Rappel java :

Classe

Une classe est la description de données appelées **attributs**, et d'opérations appelées **méthodes**.

Une classe est un modèle de définition pour des objets ayant le même ensemble d'attributs, et le même ensemble d'opérations.

A partir d'une classe on peut créer un ou plusieurs objets par **instanciation** ; chaque objet est une **instance** d'une seule classe.

Les caractéristiques de la programmation objet sont :

- Encapsulation des données (attributs) et des comportements(méthodes) : les attributs et les méthodes sont définies dans le même environnement (capsule).
- Masquage de l'information : l'utilisateur de la classe peut ne pas avoir accès directement aux attributs.
- Héritage : on peut définir une classe à partir d'une autre classe.
- Polymorphisme : une même méthode peut avoir un comportement différent en fonction de l'instance à la quelle est appliquée.

1 Définition d'une classe :

```
class X extends Y{
   // liste des attributs
   // liste des méthodes
}
```

Si *Y* est la classe de base de la hiérarchie des objets Java *Object*, alors la partie *extends* est facultative.

exemple :

```
class A extends  Object{
   int a ;
   void f(){
      ...
   }
}
```

est équivalent à

```
class A {
   int a ;
   void f(){
      ...
   }
}
```

Une classe a une visibilité :

- **public** le mot *class* est alors précédé de *public*, tout utilisateur qui importe le paquetage peut utiliser la classe. Dans ce cas elle doit être définie dans un fichier qui a pour nom le nom de la classe.
- **privé** le mot *class* est alors précédé de *private*, seules des classes définies dans le même fichier peuvent utiliser cette classe.
- **paquetage** le mot *class* n'est pas précédé de mot particulier, toutes les classes du paquetage peuvent utiliser la classe.

2 Création d'instances

Pour créer une instance de la classe *A*, on écrira :

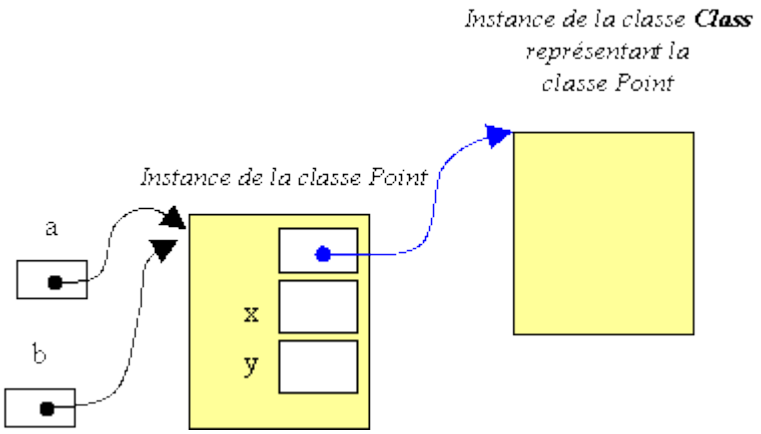
```
A a ;          // définition de la variable référence
a = new A() ; // création de l'instance
```

La création d'une instance par l'opérateur **new** se déroule en trois temps :

- Réservation de l'espace mémoire suffisamment grand pour représenter l'objet.
- Appel du constructeur de l'objet. Initialisation des attributs, et d'une référence à l'objet représentant la classe de l'instance en train d'être créée.
- Renvoi d'une référence sur l'objet nouvellement créé

```
class Point{
   int x ;
   int y ;
      . . .
}


Point a,  b ;
a = new Point(…) ;
b = a;
```



La destruction des instances se fait automatiquement par un « *thread* » le *garbage collector* qui cherche tous les objets qui ne sont plus référencés et les supprime. Le *garbage collector* peut être appelé directement par *System.gc()*.

3 Attributs et méthodes

3.1 Attribut :

Un attribut se définit en donnant son type, puis son nom, ([] pour un tableau), et éventuellement une partie initialisation.

Un attribut a une visibilité :

- **public** sa définition est précédée de *public*, et il peut être utilisé par tout utilisateur de la classe.
- **privé** sa définition est précédée de *private*, et il ne peut être utilisé qu'à l'intérieur de la classe
- **protégé** sa définition est précédée de *protected*, et il ne peut être utilisé qu'à l'intérieur de la classe, ou des classes dérivées.
- **paquetage** l'attribut peut être utilisé dans toute classe du même paquetage.

A l'intérieur de la définition d'une méthode, l'accès à un attribut se fait soit directement, soit en préfixant l'attribut par **this** qui est une référence à l'objet pour lequel est appelé la méthode.

A l'extérieur de la définition de la classe, l'accès se fait en écrivant le nom de l'instance, un point le nom de l'attribut ou de la méthode.

```
class Point{
   private int x ;
   public  int y ;
   void f(){
      x = 0;
      this.y = 1;
   }
}
```

En dehors de la définition de la classe :

```
Point p = new Point();
p.x = 0; // Erreur de compilation : la visibilité de x est privée !
p.y = 1;
```

3.2 Méthode :

Une méthode est définie par :

- Son type de retour : type de la valeur retournée par la méthode. Si la méthode ne retourne pas de valeur le type spécifié est alors **void**. Le type retourné peut être tableau.
- Son nom
- Ses paramètres : les paramètres sont spécifiés par leur type et leur nom (éventuellement []) et sont séparés par des virgules. Tous les paramètres sont des paramètres transmis par valeur. La plupart des paramètres (sauf ceux de type primitif) étant des références, la transmission de paramètre est en fait une transmission par référence : la valeur référencée peut être modifié, mais pas la référence elle même. On peut avoir un paramètre *ellipse*, et dans ce cas ce paramètre doit être le dernier de la liste des paramètres. Ce paramètre est en fait un tableau. L'appel de la méthode peut alors se faire avec un nombre variable de paramètres effectifs.

  Exemple :

```
void m(X ... e){
        e est un tableau à une dimension d'éléments de type X
}
```

Un appel de la méthode *m* se fait alors de la façon suivante :

```
    m(null);
    m();
    m(x1);
    m(x1, x2);
    m(x1, x2, x3)
    ...
    ce dernier appel est équivalent à :
    X[] t={x1, x2, x3};
    m(t);
    ou
    m(new X[]{x1, x2, x3});
}
```

Une méthode est caractérisée par sa ***signature*** :

1. son nom.
2. la liste des types des paramètres, dans l'ordre.

Dans une classe deux méthodes différentes ne peuvent pas avoir la même signature. Deux méthodes ayant le même nom, mais des liste de types de paramètres différentes sont des surcharges l'une de l'autre.

Dans le corps d'une méthode, on peut utiliser la référence **this**, qui référence l'objet pour lequel la méthode est appelée, cet usage étant facultatif. **this** ne peut pas être modifié.

Une méthode a une visibilité :

- **public** sa définition est précédée de *public*, et elle peut être utilisée par tout utilisateur de la classe.
- **privé** sa définition est précédée de *private*, et elle ne peut être utilisée qu'à l'intérieur de la classe
- **protégé** sa définition est précédée de *protected*, et elle ne peut être utilisée qu'à l'intérieur de la classe, ou des classes dérivées.
- **paquetage** la méthode peut être utilisée dans toute classe du même paquetage.

A l'extérieur de la définition de la classe, l'accès se fait en écrivant le nom de l'instance, un point le nom de l'attribut ou de la méthode.

4 Constructeurs, finalize

Certaines méthodes sont des **constructeurs** : ils ont pour nom le nom de la classe, ils ne retournent pas de valeur, et sont appelés au moment de la création d'une instance de la classe.

Si aucun constructeur n'est défini, Java fournit un constructeur sans paramètre appelé constructeur implicite ou par défaut, qui appelle le constructeur implicite de la super classe, les attributs ne sont pas initialisés.

Une classe peut être munie de la méthode **finalize()**. Cette méthode est appelée par le ramasse miettes (*garbage collector*) qui permet de récupérer l'espace occupé par les différents instances d'un programme qui ne sont plus utilisées.

5 Attributs et méthodes de classe.

Les attributs ou les méthodes d'une classe peuvent être **static**, et on parle de méthode ou d'attribut de classes, alors que pour les méthodes et attributs non statiques, on parle d'attribut et de méthode d'instances.

Pour les attributs, il n'y a qu'une seule représentation des attributs **static** et non pas une représentation par instance.

Un attribut, ou une méthode *static* peut être utilisé indépendamment de l'existence d'instances, en le préfixant par le nom de la classe. Une méthode *static* ne peut donc accéder qu'à des attributs ou méthodes *static*.

Exemples :

- La classe *Math* ne contient que des attributs statiques (*PI*, *E*) et des méthodes statiques(*sqrt*, *exp*, *sin*, *cos*, *log*, ...)

- La classe *BigInteger* contient deux attributs *static* (*ONE* et *ZERO*) et une méthode statique *BigInteger valueOf(long)*
- La classe *Font* contient plusieurs attributs *static* parmi lesquels *BOLD*, *ITALIC*, et *PLAIN* et des méthodes statiques parmi lesquelles *Font decode(String str)* qui fabrique une fonte à partie de son nom.
- La classe *Color* contient des couleurs prédéfinies *WHITE*, *RED*, *BLACK*, *BLUE*, ...

5.1 Bloc Static.

Une classe (non imbriquée) peut être munie de un ou plusieurs blocs *static* qui sont exécutés au chargement de la classe.

Un bloc *static* est défini par :

```
static{
    . . .
}
```

Un bloc statique peut servir à l'initialisation des attributs *static*, ou au chargement de librairies.

6 La classe Object

La classe *Object* est la classe racine de toute la hiérarchie des objets Java, y compris les tableaux.

`protected Object clone()`	Création d'une copie de l'objet. • Si l'objet n'implémente pas l'interface *Cloneable*, alors une exception *CloneNotSupportedException* est levée. Tous les tableaux implémentent l'interface *Cloneable*.

	• Sinon la méthode crée une nouvelle instance de la classe et initialise tous ses attributs avec les valeur des attributs de l'objet : si l'attribut est une référence c'est la référence qui est copiée : il n'y a pas **copie en profondeur**.
`public boolean equals(Object obj)`	Retourne *true* si un objet est égal à un autre. La méthode doit être : • Réflexive • Transitive • Symétrique
`protected void finalize()`	Appelée par le ramasse-miettes, quand il n'y a plus de référence à l'objet.
`public Class getClass()`	Retourne la classe de l'objet.
`public int hashCode()`	Retourne une valeur de *hash-code* pour l'objet.
`public void notify()`	Réveille un seul *thread* en attente sur le moniteur de l'objet.
`public void notifyAll()`	Réveille tous les *threads* en attente sur le moniteur de l'objet.
`public String toString()`	Retourne une représentation en chaîne de l'objet : la définition pour *Object* est `getClass().getName() + '@' + Integer.toHexString(hashCode()).`
`public void wait()`	Met le *thread* courant en attente jusqu'à ce qu'il soit réveillé par un *notify()* ou un *notifyAll()*
`public void wait(long timeout)`	
`public void wait(long timeout, int nanos)`	

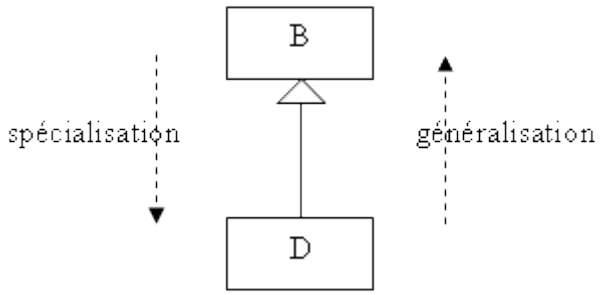
Héritage

1 Définitions.

Une classe Java dérive toujours d'un autre classe, *Object* quand rien n'est spécifié. Pour spécifier de quelle classe hérite une classe on utilise le mot-clé *extends* :

```
class D extends B {
    . . .
}
```

La classe *D* dérive de la classe *B*. On dit que le classe *B* est la super classe, la classe de base, ou la classe mère de la classe dérivée *D*, et que *D* dérive de *B*, ou que *D* est une sous-classe de *B*.



La visibilité *protected* rend l'accès possible :

- depuis les classes dérivée.
- depuis les classes du paquetage.

Exemple :

```
class B{
   private int a;

   protected int b;

   int c;

   public int d;
```

On peut alors représenter un objet de la classe D suivante :

```
}

class D extends B {

    int x;

    void f(){

        b = …;
    }
}
```

Une référence à une classe de base peut être affectée d'une référence à une classe dérivée, l'inverse doit faire l'objet d'une opération de conversion de type ou *cast* :

```
B b = new B();
D d = new D();
b = d;
d = b; interdit
d = (D) b ; // OK avec cast
```

Cette dernière instruction peut lever une exception *ClassCastException* si le cast est impossible.

2 Constructeurs et héritage.

- Si, dans une classe, un constructeur est défini sans commencer par un appel de constructeur, Java insère un appel du constructeur par défaut de la super classe.
- On peut commencer le code d'un constructeur par un appel d'un constructeur de la classe de base : `super(…)`, ou d'un autre constructeur de la classe `this(…)` alors Java n'ajoute rien.

```
class X{
    int x;

    public X
(int x){
        this.x
= x;
    }
```

```
class Y extends X{

}
```

provoque l'erreur de compilation : le super constructeur implicite *X()* n'est pas défini pour le constructeur par défaut.

```
}
```

3 this et super.

Chaque instance est munie de deux références particulières :

- **this** réfère l'instance elle-même.
- **super** réfère la partie héritée de l'instance.

4 Opérateur instanceof.

L'opérateur *instanceof* permet de savoir à quelle classe appartient une instance :

```
class B{ …}

class D extends B{…}

class C {…}

B b = new B();
D d = new D();
C c = new C();

b instanceof B // true
b instanceof D // false
d instanceof B // true
d instanceof D // true

b = d;

b instanceof B // true
b instanceof D // true
c instanceof B // erreur de compilation Erreur No. 365 :
               // impossible de comparer C avec B  en ligne …,
colonne ..
```

Exemple la méthode *equals(Object o)*, pour une classe *X*, est définie, en général, de la façon suivante :

```
public boolean equals(Object o){
   if(this==o) return true;
   if(!(o instanceof X) return false;
```

```
    X x = (X)o;
    ...
}
```

5 Redéfinition des méthodes

Une méthode définie dans une classe peut être redéfinie dans une classe dérivée.

Sans redéfinition de la méthode *m*	

`class B {`
`  void m(){`
`  System.out.println(`
`  "méthode m de B");`
`  }`
`}`
`class D extends B{`
`}`
	`cla`
	`}`
	`cla`

`B b=new B();`
`D d=new D();`
`b.m();`
`b = d;`
`b.m();`
	Affiche deux fois *m* de *B*

Remarques :

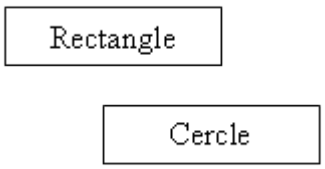
1. Ne pas confondre la redéfinition avec la surcharge :
   - **Redéfinition** : même type de retour et mêmes paramètres.
   - **Surcharge** : type de retour et paramètres différents, les paramètres au moins doivent être différents.
2. Une classe définie avec le modificateur d'accès *final* ne peut pas être dérivée.
3. Une méthode définie avec le modificateur d'accès *final* ne peut pas être redéfinie dans les classes dérivées. Ceci peut se faire pour des raisons :

- d'efficacité : le compilateur peur générer du code *inline*, sans faire appel à la méthode.
- de sécurité : le mécanisme de polymorphisme dynamique fait qu'on ne sait pas quelle méthode va être appelée.

6 Classe abstraite

Une classe définie avec le modificateur *abstract* est une classe abstraite qui ne peut pas produire d'instance. Sa définition peux contenir des méthodes abstraites. En revanche une classe qui contient une méthode abstraite doit être abstraite. Une méthode abstraite est une méthode définie uniquement par son prototype. Le rôle des classes abstraites est la factorisation de fonctionnalités communes à plusieurs classes dérivées.

Exemple : Supposons que nous voulions manipuler deux formes géométriques, rectangle (donné par son point haut, gauche, sa largeur et sa longueur) et cercle (donné par son centre et son rayon). Nous désirons pouvoir déplacer ces formes géométriques, calculer leur surface et leur périmètre .

sans classe abstraite	
	

```
class Rectangle{
   int x, y, largeur, longueur;
   Rectangle(int x, int y, int la, int lo){
      this.x = x; this.y = y;
      largeur = la; longueur = lo;
   }
   void deplaceDe( int dx, int dy){
      x+=dx; y+=dy;
   }
   double perimetre(){
      return 2*(largeur+longueur);
```

```
abs
}
```

```
    }
    double surface(){
       return largeur*longueur;
    }
}

class Cercle{
    int x, y, rayon;
    Cercle(int x, int y, int r){
       this.x = x; this.y = y;
       rayon = r;
    }
    void deplaceDe( int dx, int dy){
       x+=dx; y+=dy;
    }
    double perimetre(){
       return 2*Math.PI*rayon;
    }
    double surface(){
       return Math.PI*rayon*rayon;
    }
}
```

```
cla

}

cla

}
```

7 Exemples

7.1 Employé

Supposons que nous définissions la classe suivante :

```
public class Employe{
   protected String nom ;
   protected double salaire ;
   public Employe ( String nom, double salaire){
      this.nom = nom ; this.salaire = salaire ;
    }
   public String getNom(){ return nom ;}
   public double getSalaire() { return salaire ;}
   publicString toString() { return getNom()+" "+getSalaire();
}
}
```

Un chef «est un» employé qui a une prime en plus du salaire :

```
public class Chef extends Employe{
   private double prime;
   public Chef ( String nom, double salaire, double prime){
      super( nom, salaire) ;
      this.prime = prime;
   }
   public double getSalaire() {return salaire + prime;}
}
```

La méthode *getSalaire* pourrait être (mieux ?) programmée de la façon suivante : cela permettrait de changer les accès de nom et salaire en *private*.

```
public double getSalaire() { return super.getSalaire() + prime;}
```

Une entreprise est un tableau d'employés :

```
Employe entreprise[] = new Employe[5] ;
entreprise[0]= new Chef("Cronos", 1000, 500) ;
entreprise[1]= new Chef("Zeus ", 1000, 600) ;
entreprise[2]= new Employe("Ares", 620) ;
entreprise[3]= new Employe ("Apollon", 700) ;
entreprise[4]= new Employe ("Aphrodite ", 100) ;
```

L'affichage des noms et salaires des différents employés se fera de la façon suivante :

```
for ( int i = 0 ; i< 5 ; ++i){
   if (entreprise[i] instanceof Chef) System.out.print("Chef :
");
   System.out.println( entreprise[i].toString());
}
```

Quand on fait un appel de *toString* pour une instance de la classe *Chef*, le code de la méthode *toString* héritée de *Employe* s'exécute avec des appels de *getNom* et de *getSalaire* qui se font pour un *Chef* et ce sont les méthodes *getNom* et *getSalaire* de la classe *Chef* qui sont appelées.

Le calcul de la somme des salaires des employés qui ne sont pas des chefs :

```
double sommeSalaires = 0;
for ( int i = 0 ; I < 5 ; ++i)
   if(!(entreprise[i] instanceof Chef)) sommeSalaires +=
entreprise[i].getSalaire();
```

7.2 Des formes.

Nous nous proposons de définir une hiérarchie de classes permettant la visualisation d
Les classes *FormeA*, *Forme* et *BiPoints* sont des classes abstraites.

```
abstract public class FormeA {
   protected FormeA(){}
   abstract public void affiche(Graphics g);
   abstract public void cache(Graphics g);
   abstract public void deplaceDe(int dx, int dy, Graphics g);
}
```

```
abstract public class Forme extends FormeA{
   protected int x, y;
   protectedColor couleur;
   protected Forme(int x, int y, Color c){
      this.x = x; this.y = y;
      couleur = c;
   }
   public void deplaceDe(int dx, int dy, Graphics g){
      cache(g);
      x+=dx; y += dy;
      affiche(g);
   }
}
```

La classe *Point* est une classe concrète pour laquelle nous définissons les fonctionnalités *cache*, *affiche* et *deplaceDe* :

```
public class Point extends Forme {
   public Point(int int y, Color c) {
      super(x, y, c);
   }
   public void affiche(Graphics g) {
      g.setColor(couleur);
      g.drawRect(x, y, 1, 1);
   }
   publicvoid cache( Graphics g) {
```

```
        g.setColor(Color.white);
        g.drawRect(x, y, 1, 1);
    }
}
```

La classe *Cercle* est une classe concrète pour laquelle nous définissons les fonctionnalités *cache*, *affiche* :

```
public class Cercle extends Forme {
    int rayon;
    public Cercle(int x, int y, int r, Color c) {
        super( x, y, c);
        rayon = r;
    }
     public void affiche(Graphics g) {
        g.setColor(couleur);
        g.drawOval(x-rayon, y-rayon, 2*rayon, 2*rayon);
    }
    public void cache( Graphics g) {
        g.setColor(Color.white);
        g.drawOval(x-rayon, y-rayon, 2*rayon, 2*rayon);
    }
}
```

La classe abstraite *BiPoint* met en facteur le constructeur et la méthode *deplaceDe* de *Rectangle* et *Segment* :

```
abstract public class BiPoint extends Forme {
    int x1, y1;
    public BiPoint( int x, int y, int x1, int y1, Color c) {
        super(x, y, c);
        this.x1 = x1; this.y1 = y1;
    }
    public void deplaceDe(int dx, int dy, Graphics g) {
        cache(g);
        x+=dx; y += dy;x1+=dx; y1+=dy;
        affiche(g);
    }
}
```

Les classes *Rectangle* et *Segment* se différentient par le tracé :

```
public class Rectangle extends BiPoint {
    public Rectangle(int x, int y, int x1, int y1, Color c){
        super(x, y, x1, y1, c);<
    }
    public void affiche(Graphics g) {
        g.setColor(couleur);
```

```
        g.drawRect(x, y, x1-x, y1-y);
    }


    public void cache(Graphics g) {
        g.setColor(Color.white);
        g.drawRect(x, y, x1-x, y1-y);
    }
}

publicclass Segment extends BiPoint{
    public Segment (int x, int y, int x1, int y1, Color c){
        super(x, y, x1, y1, c);
    }
    public void affiche(Graphics g) {
        g.setColor(couleur);
        g.drawLine(x, y, x1-x, y1-y);
    }

    public void cache(Graphics g) {
        g.setColor(Color.white);
        g. drawLine (x, y, x1-x, y1-y);
    }
}
```

8 Processus de construction d'une instance

1. Chargement de la classe si elle n'est pas déjà chargée, avec :
    - Chargement de la classe de base (si elle n'est pas chargée)
    - Construction des attributs statiques et des blocs statiques de la classe dans l'ordre de leurs définitions(avec éventuellement chargement des classes et exécution de leurs statiques)
2. Allocation de la mémoire.
3. Validation du polymorphisme : pour toute méthode appelée par la suite il sera tenu compte du polymorphisme.
4. Appel du constructeur de la super classe
5. Initialisation des attributs (définis avec une initialisation)
6. Exécution du corps du constructeur.

Exemple soient les 3 classes suivantes :

```
public class Base {
    static{
```

```
        System.out.println("bloc statique Base ");
    }
    public Base() {
        System.out.println("constructeur de Base ");
    }
}
public class Derivee extends Base {<
    static {
        System.out.println("bloc statique 1 de Derivee ");
    }
    static UneAutre x1 = new UneAutre(1);
    UneAutre x2 = new UneAutre(2);
    static {
        System.out.println("bloc statique 2 de Derivee ");
    }
    public Derivee() {
        System.out.println("constructeur de Derivee ");
    }
}
public class UneAutre {
    static {
        System.out.println("statique de UneAutre ");
    }
    public UneAutre(int i) {
        System.out.println("constructeur de UneAutre "+ i);
    }
}
```

La création d'une instance de la classe *Derivee* produit :

```
bloc statique de Base       Exécution de 1.a
bloc statique 1 de Derivee  Exécution de 1.b
statique de UneAutre        Exécution de 1.b
constructeur de UneAutre 1  Exécution de 1.b
bloc statique 2 de Derivee  Exécution de 1.b
constructeur de Base        Exécution de 4
constructeur de UneAutre 2  Exécution de 5
constructeur de Derivee     Exécution de 6
```