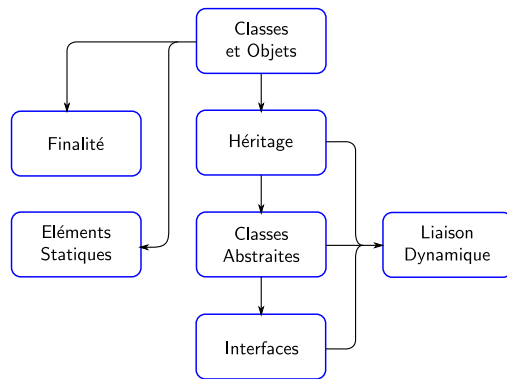


# Programmation en Java

V.Padovani, IRIF, Université Paris Cité



## 1. De C à Java

### 1.1. Eléments de base du langage

éléments communs à C et Java :

- notions de variable, de typage, d'affectation,
- types numériques `int`, `float`, `double`, etc.
- opérateurs arithmétiques `+`, `*`, `-`, `/`, `++`, `--`, etc.
- opérateurs de comparaison `<`, `<=`, `==`, `!=`, `>=`, `>`,
- conditions, connecteurs `||`, `&&`, `!`,
- structures `if/else`, `for`, `while`, `do/while`, `switch`,
- `?/:`, `break`, `continue`

les variables se déclarent à la volée ( $\neq$  C ANSI).

# PARTIE I - Classes et Objets

- Eléments communs à C et Java
- Classes, objets et références
- Initialisation des objets : les constructeurs
- Modification des objets : les méthodes.

- un type `boolean` de constantes `true`, `false`.

```
boolean b = (x == 3) && (y + 10 < z);
```

- Les conditions doivent être de type booléen :

```
if (x != 0) { ... } // et non if (x), comme en C.
```

- Un `for` peut avoir un compteur local (comme en C99) :

```
for (int i = 0; i < n; i++) {  
    // i declare et visible seulement  
    // dans le corps de la boucle,  
}
```

## 1.2. Syntaxe des tableaux

– déclarés et créés :

– à l'aide de l'opérateur `new` :

```
int[] t = new int[100];
```

– ou par initialisation explicite :

```
int[] u = {2,3,0,1};
```

même syntaxe d'accès qu'en C : `t[i] = 42;` etc.

– `int[]` est le type des tableaux d'entiers.

– Les tableaux sont gratuitement initialisés à 0.

– une forme spéciale de boucle permet de balayer le contenu d'un tableau :

```
int[] tab = {42, 17, 3};
```

```
// forme usuelle : pour chaque position i dans tab ...
for (int i = 0; i < tab.length; i++) {
    // ...
}
```

```
// forme alternative : pour chaque element v de tab ...
// (sans connaissance du rang de chaque element)
for (int v : tab) {
    // ...
}
```

– la taille des tableaux est libre (plus de `malloc/free`) :

```
double[] t = new double[n + 42];
```

– `t.length` vaut le nombre d'éléments de `t`.

– les accès sont vérifiés à l'exécution.

les accès invalides interrompent l'exécution.

```
char[] t = new char[100];
/*
 * t[-1] = 'a'; // erreur d'execution
 * t[100] = 'a'; // erreur d'execution
 */
```

## 1.3. Affichage

– équivalents de `printf` :

– `System.out.print`

– `System.out.println` (retour à la ligne)

– plus besoin de spécificateurs de format :

```
for (int i = 0; i < 10; i++) {
    System.out.print ("le carre de");
    System.out.print (i);
    System.out.print (" est ");
    System.out.println (i * i);
}
System.out.println ('a');
System.out.println (3.14); // etc.
```

- `System.out.print` ou `System.out.println` peuvent prendre plusieurs arguments séparés par des +...

```
for (int i = 0; i < 10; i = i++)
    System.out.println
        ("le carre de " + i + " est " + (i * i));
```

- l'association des + s'effectue de gauche à droite. entre deux valeurs numériques, + garde le sens de "addition".

```
int i = 2;
System.out.println (i + i);           // affiche 4.
System.out.println (i + " " + i)     // affiche 2 2
System.out.println (i + i + " " + i) // affiche 4 2
```

#### 1.4. Hiérarchie des types de base et conversions

- `boolean` (`true`, `false`)
- `byte` (8 bits, signé)
- `char` (16 bits, non signé) (i.e. jeu de caractères étendu)
- `short` (16 bits, signé)
- `int` (32 bits, signé)
- `long` (64 bits signé)
- `float`
- `double`

- il existe un type `String` pour les chaînes de caractères.
- + est l'opérateur de concaténation des chaînes.
- on peut concaténer une chaîne et une valeur numérique, convertie dans ce cas en `String`.

```
String s = "abc";
System.out.println (s); // affiche "abc"
s = s + "def";
System.out.println (s); // affiche "abcdef"
s = s + 42;
System.out.println (s); // affiche "abcdef42"
// etc.
```

- un argument numérique de `println` sera converti en `String` (les chaînes sont des *objets*, sens à préciser).

- les conversions vers un type plus petit doivent toujours être explicites.

```
/* erreur a la compilation
 * int x = 1;
 * char c = 'a' + x; // 'a' promu en int
 */
/* correct. d vaut 'b' */
int x = 1;
char d = (char) ('a' + x); // somme convertie en char.

/* erreur a la compilation. 2.5f n'est pas dans
 * les valeurs du type int :
 * int x = 2.5f;
 */
/* correct. x vaut l'arrondi 2 */
int x = (int) 2.5f;
```

- les conversions vérifiables dès la compilation sont inutiles :

```
/* correct : la valeur 'b' assignee peut etre
 * pre-calculée des la compilation, et cette valeur
 * reste dans les limites du type char */
char e = 'a' + 1;
```

- ++ et -- opèrent sans conversions :

```
char c = 'a';
c++;
/* c vaut 'b' */
```

```
char d = 'a';
/* erreur ! d promu en int, d + 1 en int
 * d = d + 1;
 */
```

```
/* correct */
d = (char) d + 1;
```

```
// fichier exemple.c
// definition d'un nouveau type de structure
struct point2D { // modele de structure a deux champs
    int abs; // de type int, indefinis par default
    int ord;
};
// fonction principale
int main (int argc, char *argv[]) {
    // declaration de pointeur vers struct point2D
    struct point2D *p;
    // stockage d'une adresse d'allocation dans p
    p = malloc (sizeof (struct point2D));
    // acces aux champs de la structure allouee via p
    (*p).abs = 42;
    (*p).ord = 10;

    free (p);
    return 0;
}
```

## 2. Objets, Classes et Références

### 2.1. Des structures de C aux classes de Java

- *objet*  $\equiv$  une généralisation de la notion de *structure* en C.

- comme une structure C, un objet

- encapsule des données désignées par des *noms de champs*,
- est créé suivant un certain *modèle* défini à l'avance.

modèle d'objet  $\equiv$  *classe* de cet objet.

objets d'une classe  $\equiv$  *instances* de cette classe.

- un programme Java est une suite de définitions de classes.

```
// fichier Exemple.java (meme nom que la classe principale)
// definition d'une nouvelle classe Point2D
class Point2D { // modele d'objet a deux champs
    int abs; // de type int, valeurs par default 0
    int ord;
}
// definition de la classe principale du fichier
public class Exemple {
    // "fonction" principale
    public static void main (String[] args) {
        // declaration de reference vers les objets de Point2D
        Point2D p;
        // creation d'une instance de Point2D,
        // stockage de son adresse dans p
        p = new Point2D ();
        // acces aux champs de l'objet via p
        p.abs = 42;
        p.ord = 10;
    }
}
```

- les valeurs de champs sont nulles à la création d'un objet, sauf si l'on spécifie explicitement d'autres valeurs :

```
class Point2D {
    int abs = 10;    // valeur par défaut pour chaque instance
    int ord;        // valeur nulle a la creation
}
```

### 2.3. Création et référencement des instances

- création d'une instance via `new` + nom de classe :

```
Point2D p;          // p de valeur indefinie

p = new Point2D (); // allocation d'un objet en memoire,
                    // retour de son adresse, stockage de
                    // cette adresse dans p

// p reference/designe/pointe a present vers l'objet cree

p.abs = 42;         // acces aux champs de l'objet
p.ord = 10;         // via la reference p.
```

### 2.2. Les références

- *référence* en Java :

- variable pouvant contenir l'adresse-mémoire d'un objet
- similaire à un pointeur vers structure de C, mais sans la notation `*` pour l'accès aux champs.

- la classe des objets sur lesquels pourra pointer une référence est spécifiée par sa déclaration:

```
nom_de_classe nom_de_reférence;
Point2D p;
```

(aucun objet créé, `p` est pour l'instant de valeur indéfinie).

- on peut donner à une référence la valeur nulle : `p = null;`.

- deux références peuvent référencer le même objet :

```
Point2D p, q;

p = new Point2D (); // nouvel objet, reference par p
q = p;              // p et q referencent a present le
                    // meme objet

p.abs = 42;         // acces a l'objet via p
q.ord = 10;         // acces au meme objet via q

System.out.println (" p.abs = " + p.abs); // 42
System.out.println (" p.ord = " + p.ord); // 10

System.out.println (" q.abs = " + q.abs); // 42
System.out.println (" q.ord = " + q.ord); // 10
```

- une même référence peut pointer vers **plusieurs** objets au cours du temps :

```
Point2D p = new Point2D (); // une 1ere instance creee
p = new Point2D ();        // une 2nde instance creee
```

La 1ère instance créée est définitivement perdue.

- mais pas de fuite de mémoire : les objets qui ne sont plus référencés seront tôt ou tard effacés de la mémoire ("garbage collecting" automatique).

```
Segment s = new Segment ();

// pour l'instant, s.debut et s.fin valent null.
// creation des extremités et cablage :

Point2D p1 = new Point2D ();
Point2D p2 = new Point2D ();
s.debut = p1; // s.debut et p1 designent le meme objet
s.fin   = p2; // s.fin   et p2 designent le meme objet

// creation directe des extremités :

Segment t = new Segment ();
t.debut = new Point2D ();
t.fin   = new Point2D ();
```

- les champs d'un objet peuvent être de type référence :

```
class Point2D {
    int abs;        // valeurs par défaut : 0
    int ord;
}

class Segment {
    Point2D debut; // valeurs par défaut : null
    Point2D fin;
}
```

la création d'un segment ne crée pas ses extrémités :  
il faut deux **new** supplémentaires par segment...

- on peut forcer la création des extrémités via l'initialisation des champs :

```
class Segment {
    Point2D debut = new Point2D (); // (re)-evalues a chaque
    Point2D fin   = new Point2D (); // creation de segment.
}

//...

// chaque creation de segment cree aussi deux points :
Segment s = new Segment ();
Segment t = new Segment ();
```

## 2.4. Remarques

- on confond souvent objet et référence vers objet :
  - "l'objet référencé par p", ou
  - "l'objet p"
  - "le point 2D p"
- une référence contient plus d'information qu'un pointeur :

```
Point2D p = new Point2D ();  
System.out.println (p);
```

affiche e.g. `Point2D@defala` (classe + adresse virtuelle).

## 3. Initialisation d'Objets : les Constructeurs

### 3.1. Définitions de constructeurs

- *constructeur*  $\equiv$  code d'initialisation placé dans une définition de classe. ce code peut être appelé au moment de la création d'une instance.
- chaque constructeur porte le même nom que la classe et peut avoir des paramètres.

```
class Point2D {  
    int abs;  
    int ord;  
    Point2D () {} // constructeur sans arguments  
    Point2D (int x, int y) { // constructeur a deux arguments  
        abs = x; ord = y;  
    }  
}
```

- l'appel d'un constructeur ne peut se faire qu'avec un `new`, au moment de la création d'une nouvelle instance :

```
// appel du 1er constructeur :  
Point2D p = new Point2D ();  
// appel du 2nd :  
Point2D q = new Point2D (42, 10);
```

- `new Point2D(42, 10)` :
  1. création (`new`) d'une nouvelle instance de `Point2D`
  2. exécution de `Point2D (int x, int y)` avec passage par valeur  $x \leftarrow 42$ ,  $y \leftarrow 10$ , et `abs` et `ord` désignant les champs de l'objet créé en (1).
  3. retour d'une référence vers l'objet créé.

```
class Point2D {  
    int abs;  
    int ord;  
    Point2D () {}  
    Point2D (int x, int y) {  
        abs = x;  
        ord = y;  
    }  
}
```

- codes équivalents :

```
– Point2D p = new Point2D ();  
  p.abs = 42;  
  p.ord = 10;
```

- et :

```
Point2D p = new Point2D (42, 10);
```

### 3.2. Constructeur par défaut et définitions multiples

- sans constructeurs définis, la classe est munie implicitement d'un **constructeur par défaut** (sans arguments, de corps vide).

```
class Point2D {
    int abs, ord;
}
```

est équivalent à

```
class Point2D {
    int abs, ord;
    Point2D () {} // pas d'arguments, et ne faisant rien.
}
```

(d'ou le `new Point2D ()` dans les exemples précédents)

- Un constructeur peut en appeler un autre par `this(...)` mais seulement en **1ère instruction** (pas nécessairement unique) de ce constructeur.

```
class Segment {
    Point2D debut;
    Point2D fin;
    Segment () {
        this (0, 0, 0, 0); // appel du constructeur
                        // en (int,int,int,int)
        // ... et éventuellement, autres instructions
    }
    Segment (int x1, int y1, int x2, int y2) {
        debut = new Point2D (x1, y1);
        fin   = new Point2D (x2, y2);
    }
}
```

- **signature** de constructeur  $\equiv$  suite des types de ses paramètres.
- on peut définir autant de constructeurs que l'on veut, pourvu que leurs signatures soient distinctes :

```
class Point2D {
    int abs;
    int ord;
    Point2D () {} // en ()
    Point2D (int x, int y) { // en (int,int)
        abs = x; ord = y
    }
    Point2D (Point2D p) { // en (Point2D)
        abs = p.abs; ord = p.ord; // copie des champs du
    } // point p
}
```

## 4. Modifications d'objets : les méthodes

### 4.1. Définitions de méthodes

- **méthode**  $\equiv$  code placé dans une définition de classe, permettant la modification des champs de ses instances

```
class Point2D {
    int abs;
    int ord;
    // ... constructeurs ...
    // exemples de méthodes :
    void assigner (int x, int y) { abs = x; ord = y; }
    void translator (int dx, int dy) {
        abs = abs + dx;
        ord = ord + dy;
    }
}
```



## 4.2. Invocations de méthodes

- l'exécution d'une méthode se fait toujours relativement à un objet via une référence : on dit que la méthode est *invoquée* sur l'objet.

```
Point2D p = new Point2D ();
p.assigner (42, 10);
p.translater (20, 20);
```

- `p.assigner (42, 10)` :

exécution de `assigner (int x, int y)`

avec passage par valeur  $x \leftarrow 42, y \leftarrow 10$ ,

où dans le code de la méthode,

`abs` et `ord` désignent les champs de l'objet référencé par `p`.

- une méthode peut renvoyer une valeur :

```
class Point2D {
    // ... champs abs, ord, constructeurs ...
    Point2D cloner () {
        Point2D clone = new Point2D ();
        clone.abs = abs; // les champs abs et ord de l'objet
        clone.ord = ord; // courant sont copiés dans le clone
        return clone;
    }
}
```

```
// ... et dans une autre partie du programme :
Point2D p = new Point2D (42, 10);
Point2D q = p.cloner (); // q est a present une copie de p
```

- une méthode peut invoquer une autre méthode (voire elle-même si elle est réursive) :

```
class Point2D {
    // ... champs abs, ord, constructeurs ...
    void assigner (int x, int y) {
        abs = x;
        ord = y;
    }
    void reinitialiser () {
        assigner (0, 0); // invoquee sur l'objet courant
    }
}
```

```
// ... et dans une autre partie du programme :
Point2D p = new Point2D (42, 10);
p.reinitialiser (); // equivalent a : p.assigner(0, 0)
```

- un constructeur peut invoquer une méthode :

```
class Point2D {
    int abs;
    int ord;
    Point2D () {}
    Point2D (int x, int y) {
        assigner (x, y); // invoquee sur l'objet qui vient
        // d'etre cree par new.
    }
    void assigner (int x, int y) {
        abs = x;
        ord = y;
    }
}
```

### 4.3. Surcharge de noms de méthodes

- **signature** de méthode  $\equiv$  suite des types de ses paramètres.
- deux méthodes peuvent porter le même nom à condition que leurs **signatures** soient distinctes. le nom de méthode est dit **surchargé**.
- lors de l'invocation, l'implémentation choisie sera celle qui correspond à la suite des types des arguments fournis.

- ...ce qui permet aussi d'écrire :

```
class Point2D {
    int abs;
    int ord;
    // ... constructeurs ...
    void assigner (int x, int y) {
        abs = x;
        ord = y;
    }
    void assigner (Point2D p) {
        assigner (p.abs, p.ord);
    }
    void assigner () {
        assigner (0,0);
    }
}
```

où les deux dernières méthodes en (**Point2D**) et en () appellent la première en (**int, int**).

```
class Point2D {
    int abs;
    int ord;
    // ... constructeurs ...

    void assigner (int x, int y) {
        abs = x;
        ord = y;
    }
    void assigner (Point2D p) {
        abs = p.abs;
        ord = p.ord;
    }
    void assigner () {
        abs = 0;
        ord = 0;
    }
}
```

### 4.4. Référence à l'objet courant : **this**

- **this** sans arguments dans un constructeur  $\equiv$  une référence vers l'objet qui vient d'être créé par **new**.
- **this** sans arguments dans une méthode  $\equiv$  une référence vers l'objet sur lequel cette méthode est invoquée.

```
class Point2D {
    //... l'écriture suivante est usuelle :
    Point2D (int abs, int ord) {
        this.abs = abs; this.ord = ord;
    }
    void set (int abs, int ord) {
        this.abs = abs; this.ord = ord;
    }
}
```

#### 4.5. Champs vs méthodes et masquage de champs

- tout champ, constructeur ou méthode précédé du mot-clef `private` est inaccessible à l'extérieur de sa classe.
- en pratique, on privilégie les `invocations de méthodes`, en déclarant souvent tous les champs privés :

```
class Point2D {
    private int abs; // representation interne des donnees,
    private int ord; // invisible depuis l'exterieur.
    // constructeurs + methodes d'accès aux champs
    ...
}
```

... et les noms `abs` et `ord` deviennent `inconnus` à l'extérieur de la classe `Point2D` : il *faudrait* passer par les méthodes pour accéder aux champs.

#### 4.7. Classe des points, version finale

```
public class Point2D {
    // champs privés
    private int abs;
    private int ord;

    // constructeurs
    public Point2D () {}
    public Point2D (int abs, int ord) {
        this.abs = abs;
        this.ord = ord;
    }
    // ...
}
```

#### 4.6. `private` v.s. `package` v.s. `public`

Forme standard d'un programme :

- Plusieurs répertoires, chacun formant un "package".
- Plusieurs fichiers dans chaque package,
- Une seule classe par fichier.
- Une seule méthode `main` dans l'une des classes d'un seul package.
- Par défaut, les éléments non privés d'un package (classes, champs, constructeurs, méthodes, etc.) ne sont visibles que dans ce package.
- On peut étendre leur visibilité à tout autre package en les faisant précéder du mot-clef `public` (`main` doit être `public`).

```
// methodes d'accès en lecture : nom standard = getChamp
public int getAbs () {
    return abs;
}
public int getOrd () {
    return ord;
}
// ...
```

```

// methodes d'accès en lecture : nom standard = setChamp
public void setAbs (int abs) {
    this.abs = abs;
}
public void setOrd (int ord) {
    this.ord = ord;
}
public void set (int abs, int ord) {
    this.abs = abs;
    this.ord = ord;
}
public void set (Point2D p) {
    this.set (p.abs, p.ord) // this facultatif, mais courant
}

```

– en pratique, les champs sont:

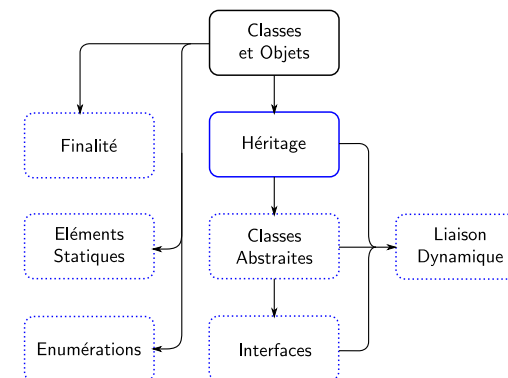
- inaccessibles (`private`),
- initialisés via les constructeurs,
- modifiés via les méthodes.

*i.e.* l'utilisateur n'a a priori aucune connaissance de la représentation interne des données, mais seulement de l'effet de l'invocation de tel ou tel constructeur ou méthode.

## 4.8. Conclusion

- une définition de classe contient :
  - des **champs** servant au stockage des données internes de ses instances,
  - du **code d'initialisation** placé dans des **constructeurs**, permettant l'initialisation automatique des champs lors de la création d'instances,
  - du **code de gestion** des données stockées dans les instances de cette classe, placés dans des **méthodes**, permettant de modifier les données internes d'un objet.

## PARTIE II - Héritage



## 5. Extensions de classes

### 5.1. Principe de l'extension de Classe

- *extension* de classe  $\equiv$  permet de définir une nouvelle classe
  - à partir d'une classe déjà écrite,
  - en récupérant les implémentations de ses méthodes,
  - en modifiant éventuellement certaines d'entre elles,
  - en ajoutant éventuellement d'autres champs et méthodes
- objectifs :
  - réutiliser du code déjà écrit,
  - adapter une classe générique à un besoin particulier,
  - écrire plusieurs variantes d'une même classe, en écrivant une seule fois le code commun à ses variantes.

```
Point2DColore pc = new Point2DColore ();
pc.abs = 10;
pc.ord = 42;
pc.col = 255;
pc.afficher ();           // affiche 10 42
pc.afficherCouleur ();   // affiche 255
```

les instances de `Point2DColore` disposent de tous les champs et méthodes de la classe `Point2D` :

- la classe `Point2DColore` hérite de ces champs et méthodes.

terminologie pour les liens de parenté :

- `Point2DColore` est une *extension*/une *classe héritière* de `Point2D`
- `Point2D` est une *classe parente*/un *ancêtre* de `Point2DColore`

### 5.2. Syntaxe de l'extension

```
class Point2D {
    int abs, ord;
    void afficher () {
        System.out.println (abs + " " + ord);
    }
}
```

... variante par extension (*extends*) : points colorés.

```
class Point2DColore extends Point2D {
    int col;
    void afficherCouleur () {
        System.out.println (col);
    }
}
```

### 5.3. Redéfinitions de méthodes

- Une extension peut *redéfinir* certaines des méthodes de sa classe parente :

```
class Point2D {
    int abs, ord;
    void afficher () {           // ancienne implementation
        System.out.println (abs + " " + ord);
    }
}

class Point2DColore extends Point2D {
    int col;
    void afficher () {           // nouvelle implementation
        System.out.println (abs + " " + ord + " " + col);
    }
}
```

l'implémentation invoquée pour une instance donnée sera toujours celle de sa classe :

```
Point2D p = new Point2D ()
p.abs = 23;
p.ord = 37
p.afficher (); // invoque l'implémentation de
                // la classe Point2D : "23 17"

Point2DColore pc = new Point2DColore ();
pc.abs = 10;
pc.ord = 42;
pc.col = 255;
pc.afficher (); // invoque l'implémentation de
                // la classe Point2DColore : "10 42 255"
```

#### 5.4. Accès aux anciennes implémentations : `super`

– lorsqu'une méthode est redéfinie par une extension, son ancienne implémentation reste accessible via `super`. + nom de méthode :

```
class Point2D {
    int abs, ord;
    void afficher () { System.out.println (abs + " " + ord); }
}

class Point2DColore extends Point2D {
    int col;
    void afficher () {           // nouvelle implémentation
        super.afficher ();      // appel de l'ancienne, puis
        System.out.println (col); // affichage supplémentaire.
    }
}
```

– Les méthodes de `nouvelles signatures` ajoutées à une extension sont considérées comme `nouvelles` :

```
class Point2D {
    int abs, ord;
    // methode set en (int, int)
    void set (int abs, int ord) {
        this.abs = abs; this.ord = ord;
    }
}

class Point2DColore extends Point2D {
    int col;
    // nouvelle methode set en (int, int, int)
    void set (int abs, int ord, int col) {
        set (abs, ord); // invocation du set en (int, int),
        herite.
        this.col = col;
    }
}
```

#### 5.5. Appel implicite du super-constructeur en ()

– par défaut, l'exécution d'un constructeur de `Point2DColore` est précédée de celle du constructeur en () de sa classe parente.

```
class Point2D {
    int abs, ord;
    Point2D () { System.out.println ("Point2D !"); }
}

class Point2DColore extends Point2D {
    int col;
    Point2DColore () {
        // appel implicite de Point2D () avant l'instruction qui
        // suit : new Point2DColore () affiche "Point2D !", puis
        System.out.println ("Point2DColore !"); }
}

...
Point2DColore pc = new Point2DColore ();
// affiche "Point2D !", puis "Point2DColore !"
```

- dans ce cas, un constructeur en `()` *doit* être défini dans la classe parente. l'erreur suivante est classique :

```
class Point2D {
    int abs, ord;           // pas de constructeur en ()
    ?!...
    Point2D (int abs,int ord) {
        this.abs = abs;
        this.ord = ord;
    }
}
class Point2DColore extends Point2D {
    int col;
    Point2DColore (int col) { this.col = col }
}

// erreur de compilation ! le constructeur de Point2DColore
// appelle implicitement Point2D () {}... non implemente.
```

```
class Point2D {
    int abs, ord; // valeurs nulles par défaut
    Point2D () {}
    Point2D (int abs,int ord) {
        this.abs = abs;
        this.ord = ord;
    }
}
class Point2DColore extends Point2D {
    int col;
    Point2DColore () {} // appel implicite de Point2D ()
    Point2DColore (int col) {
        this (0, 0, col) // appel de
                        // Point2DColore(int, int, int)
    }
    Point2DColore (int abs, int ord, int col) {
        super (abs, ord); // appel de Point2D (int, int)
        this.col = col;
    }
}
```

## 5.6. Appel explicite d'un autre constructeur

- un constructeur de `Point2DColore` peut demander explicitement l'appel d'un autre constructeur que celui en `()` de sa classe parente, via
  - `this (...)` (c.f. Partie I)
    - qui appelle le constructeur de la même classe spécifié par les types des arguments.
  - `super (...)`
    - qui appelle le constructeur de la classe parente spécifié par les types des arguments.
- dans les deux cas, l'appel doit être la *première* instruction du constructeur, et cette forme d'appel n'est pas renouvelable.

## 5.7. Héritage et masquage

- Les champs masqués (par `private`) d'une classe parente **ne sont pas hérités** par ses extensions – ils sont inaccessibles à l'extérieur de la classe parente...
- ... mais ces champs **existent** dans les instances de l'extension, même s'ils sont inaccessibles dans l'extension elle-même.
- ... et les méthodes de la classe parente héritées par l'extension continuent à accéder librement à ces champs (puisqu'elles les "voient" depuis la classe parente).

```

class Point2D {
    // abs et ord sont visibles dans cette classe,
    // pas dans les classes heritieres
    private int abs, ord;

    int getAbs () { return abs; }
    int getOrd () { return ord; }
    void set (int abs, int ord) {
        this.abs = abs;
        this.ord = ord;
    }
}
class Point2DColore extends Point2D {

    // ...
    void set (int abs, int ord, int col) {
        set (abs, ord); // invocation du set en (int,int) qui,
        this.col = col; // depuis sa classe, accede aux champs
    } // masques
}

```

- les constructeurs d'une classe ou ses méthodes peuvent également être déclarés `private` (plus rare).
- les règles sont les mêmes qu'avec les champs (non hérités, accessibles indirectement via les implémentations héritées).

```

Point2DColore pc = new Point2DColore ();
pc.set (10, 42, 255);
int i = pc.getAbs (); // 10
int j = pc.getOrd (); // 42

```

- les expressions `pc.abs` ou `pc.ord` sont incompilables, même si tout point coloré dispose bien de ces deux champs.
- les méthodes `getAbs`, `getOrd` sont héritées par `Point2DColore`. elles sont implémentées dans `Point2D`, donc accèdent librement aux deux champs depuis leur classe.
- la nouvelle méthode `set` invoque une implémentation de `set` héritée de `Point2D` qui accède librement aux deux champs depuis sa classe.

## 5.8. Extensions multiples et extensions d'extensions

- une même classe peut avoir un nombre arbitraire d'extensions.
- toute extension de classe peut elle-même être étendue.
- restrictions pour `super.+ méthode` et `super(...)` : une classe ne peut accéder qu'à l'implémentation (héritée ou non) d'un constructeur ou d'une méthode de son ancêtre immédiat : on ne peut pas écrire `super.super ...`



```

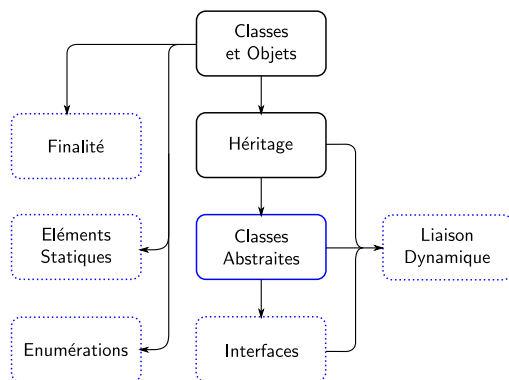
class Point2D {
    private int abs, ord;
    ...
}
class Point2DColore extends Point2D {
    private int col; // un champ de plus...
    ...
}
class Point2DColoreNomme extends Point2DColore {
    private char nom; // encore un champ de plus...
    ...
} // etc..

// Point2D est parente de Point2DColore et de
// Point2DColoreNomme,
// Point2DColore est parente de Point2DColoreNomme ...

```

- *hiérarchie de classes*  $\equiv$  une classe, et l'ensemble de ses descendants (directs ou indirects)
- une classe peut avoir plusieurs classes *ancêtres* (classe parente, classe parente de cette classe parente, etc.) par convention, les ancêtres d'une classe incluent *la classe elle-même*.
- une classe prédéfinie est implicitement ancêtre de toutes les autres : la classe `Object`.
- un programme java se conçoit en général en concevant un ensemble de hiérarchies de classes (classes génériques, spécialisations de ces classes, spécialisations des spécialisations...)

## PARTIE III - Classes Abstraites



## 6. Notion de classe abstraite

- *classe abstraite*  $\equiv$  classe incomplète dont certaines parties restent à écrire :
  - *non instanciable* (pas de `new` possible) mais elle peut contenir des champs, des constructeurs et des méthodes,
  - elle peut mentionner des noms de méthodes *non encore implémentées*, appelées méthodes *abstraites*,
  - une classe abstraite est uniquement destinée à être *étendue en classes concrètes* (instanciables), qui implémenteront les parties manquantes.
- objectif : toujours le même, écrire une seule fois les éléments communs à toutes les extensions, écrire ensuite les spécialisations concrètes.

## 7. Extension de classe abstraite - un exemple

- on souhaite définir une classe abstraite permettant de représenter des formes géométriques, sans précision de leur nature exacte (cercle, rectangle, carré ...).
- toute forme est définie relativement à un certain point de référence (deux champs `x` et `y` communs à toutes les formes).
- toute forme doit pouvoir afficher sa description (au moins les valeurs des champs `x` et `y`).
- toute forme occupe une certaine surface... qui ne peut se calculer qu'en connaissant la nature exacte de cette forme.

```
// fichier Rectangle.java
public class Rectangle extends Forme {
    private int hauteur, largeur; // champs ajoutés
    public Rectangle () {} // constructeurs
    public Rectangle (int x, int y, int hauteur, int largeur) {
        super (x,y);
        this.hauteur = hauteur; this.largeur = largeur;
    } // accesseurs ajoutés :
    public int getHauteur () { return hauteur; }
    public int getLargeur () { return largeur; }
    public void setDimension (int hauteur, int largeur) {
        this.hauteur = hauteur; this.largeur = largeur;
    }
    public void afficher () { // redefinition de l'affichage
        super.afficher (); // appel de l'implementation de Forme,
        System.out.println ("hauteur : " + hauteur);
        System.out.println ("largeur : " + largeur);
    }
    public double surface () { // implementation de la
        return hauteur * largeur; // methode abstraite de Forme.
    }
}
```

```
// fichier Forme.java
public abstract class Forme { // "abstract", i.e. abstraite
    private int x, y; // champs communs
    public Forme () {} // constructeurs
    public Forme (int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX () { return x; } // accesseurs pour les
    public int getY () { return y; } // champs communs
    public void setXY (int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void afficher () {
        System.out.println ("base : " + x + "," + y);
    }
    // methode abstraite : a implementer par les
    // heritiers concrets de cette classe abstraite.
    public abstract double surface (); // pas d'implementation.
}
```

```
// fichier Cercle.java
public class Cercle extends Forme {
    private int rayon; // champ ajoute
    public Cercle () {} // constructeurs
    public Cercle (int x, int y, int rayon) {
        super (x,y);
        this.rayon = rayon;
    }
    public int getRayon () { return rayon; } // accesseurs
    ajoutés
    public void setRayon (int rayon) {
        this.rayon = rayon;
    }
    public void afficher () { // redefinition de l'affichage
        super.afficher ();
        System.out.println ("rayon : " + r);
    }
    public double surface () { // implementation de la methode
        return Math.PI * rayon * rayon; // abstraite de Forme
    }
}
```

- règles d'héritage, de constructions, d'accessibilité des champs, etc. : les mêmes que pour une extension de classe concrète.
- toute classe abstraite ou concrète peut être **étendue** en une classe concrète ou abstraite, toutes les extensions sont extensibles.
- une **extension abstraite** peut laisser abstraites une partie des méthodes abstraites de sa classe parente et en introduire de nouvelles.
- seule contrainte sur les extensions : toute extension **concrète** doit fournir une implémentation de chaque méthode encore abstraite dans sa classe parente (héritée ou non).

## 8. Conversions de références

### 8.1. Conversion ascendante de référence

- **Règle de déclaration libre** : on peut déclarer des références vers n'importe quelle classe, même abstraite.

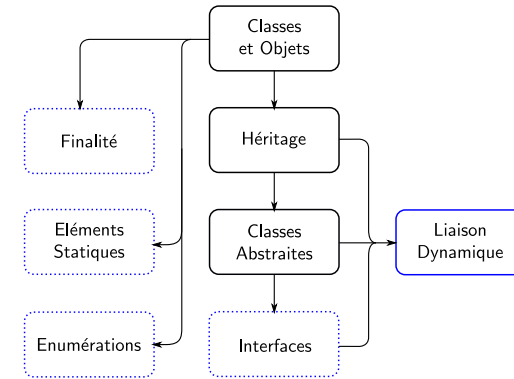
```
Forme f; // legal, meme si Forme est abstraite
```

- **Règle de conversion ascendante** : on peut référencer un objet par toute référence vers l'une de ses classes ancêtres.

```
Forme f;
Cercle c = new Cercle ();
Rectangle r = new Rectangle ();
```

```
f = c; // conversion "ascendante" de c
// en une reference vers Forme... legal.
f = r; // conversion "ascendante" de r... legal.
```

## PARTIE IV - Liaison dynamique



### 8.2. Conversion descendante

- **Règle de conversion descendante** : on peut convertir *explicitement* une référence dans la direction opposée, pourvu que la référence-cible soit vers une classe ancêtre de celle de l'objet.

```
Forme f = new Cercle ();
```

```
Cercle c = (Cercle) f; // valide
```

```
// Rectangle r = (Rectangle) f; // erreur !!
```

```
f = new Carre ();
```

```
Rectangle r = (Rectangle) f; // valide
```

- ces conversions **ne changent pas les classes des objets**, mais simplement le type des références qui servent à les manipuler.

### 8.3. Vérification de la validité d'une conversion

- on peut vérifier (à l'exécution seulement) la validité d'une conversion descendante à l'aide de `instanceof` :

```
Forme f = Math.random() > .5 ? new Cercle() : new Carre();
Rectangle r = null;
if (f instanceof Rectangle) {
    r = (Rectangle) f;
}
```

- `f instanceof Rectangle` vaut `true` si et seulement si `Rectangle` est ancêtre de la classe de `f` – i.e. si la conversion `(Rectangle) f` est valide.
- `instanceof` ne permet pas de déterminer la classe exacte d'un objet, mais simplement de vérifier si une classe est ancêtre de celle-ci. en pratique, son usage est réservé à la vérification de validité d'une conversion.

## 9. Liaison dynamique

### 9.1. Règle de liaison dynamique – la règle centrale

- lorsqu'une méthode est invoquée sur une référence, quel que soit son type, *l'implémentation choisie est toujours celle de la classe de l'objet référencé* (écrite dans sa classe, ou héritée).

```
Cercle c = new Cercle ();
Forme f = c;
f.afficher (); // dans les deux cas, l'implémentation
c.afficher (); // choisie est celle de Cercle.

f = Math.random () > .5 ? new Cercle () : new Rectangle ();
f.afficher (); // implémentation choisie a l'exécution !
```

`afficher ()` sera **lié** (associé) **dynamiquement** (à l'exécution) à une certaine implémentation, suivant la classe de l'objet.

### 8.4. Accessibilité des noms suivant le type des références

- lorsqu'un objet est référencé par une référence vers une certaine classe, **seuls les champs et méthodes de cette classe** (non masqués, dans la classe ou hérités) sont accessibles et invocables via cette référence.

```
Forme f = new Cercle ();

f.afficher (); // legal : afficher est dans Forme
// f.setRayon (10); // erreur !! Forme ne contient pas
// cette methode, definie dans Cercle.

Cercle c = (Cercle) f; // descente vers Cercle
c.setRayon (10); // legal.
```

```
// exemple precedent, tres simplifie :
abstract class Forme {
    abstract void afficher ();
}
class Rectangle extends Forme {
    void afficher () { System.out.println ("Rectangle"); }
}
class Cercle extends Forme {
    void afficher () { System.out.println ("Cercle"); }
}
Rectangle r = new Rectangle ();
Forme f = r;
f.afficher (); // affiche "Rectangle", comme r.afficher ()
Cercle c = new Cercle ();
f = c;
f.afficher (); // affiche "Cercle", comme c.afficher ()

Forme[] dessin = {new Rectangle (), new Carre ()};
for (int i = 0; i < 2; i++) dessin[i].afficher ();
// affiche "Rectangle" puis "Cercle"
```

```

// autre exemple : type de donnees abstrait
// fichier Pile.java
public abstract class Pile {
    abstract int pop ();
    abstract void push (int n);
    abstract boolean isEmpty ();
}

// ailleurs : classe(s) concrete(s) etendant Pile...
// implementation(s) par tableau, liste chaine, etc.

// ... et quelque part dans une autre classe :
void m (Pile p) {
    if (!p.isEmpty ()) { // invoquera les methodes de la
        int n = p.pop (); // classe (concrete) de p.
        // etc.
    } // la methode est dite "polymorphe" : elle accepte des
} // objets de classes differentes, en ne se servant que
// de leurs noms de methodes communs.

```

## 9.2. Règle de choix d'implémentation des méthodes

– suite d'extensions :  $A \rightarrow B \rightarrow C \rightarrow D$

```

// ... dans une classe E quelconque :
void m (A a) {...}
void m (C c) {...}

```

// ... et ailleurs :

```

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

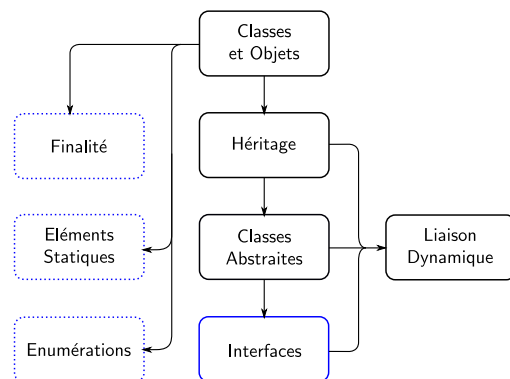
```

```

E e = new E(); // methode invoquee :
e.m (a); // m(A a) : A plus proche ancetre de A
e.m (b); // m(A a) : A plus proche ancetre de B
e.m (c); // m(C c) : C plus proche ancetre de C
e.m (d); // m(C c) : C plus proche ancetre de D

```

## PARTIE V - Interfaces



## 10. Notion d'interface

– **interface**  $\equiv$  similaire à une classe abstraite, mais encore plus incomplète. une simple liste de noms de méthodes.

```

// fichier Deplacable.java
public interface Deplacable {
    Point2D getPosition ();
    void setPosition (Point2D p);
    void alignerAvec (Deplacable d);
}

```

– une classe **implémente** une interface lorsqu' elle implémente chacune de ses méthodes. Les méthodes de l'interface **doivent** être **public**.

– une classe ne peut être l'extension que d'une seule classe, mais elle peut implémenter **plusieurs** interfaces.

```

// fichier Deplacable.java
public interface Deplacable {
    Point2D getPosition ();
    void setPosition (Point2D p);
    void alignerAvec (Deplacable d);
}

// fichier Segment.java
public class Segment implements Deplacable {
    private Point2D debut, fin; // + constructeurs, etc.
    public Point2D getPosition () {
        return debut;
    }
    public void setPosition (Point2D p) {
        debut.translater (p);
        fin.translater (p);
    }
    public void alignerAvec (Deplacable d) {
        setPosition (d.getPosition ());
    }
}

```

- pas de lien de parenté entre `Segment` et les héritiers concrets de `Forme...` mais "pont" créé entre toutes ces classes : une interface commune.
- `alignerAvec` accepte tout objet dont la classe implémente `Deplacable`. tous ces objets sont référençables par des références de type `Deplacable`.

```

Deplacable d = new Cercle (); // conversion ascendante
Deplacable e = new Segment (); // conversion ascendante
d.alignerAvec (e); // legal
e.alignerAvec (d); // legal

Segment s = new Segment (); // conversion ascendante
Cercle c = new Rectangle (); // de c en reference vers
s.alignerAvec (c); // Deplacable. legal.

```

```

// fichier Deplacable.java
public interface Deplacable {
    Point2D getPosition ();
    void setPosition (Point2D p);
    void alignerAvec (Deplacable d);
}

// fichier Forme.java
public abstract class Forme implements Deplacable {
    // champs x, y, constructeurs, etc.
    public Point2D getPosition () {
        return new Point2D (getX (), getY ())
    }
    public void setPosition (Point2D p) {
        setX (p.getX ()); setY (p.getY ());
    }
    public void alignerAvec (Deplacable d) {
        setPosition (d.getPosition ());
    }
}

// toute classe heritiere de Forme implémente Deplacable
// par heritage.

```

- une classe peut implémenter plusieurs interfaces :

```
class B extends C implements I, J, K {...}
```
- une classe `abstraite` peut être marquée comme implémentant une interface, mais ne pas implémenter toutes ses méthodes.
- les interfaces sont extensibles, et même multi-extensibles

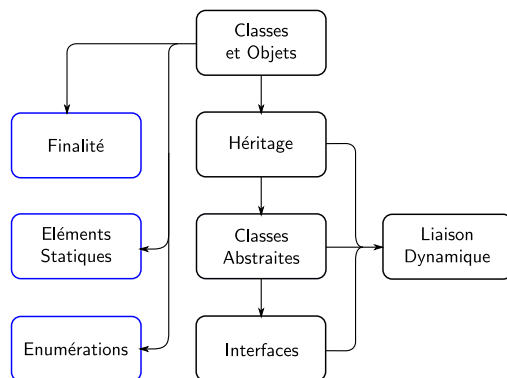
```
interface I { ... }
interface J { ... }
interface K extends I, J { ... }
```
- on peut ajouter des champs - implicitement `final` et `static` (*i.e.* des constantes).

- depuis Java 8 : une interface peut contenir une **implémentation par défaut** de certaines de ses méthodes. On peut y invoquer sur **this** toutes les méthodes de l'interface.

```
public interface Stack {
    int getSize ();
    void push (int n);
    int pop ();
    // empiler tous les elements d'un tableau
    default void push (int[] tab) {
        for (int n : tab) {
            this.push (n);
        }
    }
}
```

- toute classe implémentant l'interface hérite de cette implémentation par défaut, et peut la redéfinir. (**Stack.super.push** accède à l'ancienne implémentation)

## Partie VI - Compléments



- si une classe implémente plusieurs interfaces dont deux proposent une implémentation par défaut d'une *même* méthode, la classe *doit* la redéfinir (pour éviter tout conflit entre les implémentations par défaut).
- toujours depuis Java 8, une interface peut contenir des **méthodes statiques** (qui doivent être toutes implémentées, et qui ne peuvent accéder qu'aux éléments statiques de l'interface, champs statiques ou méthodes statiques).

## 11. Finalité de classes, champs, méthodes variables

- **final**  $\equiv$  plus modifiable, en plusieurs sens.
- **final class C {...}** : la classe **C** n'est plus extensible.
- **class C { final int x; ... }**  
le champ **x** doit être initialisé dès la création, et ne sera plus modifiable.
- **class C {... final int m () {...} }**  
le méthode **m ()** ne pourra pas être redéfinie dans les extensions de **C**.
- **final** permet de créer des objets **non-mutables**.
- variables en **final**  $\equiv$  **const** de C.

## 12. Champs et méthodes statiques

- **champ statique**  $\equiv$  partagé par toutes les instances d'une classe.
- un seul exemplaire de ce champ en mémoire. réservé dès le lancement du programme, et indépendamment de toute création d'objet.

```
// fichier Point2D.java
public class Point2D {
    // compteur commun a tous les points :
    static int nbreInstancesCrees = 0;
    private int abs, ord;
    public Point2D () { nbreInstancesCrees++; }
    public Point2D (int abs, int ord) {
        this ();
        this.abs = abs;
        this.ord = ord;
    }
}
```

- accès à un champ statique :

- via le **nom de la classe**
- ou via une référence initialisée dont la classe voit ce champ

```
// lancement du programme
System.out.println (Point2D.nbreInstancesCrees);
// affiche 0

// premiere instance creee
Point2D p = new Point2D ();
System.out.println (Point2D.nbreInstancesCrees);
// affiche 1

// etc.
```

- **méthode statique**  $\equiv$  une méthode qui n'accède qu'à des champs statiques. invocable indépendamment de tout objet.

```
// fichier Point2D.java
public class Point2D {
    // compteur commun a tous les points
    private static int nbreInstancesCrees = 0;
    private int abs, ord;
    public Point2D () {
        nbreInstancesCrees++;
    }
    // accesseur statique
    public static getNbrInstancesCrees () {
        return nbreInstancesCrees;
    }
}
```

- invocation via le **nom de la classe** (ou une référence initialisée):

```
System.out.println (Point2D.getNbreInstancesCrees ());
```

### 12.1. Applications des champs et méthodes statiques

- "bibliothèque" de fonctions utilitaires (comme en C)

```
// fichier Divers.java
public class Divers {
    public static final double Pi = 3.1415927;
    public static int max (int x, int y) {
        return x > y ? x : y;
    }
    public static int min (int x, int y) {
        return x < y ? x : y;
    }
    private Divers () {} // unique constructeur prive...
} // donc classe non instantiable,
// et non extensible !
```



- définitions de constantes, et même constantes d'objets :

```
// fichier Couleur.java
public class Couleur {
    public static final
        BLANC = new Couleur (255,255,255),
        NOIR = new Couleur (0,0,0);
    private final int r, v, b; // champs non mutables

    public Couleur () {
        this (0,0,0);
    }
    public Couleur (int r, int v, int b) {
        this.r = r;
        this.v = v;
        this.b = b;
    }
    // etc.
}
```

- classe singleton : une, et une seule instance.

```
public class Elvis {
    public static final THE_KING = new Elvis ();
    private Elvis () {} // prive... pas d'autre new possible
                        // pas d'extension possible.

    // etc.
}
```

- classe à deux éléments et seulement deux :

```
public class Boolean {
    private final boolean b;
    public static final // deux instances, et pas plus
        Boolean TRUE = new Boolean (true),
        FALSE = new Boolean (false);

    private Boolean (Boolean b) {
        this.b = b;
    }
    // "usine statique"
    public static Boolean getInstance (boolean b) {
        return b ? TRUE : FALSE;
    }
}
```

### 13. Enumérations

- Equivalent d'une classe dont les instances sont créés au lancement du programme, et qui n'est plus instanciable.

```
public enum Joueur {
    BLANCS, NOIRS;
}
```

équivalent à :

```
public class Joueur {
    public static final Joueur
        BLANCS = new Joueur (),
        NOIRS = new Joueur ();
    private Joueur () {}
}
```

- Les deux instances se manipulent via les deux références `Joueur.BLANCS`, `Joueur.NOIRS`

- Une énumération est une *classe*, on peut lui ajouter des champs, des méthodes, et des constructeurs implicitement privés, invoqués implicitement :

```
public enum Joueur {
    // invocations implicites de Joueur (String string)
    BLANCS ("blancs"), NOIRS ("noirs");

    private String string;

    Joueur (String string) {
        this.string = string;
    }
    public String toString() {
        return this.string;
    }
}
```

- "new Joueur ("...")" est interdit, même dans Joueur.

```
public enum Pion {
    PION_BLANC ("B", Joueur.BLANCS);
    PION_NOIR ("N", Joueur.NOIRS);

    public final Joueur joueur;
    public final String string;

    Pion (String string, Joueur joueur) {
        this.string = string;
        this.joueur = joueur;
    }

    public String toString () {
        return this.string;
    }
    public Joueur getJoueur () {
        return this.joueur;
    }
}
```

```
- public enum Dir {
    NORD, SUD, EST, OUEST
}
```

Une énumération est implicitement héritière de [Enum](#), et hérite de deux méthodes :

- `values()` : statique, tableau des éléments énumérés.  
`Dir.values()` → {NORD, SUD, EST, OUEST} en `Dir[]`
- `ordinal()` : rang d'un élément.  
`Dir.NORD.ordinal()` → 0,  
`Dir.SUD.ordinal()` → 1, etc.

- On peut donc aussi écrire :

```
// fichier Joueur.java
public enum Joueur {
    BLANCS, NOIRS;
}

// fichier Pion.java
public enum Pion {
    PION_BLANC, PION_NOIR;

    public static final String[] strings = {"B", "N"};
    public String toString () {
        return string[this.ordinal()];
    }
    Joueur getJoueur () {
        return Joueur.values()[this.ordinal()];
    }
}
```

- On peut définir un tableau ou une matrice de références vers les éléments d'une énumération (`contenu[i][j]` vaut `Pion.PION_NOIR`, `Pion.PION_BLANC`, ou `null` si la case est vide) :

```
public class Plateau {
    private Pion[][] contenu;
    private int hauteur;
    private int largeur;
    Plateau (int hauteur, int largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
        this.contenu = new Pion[hauteur][largeur];
    }
}
```

- Les champs d'une énumération sont le plus souvent non-mutables (privés, accesseurs en `set` mais pas en `get`), mais ce n'est pas une obligation.

- on ne peut pas définir une extension d'énumération (elle est implicitement `final`).
- une énumération ne peut pas hériter d'une autre classe (elle est implicitement héritière de la classe `java.lang.Enum`).
- mais une énumération peut implémenter une ou plusieurs interfaces (dans ce cas, des implémentations par défaut simulent plus ou moins le mécanisme d'héritage).