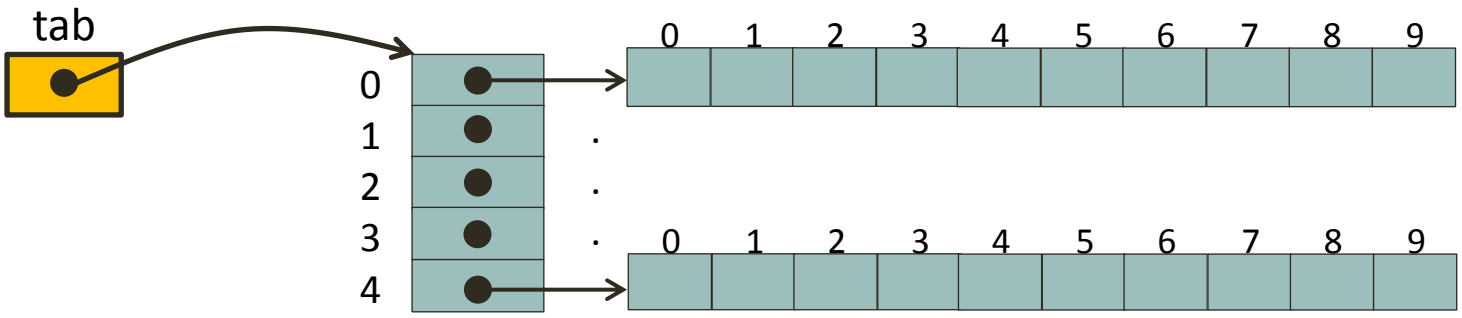
9. Les tableaux et les collections

- Java ne supporte pas directement les tableaux à plusieurs dimension, la solution consiste à déclarer **un tableau de tableaux**:

```
<type> <nomTableau>[][]=new <nomTableau> [n][p]
ou
<type> [][] <nomTableau>=new <nomTableau> [n][p]
```

Exemple

```
int tab [][]=new int[5][10]
```



9.1.2. L'initialisation

- Il est possible d'initialiser le tableau à sa création.

Exemple:

```
int tab1[] = {10,20,30,40,50};
int tab2[][] = {{5,1},{6,2},{7,3}};
```

9.1.3. Le parcours

- Un tableau possède une constante publique **`length`** dont la valeur est la taille du tableau. Il est possible d'utiliser cette constante pour le parcourir.

Exemple

```
int tab [] = new int[10];
...
for (int i = 0; i < tab.length ; i ++)
{
}
```

9.2. Les collections

- Une fois un tableau est créé, sa taille ne peut pas être modifiée. Par exemple, si nous créons un tableau de 20 éléments, **nous sommes limités à stocker au maximum 20 éléments** en utilisant ce tableau!

- Le package **`java.util`** inclut des classes permettant de gérer des ensembles d'objets. Ces classes sont appelés **Java Collection Framework (JCL).**

- Les collections n'ont pas une taille prédéfinie: **Le nombre d'élément qu'on peut stocker dans une collection n'est pas limité.**

- Il existe de nombreuses collections: **`ArrayList`**, **`LinkedList`**, **`Vector`**, **`HashSet`**, **`HashMap`**, etc.

9. Les tableaux et les collections

Exemple: `ArrayList`

- Les listes de type **`ArrayList`** sont souvent utilisées, et possèdes de nombreuses caractéristiques.
- Pour utiliser une liste de type ArrayList, il faut importer la classe **`ArrayList:`**

```
import java.util.ArrayList;
```

- Création de la liste:

```
ArrayList maListe=new ArrayList();
```

- Après la création, il suffit d'utiliser les méthodes de la classe **`ArrayList`** pour manipuler la liste:
  - **`add()`** permet d'ajouter un élément ;
  - **`get(int i)`** retourne l'élément à l'indice **`i`**;
  - **`remove(int i)`** supprimer l'élément l'indice **`i`**;
  - **`size()`** retourne le nombre d'elements dans la liste ;
  - **`isEmpty()`** renvoie «`true`» si la liste est vide ;
  - **`clear()`** supprimer tous les éléments de la liste;
  - **`contains(Object element)`** retourne « `true`» si l'élément passé en paramètre est dans la liste, et `false` sinon.

Exemple: `ArrayList`

- Une liste de type **`ArrayList`** est **hétérogène**: elle peut contenir des objets de différents types.

Exemple

```
public class Personne{...}
```

```
import java.util.ArrayList;
public class TestArrayList{
  public static void main(String arg[]){
   ArrayList maListe=new ArrayList();
   maListe.add("Bonjour");
   maListe.add(12);
   Personne p=new Personne("Ahmed", "Ali");
   maListe.add(p);
   for(int i = 0; i < maListe.size(); i++)
        System.out.println("Element "+i+" = "+maListe.get(i));
  }
}
```

Résultats affiché:
Element 0 = Bonjour
Element 1 = 12
Element 2 = Nom: Ahmed, Prenom: Ali

Exemple: `ArrayList`

- Il est possible de créer une liste **homogène**, dans laquelle les éléments sont limités à un type spécifique:

```
ArrayList<typeElements> maListe=new ArrayList();
```

Exemple

```
ArrayList<Personne> maListe=new ArrayList();
maListe.add("Bonjour");// Erreur compilation
maListe.add(12); // Erreur compilation
Personne p=new Personne("Ahmed", "Ali");
maListe.add(p); //Correcte
```

10. Les packages de base

- le JDK contient un certain nombre de packages,
- Chaque package regroupe un ensemble de classes qui couvrent un même domaine et apportent de nombreuses fonctionnalités.
- Il est possible de consulter toutes la documentation des APIs sur le site :

http://download.oracle.com/javase/1.4.2/docs/api/

java.io	Entrées/Sorties
java.applet	Manipulation des applets
java.lang	Primitives du langage
java.util	Utilitaires, structures de données
java.awt	Interface graphique AWT
java.awt.events	Evénements graphiques AWT
javax.swing	Interface graphique SWING
javax.swing.event	Evénements graphique SWING

10. Les packages de base

10.1. Le package java.lang

- Le package **java.lang** regroupe les classes fondamentales telles que:
  - **`Object`**,
  - **`Class`**,
  - **`Math`**,
  - **`System`**,
  - **`String`**,
  - **`Thread`**,
  - Les classes enveloppes des types primitifs (wrappers): **`Int`**, **`Double`**, **`Boolean`**, **`Byte`**, etc.
  - etc.

java.lang est un package par défaut: Il n'est pas nécessaire d'importer ce package pour pouvoir référencer ses classes.

10.1.1. La classe *java.lang.*Object

- **Object** est la super classe de toutes les classes Java : toutes ces méthodes sont héritées par toutes les classes;
- Quelques méthodes de la classe Object:

a)La méthode `Class getClass()`

La méthode **`getClass()`** renvoie un objet de la classe **`Class`** qui représente la classe de l'objet.

Exemple

```
Personne p= new Personne("Mohammed","Ali",20);
String nomClasse = p.getClass().getName();
System.out.println(nomClasse);
```

// Le résultat affiché est:
Personne

b)La méthode `String toString()`

- La méthode `toString()` de la classe `Object` renvoie le nom de la classe, suivi du séparateur @, lui même suivi par l'adresse de cet objet.

Exemple

```
Personne p= new Personne("Mohammed","Ali");
System.out.println(p.toString());
```

// Le résultat affiché peut être:
Personne@190d11

c) La méthode `boolean equals(Object obj)`

- La méthode `equals()` implémente une comparaison par défaut. Elle compare les références de l'objet appelant et de l'objet passé en paramètre.
- `o1.equals(o2)` renverra `true` si `o1` et `o2` désignent le même objet.

Exemple 1

```
Personne p1 = new Personne("Mohammed","Ali");
Personne p2 = new Personne("Mohammed","Ali");
System.out.println(p1.equals(p2)) ; //Affiche false
p1=p2;
System.out.println(p1.equals(p2)) ; //Affiche true
```

- Il est possible redéfinir la méthode `equals()`. Par exemple la classe `String` redéfinit `equals()`.

Exemple 2

```
String a ="Pomme";
String b ="Pomme";
String c ="Orange";
String d =a;
System.out.println(a.equals(b));
System.out.println(a.equals(d));
System.out.println(a.equals(c));
```

// Le résultat affiché :
true
true
false

Exemple: Redéfinition de `equals()` dans la classe Personne.

```
public class Personne{
   ...
 public boolean equals (Object o)
   {
   if (this.nom.equals(((Personne)o).nom)
      &&(this.prenom.equals(((Personne)o).prenom)))
      return true;
   else
      return false;
   }
```

```
Personne p1 = new Personne("Mohammed","Ali");
Personne p2 = new Personne("Mohammed","Ali");
System.out.println(p1.equals(p2)) ; //Affiche true
```

// Le résultat affiché :
true

d) La méthode `clone()` de la classe `Object`

Permet de créer un deuxième objet indépendant mais identique à l'original (voir le chapitre II) .

10.1.2. La classe *java.lang.String*

- En java, une chaîne de caractères est contenue dans un objet de la classe `String`.
- On peut initialiser une variable String sans appeler explicitement un constructeur.

Exemple: Les deux instructions suivantes sont identiques:

```
1  String s= "bonjour";
2  String s= new String("bonjour");
```

- L'opérateur + permet la concaténation de chaînes de caractères.
- La comparaison de deux chaînes doit se faire en utilisant la méthode **`equals()`** qui compare les chaine de caractères, et non l'opérateur **`==`** qui compare les références de ces objets :

```
String s1 = new String("Bonjour");
String s2 = new String("Bonjour");
System.out.println(s1 == s2); // affiche false
System.out.println( s1.equals(s2)); // affiche true
```

10.1.2. La classe *java.lang.String*

- La classe String possède de nombreuses méthodes. Voici quelques une:

`length()`	renvoie la longueur de la chaîne
`charAt(int)`	renvoie le nième caractère de la chaîne
`startsWith(String )`	vérifie si la chaîne commence par la sous chaîne
`endsWith(String)`	vérifie si la chaîne se termine par l'argument
`concat(String)`	ajoute l'argument à la chaîne et renvoie la nouvelle chaîne
`substring(int,int)`	renvoie une partie de la chaine
`toLowerCase()`	renvoie la chaîne en minuscule
`toUpperCase()`	renvoie la chaîne en majuscule

Exemple:

```
String s = "INFORMATIQUE";
System.out.println(s.length());// Affiche 12
System.out.println(s.charAt(2));// Affiche F
System.out.println(s.startsWith("INFO"));// Affiche true
System.out.println(s.endsWith("QUE"));// Affiche true
System.out.println(s.concat(" ET MATHEMATIQUE"));// Affiche
INFORMATIQUE ET MATHEMATIQUE
System.out.println(s.substring(0,4));// Affiche INFO
System.out.println(s.toLowerCase());// Affiche informatique
System.out.println(s.toUpperCase());// Affiche INFORMATIQUE
```

10.1.3. La classe *java.lang.System*

- La classe System défini trois attributs statiques qui permettent d'utiliser les flux d'entrée/sortie.

Attribut	Type	Rôle
`in`	`InputStream`	Entrée standard du système. Par défaut, c'est le clavier.
`out`	`PrintStream`	Sortie standard du système. Par défaut, c'est le moniteur.
`err`	`PrintStream`	Sortie standard des erreurs du système. Par défaut, c'est le moniteur.

- **`InputStream`** et **`PrintStream`** sont des classes du package **`java.io`**
- Le package **`Java.io`** définit un ensemble de classes pour la gestion des flux d'entrées-sorties.
- Les méthodes ***`print`*** et ***`println`*** de la classe ***`PrintStream`*** existent pour les types *int*, *long*, *float*, *double*, *boolean*, *char* et *String*.
- La méthode **printf()** de la classe ***`PrintStream`*** reprend le mode de fonctionnement bien connu dans le langage C.

10.1.3. La classe *java.lang.System*

Exemple

```
System.out.println(Math.PI);
System.out.printf("%.2f",Math.PI);
```

Résultat affiché:

3.141592653589793
3,14

10.1.4. *La classe java.lang.Math*

- Contient une série de méthodes et attributs mathématiques.
- La classe `Math` fait partie du package java.lang, elle est automatiquement importée.
- Toutes les méthodes et attributs de la classe Math sont **static**.
- **Les attributs de classe Math sont:**

```
public static final double PI; //3,14159265358979323846
public static final double E;  //2,7182818284590452354
```

10. Les packages de base

- La classe Math définit de nombreuses méthodes mathématique:

abs (double a)	retourne la valeur absolue de a.(Surchargée pour les float, int et long)
sin (double a)	retourne le sinus d'un angle exprimés en radians. (Les autres fonctions trigonométriques sont disponibles :(`cos`, `asin`, `acos`, `tan` et `atan`).
max(double a, double b)	Retourne la valeur maximale des deux paramètres. (Surchargée pour les float, int et long)
min(double a, double b)	Retourne la valeur minimale des deux paramètres. (Surchargée pour les float, int et long)
sqrt (double a)	Retourne la racine carrée de son paramètre.
exp (double a)	Retourne l'exponentielle de l'argument.
log (double a)	Retourne le logarithme naturel de l'argument
random()	Retourne un nombre aléatoire compris entre 0.0 et 1.0.
…	**…**

Exemple

```
System.out.println(" exponentiel de 5  = "+Math.exp(5.0) );
System.out.println(" logarithme de 5  = "+Math.log(5.0) );
```

Résultat affiché:

exponentiel de 5 = 148.4131591025766

logarithme de 5 = 1.6094379124341003

10.2. Le package java.util

Ce package contient un ensemble de **classes utilitaires** :

- La gestion des dates (**`Date`** et **`Calendar`**),
- La génération de nombres aléatoires (**`Random`**),
- La gestion des collections (**`Arraylist`**, **`LinkedList`**, etc),
- etc ...

10.2.1. La classe java.util.Random

- La classe Random permet de générer des nombres pseudo-aléatoires.

boolean nextBoolean ()	Renvoie une valeur booléenne pseudo-aléatoire
double nextDouble ()	Renvoie une valeur double pseudo-aléatoire comprise entre 0.0 et 1.0
float nextFloat ()	Renvoie une valeur float pseudo-aléatoire comprise entre 0.0 et 1.0
int nextInt ()	Renvoie la prochaine valeur int pseudo-aléatoire
int nextInt (int n)	Renvoie une valeur int pseudo-aléatoire, comprise entre 0 (inclus) et la n (exclus)
…	

Exemple:

```
import java.util.Random;
public class Test {
  public static void main(String arg[])
    {
       Random r = new Random();
       double d=r.nextDouble();
       System.out.println("d = "+d);
       int a = r.nextInt();
       System.out.println("a = "+a);
       int b = r.nextInt(10);
       System.out.println("b = "+b);
    }
}
```

Le résultat affiché peut être:

d = 0.5699914909580789

a = -338360121

b = 4

10.2.2. La classe java.util.Scanner

- La classe Scanner est particulièrement utile pour réaliser une lecture de données à partir du clavier dans une application de type console.

Exemple

```
Import java.util.Scanner ;
public class MaClasse {
public static void main(String[] args)
 {
        Scanner sc = new Scanner(System.in);
        System.out.println("Veuillez saisir un mot :");
        String str = sc.next();
        System.out.println("Vous avez saisi : " + str);
        System.out.println("Veuillez saisir un entier :");
        int a = sc.nextInt();
        System.out.println("Vous avez saisi : " + a);
        System.out.println("Veuillez saisir un double :");
        double x = sc.nextDouble();
        System.out.println("Vous avez saisi : " + x);
}}
```