Constructeurs

Pour simplifier l'initialisation, on peut utiliser un *constructeur*.

Avant :

```
public class Point
{
    private double x,y;  // les coordonnees
    public void initialise(double nX, double nY) { x=nX; y=nY; }
}
```

Après :

```
public class Point
{
    private double x,y;  // les coordonnees
    public Point(double nX, double nY) { x=nX; y=nY; }
}
```

Quelle est la différence ?

Constructeurs

Utilisation sans constructeur :

```
Point A = new Point();
A.initialise(10.,20.);
```

Utilisation avec constructeur :

```
Point A = new Point(10.,20.);
```

En pratique : Les constructeurs sont très utiles.

Exemple : une classe `EnsemblePoints` avec propriété `Point[] points` — on ne veut pas permettre `points==null`

Point IV

Pour déplacer et afficher un point :

```
public class Point
{
    ...
    public void deplace(double dx, double dy)
    {
        x+=dx;
        y+=dY;
    }
    public void affiche()
    {
        System.out.println("( " + x + " , " + y + " )");
    }
    ...
```

Pour utiliser :

```
    Point A = new Point(10.,20.);
    A.deplace(5.,-7.); A.affiche();
```

Méthode de classe, méthode de l'objet

Déclaration :

```
public Type-de-valeur-retournée Nom-de-la-fonction(Liste-d'arguments)
```

Ou bien `private Type-de-valeur-retournée ...`

Exemples : `public int length()` ou

```
private Classe[] creditsReussis(int annee)
```

Exécution : `objet.Nom-de-la-fonction`

Exemple : `S.length()`

Références : copie d'objets

```
Point A = new Point(3,5);
Point B;
```

Si on veut que B soit une copie de A, on peut faire `A=B;` mais dans ce cas les deux variables réfèrent au même objet. Si on veut deux objets, alors il faut copier explicitement...

```
public class Point
{
    ...
    public Point copie()
    {
        Point p = new Point(x,y);
        return p;
    }
```

Utilisation :

```
Point A = new Point(3.,5.);
Point B = A.copie();
```

Références : comparaison d'objets

```
Point A,B;
```

On veut comparer A et B. Si on utilise `A==B` ou `A!=B`, on compare les adresses des objets, pas les objets eux-mêmes !

Comment faire ?

```
public class Point
{
    ...
    public boolean estEgal(Point B)
    {
        return (x == B.x) && (y == B.y);
    }
```

(Notez que la méthode en `A` peut accéder aux variables privées de `B` dans la même classe)

Utilisation :

```
if( A.estEgal(B) ) { ... }
```

Références : comparaison d'objets

```
Point A,B;
```

On peut aussi utiliser une méthode de classe.... le bon vieux `static`, utilisé depuis le début du cours...

```
public class Point
{
...
    public static boolean compare(Point A,Point B)
    {
        return (A.x == B.x) && (A.y == B.y);
    }
}
```

(Notez que la méthode statique peut accéder aux variables privées dans la même classe)

Utilisation :

```
if( Point.compare(A,B) ) { ... }
```

String

java.lang

Class String

Constructor Summary
String() Initializes a newly created String object so that it represents an empty character sequence.
String(char[] value) Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
String(String original) Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

int	**length**() Returns the length of this string.

Vérifier si deux `String`s sont égaux : `S1.equals(S2);`

Différence entre `Nom.equals("Colette")` et `"Colette".equals(Nom)` ?

→ si `Nom==null`, alors le premier donne une erreur (`NullPointerException`)

Droits d'accès

`public` et `private` sont des mots-clé qui définissent les droits d'accès

`private` accès permis seulement dans la classe de déclaration

`public` accès permis pour tout le monde

Ces modificateurs de droits d'accès s'appliquent sur *tout* (variables d'instance ou de classe, méthodes, constructeurs).

Exemple

```
// dans le fichier Bidon.java
public class Bidon
{
  public int contenant; // accessible de partout
  private int contenu;  // accessible seulement dans la classe

  private void init()
  {
    contenant = 1;
    contenu = 2; // accèss légal, init() est dans la classe Bidon
  }

  public Bidon()
  {
    init(); // opération légale, on est dans la classe
  }
}
```

Exemple — cont.

```
// dans le fichier TestBidon.java
public class TestBidon
{
  public static void main( String[] args )
  {
    Bidon b = new Bidon(); // constructeur publique => ok

    b.init(); // méthode privée => illégale! NE COMPILE PAS!

    // publique donc accès permis:
    System.out.println( "contenant = " + b.contenant );

    // privé donc accès non permis. NE COMPILE PAS!
    System.out.println( "contenu = " + b.contenu );
  }
}
```

un accès illégal est détecté lors de la compilation → pas de bytecode

Pourquoi utiliser `private` ?

Raisons pour avoir des variables privées :

- pour protéger les données de l'objet, de sorte qu'il ait plein contrôle sur la valeur de ses attributs (son état).
- on procède par méthodes publiques pour consulter ou modifier l'objet.

Raisons pour avoir des méthodes privées :

- pour définir des opérations que seul l'objet même peut faire, car autrement l'opération pourrait invalider son état, ou alors parce qu'il s'agit d'une opération interne qui n'a pas de raison d'être publique.
- tout comme la protection des attributs, on protège certaines opérations pour que l'objet contrôle son état.

Raisons pour avoir des constructeurs privés :

- on n'a que des méthodes statiques (constructeur publique sans arguments est là par défaut)
- instanciation spéciale qu'on ne veut pas permettre à l'extérieure de la classe