Programmation en Java 2008-2009
Frédéric Mallet & Jean-Paul Roy
L1 Info&Math - Faculté des Sciences de Nice
http://deptinfo.unice.fr/~roy

Compléments sur les objets

L'objet courant : this

- Reprenons deux méthodes de la classe `Accidents` du TP8.

```
class Accidents {
  int[] data;

  ......

  void afficher() {
    for (int i = 0; i < data.length; i = i + 1) {
      System.out.print(data[i] + " ");
    }
    System.out.println();
  }

  public String toString() {
    String res = "{";
    for (int i = 0; i < 11; i = i + 1) {
      res = res + data[i] + ",";
    }
    res = res + data[11] + "}";
    return res;
  }
}
```

- La méthode `afficher()` peut-elle utiliser la méthode `toString()` ?

- Dans le texte d'une classe, une méthode peut appeler une autre méthode simplement **en tant que fonction** :

```
void afficher() {
  System.out.println(toString());
}
```

- Si l'on pense à la méthode **en tant que message** envoyé à un objet, à qui est envoyé le message `toString()` ? *A* **l'objet courant** !

- Qui est **l'objet courant** ?

  - dans une méthode : l'objet qui va exécuter la méthode

  - dans un constructeur : l'objet en cours de construction

- L'**objet courant** se nomme ***this*** en Java. Il est **sous-entendu**.

```
void afficher() {
  System.out.println(this.toString());
}
```

```
void afficher() {
  System.out.println(this);
}
```

Comment nommer les paramètres d'un constructeur ?

- Souvenez-vous du constructeur de la classe `Balle` (cours 7 p. 15) :

```
class Balle {
  int r, y, vy;    // champs : rayon, position, vitesse verticale

  Balle(int rInit, int yInit, int vyInit) {
    r = rInit;
    y = yInit;
    vy = vyInit;
  }
  .........
}
```

- Le nom des paramètres n'a pas d'importance... jusqu'à un certain point. Peut-on les nommer `r`, `y`, `vy` comme les champs ?

```
Balle(int r, int y, int vy) {
    r = r;
    y = y;
    vy = vy;
}
```

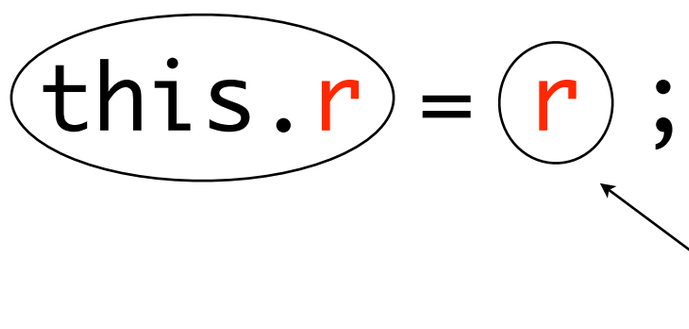
Oups ! Ce contructeur... ne fait rien !

• Pourtant, pour ne pas multiplier les noms de variables, les programmeurs Java utilisent souvent les noms des champs. Ils lèvent l'ambiguité avec le mot ***this*** :

```
class Balle {
  int r, y, vy;    //  champs : rayon, position, vitesse verticale

  Balle(int r, int y, int vy) {

    this.r = r;
    this. y = y;
    this. vy = vy;
  }
  .........
}
```



`this.r` : le champ `r` de l'objet courant `this`

`r` : le paramètre `r` du constructeur

La surcharge des constructeurs

- Nous avons déjà rencontré des classes avec plusieurs constructeurs :

```
class Point {
  int x, y;

  Point(int a, int b) {
   x = a;
   y = b;
  }

  Point() {
   x = 0;
   y = 0;
  }

  Point(Point p) {
    x = p.x;
    y = p.y;
  }

  ........
}
```

Point(int, int)

Point()

Point(Point)

***surcharge** ou*
surdéfinition

• Notez au passage qu'il ne serait pas possible de programmer deux constructeurs `Point(int,int)` car lors de l'appel :

`Point p = new Point(5,8)`

Java ne saurait pas lequel choisir !

• Les deux derniers constructeurs sont clairement des cas particuliers du premier. Comment **dans un constructeur faire appel à un autre constructeur de la même classe** ? En utilisant encore le mot ***this***, mais suivi cette fois d'une parenthèse :

• Si d'autres instructions doivent être exécutées pour compléter la construction, il faut que `this(...)` soit en première ligne du constructeur !

```
class Point {
  int x, y;

  Point(int a, int b) {
   x = a;
   y = b;
  }

  Point() {
   this(0,0);
  }

  Point(Point p) {
    this(p.x, p.y);
  }

  ........
}
```

N.B. Si une classe `Truc` *n'a pas de constructeur, le constructeur par défaut est* `Truc()`.

La notation pointée

- Que peut signifier le **point** en Java ?

  - ce peut être la virgule décimale d'un nombre approché :

```
3.14
```

  - ce peut être l'accès à un champ d'un objet :

```
Point p = new Point(5, 18);
int abscisse = p.x;
```

- ce peut être l'envoi d'un message (méthode) à un objet :

```
TicketMachine tm = new TicketMachine(6);
tm.insertMoney(4);
```

L'accès permissif aux champs d'un objet

• Dans la classe `Point` (celle que nous avons programmée ou celle de l'API), il est possible d'accéder aux champs `x` et `y` d'un point ou de les modifier directement en-dehors de la classe `Point` :

```
class Dessin {
   Point p = new Point();
   Dessin(int a, int b) {
      p.x = a;
      p.y = b;
   }
   .........
}
```

• Or VOUS êtes un objet, et votre portefeuille est l'un de vos champs ! Imaginez que quelqu'un vienne modifier de l'extérieur le contenu de votre portefeuille...

La paranoïa : public ou privé ?

• A la fois pour des raisons de **sécurité**, et de solidité du logiciel (il y a des *règles de l'art*), Java est amené à distinguer le caractère **public** ou **privé** des classes, des champs, des constructeurs et des méthodes !

• Lorsqu'on ne précise rien (comme jusqu'à présent), il s'agit d'un niveau intermédiaire entre **public** et **private**. Disons *quasi-public* ?...

• En Processing, on ne précise rien : tout est *quasi-public* par défaut...

Sécurité des classes

• Nous n'écrirons qu'**une seule classe par fichier**, et cette classe sera **publique**.

```
public class TicketMachine {
   .......
}
```

• Si elle était privée, vous ne pourriez pas l'utiliser de l'extérieur !

Sécurité des champs

• La classe `Point` de l'API Java a des champs `x` et `y` publics de manière à simplifier la vie du programmeur... L'abscisse du point `p` est `p.x` et vous pouvez modifier `p.x`

• Ce n'est pas le cas général. Il est considéré comme propre d'avoir le **maximum de champs privés**, et de prévoir des méthodes d'accès ou de modification pour les champs que l'on accepte de laisser consulter ou modifier de l'extérieur.

```
public class MyPoint {
  private int x;
  private int y;

  MyPoint(int x, int y) {
    this.x = x;
    this.y = y;
  }
  ........
}
```

• La classe `MyPoint` est publique : n'importe qui peut l'utiliser. Mais certaines portions de cette classe **ne sont directement utilisables que dans le texte de la classe**, pas dans une autre classe. C'est ce que signifie le mot **private** !

• Les champs x et y étant privés, il faut prévoir deux méthodes d'accès (des **accesseurs** aux champs) :

```
public int getX() {
  return this.x;
}
```

```
public int getY() {
  return y;
}
```

• Si l'on autorise une classe extérieure à modifier les champs d'un point, on programmera des méthodes ad-hoc (**modificateurs**) :

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

```
public int setY(int new_y) {
  y = new_y;
}
```

• Les accesseurs et modificateurs sont particulièrement intéressants s'ils font autre chose (par exemple compter le nombre de modifications, afficher un avertissement, etc).

Sécurité des constructeurs

- En principe **les constructeurs sont publics**.

Sécurité des méthodes

- Une méthode `truc()` peut faire appel à une autre méthode `machin()` de la même classe, mais l'utilisateur de la classe n'a besoin que de connaître `truc()`.

Dans ce cas, on déclare que la **méthode purement utilitaire** `machin()` est **privée**.

N.B. La méthode `toString()` *est* ***toujours publique*** *! En effet, Java en propose une par défaut (peu d'intérêt) qui est publique, et on ne peut pas restreindre les droits d'accès...*

```
public class Toto {
  ........
  public int truc() {
    return 35 - 2 * machin();
  }
  private int machin() {
    return ...
  }
}
```

```
public class Titi {
  ........
  public int chose() {
    Toto t = new Toto();
    int a = t.truc();   // GOOD
    int b = t.machin(); // BAD
  }
}
```

Les variables constantes (!)

• Pour se protéger soi-même et éviter de modifier quelque part une variable (en général un **champ**) qui est en réalité **constante**, on utilise le mot `final`. Le nom de la constante s'écrit ENTIEREMENT EN MAJUSCULES. Les variables non `final` commencent par une minuscule !

```
public class Toto {
  final double TAUX_TVA = 1.186;

  .........
  double prixTTC(double prix) {
    return prix * TAUX_TVA;
  }
}
```

```
TAUX_TVA = TAUX_TVA * 0.1;
```

Impossible !

• Notez que le champ `length` d'un tableau est `public` puisqu'on peut y accéder sans passer par une méthode accesseur. Mais il est déclaré `final`, on ne peut pas le modifier :

```
t.length = 3;
```

Error: cannot assign a value to final variable length

Les champs et méthodes de classe (static)

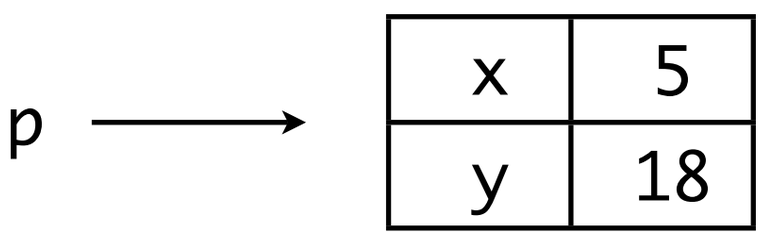
- Dans la modèle d'objets présenté jusqu'à présent :

  - les **champs** d'un objet décrivaient **l'état de l'objet** à un instant donné.

  - les **méthodes** d'un objet permettent d'envoyer un **message à cet objet** qui est dans un **état** donné, et qui va peut-être modifier son état en réaction au message.

```
public class Point {
  int x, y;
  .........
  void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

```
Point p = new Point(5, 18);
```



objet = instance d'une classe

- Les programmeurs parlent de **champs et méthodes d'instance**.

Les champs de classe

• Problème : dans une classe `Cercle`, je souhaite compter le nombre de cercles construits dans mon logiciel. Où placer un compteur ?

• Sûrement pas dans un cercle ! Une fois construit, il n'est plus au courant de ce qui se passe dans l'usine à cercles.

• C'est à l'usine que l'on sait combien de cercles l'on a construit. Mais **l'usine, c'est la classe** `Cercle` !

• Le compteur, qui sera incrémenté à chaque construction d'un nouveau cercle, sera un **champ de la classe** et non un champ d'instance. Sa déclaration sera précédée du mot `static`.

```
public class Cercle {
  private int rayon;
  private Point centre;
  private static int nbCercles = 0;

  public Cercle (int rayon, int centre) {
    this.rayon = rayon;
    this.centre = centre;
    nbCercles = nbCercles + 1;
  }
  ........
}
```

Les méthodes de classe

• Problème : dans ma classe `Cercle`, je souhaite écrire une fonction factorielle. Est-ce une méthode d'instance ? Dépend-elle de l'état d'un objet cercle, de son rayon, de son centre ? Non.

• Ce sera donc une **méthode de classe**. C'est d'ailleurs assez typique de la plupart des fonctions purement mathématiques.

• Le choix ***private*** pour le champ et la méthode statiques est négociable : leur usage est-il restreint au seul texte de cette classe ?...

```
public class Cercle {
  private int rayon;
  private Point centre;
  private static int nbCercles = 0;

  public Cercle (int rayon, Point centre) {
    this.rayon = rayon;
    this.centre = centre;
    nbCercles = nbCercles + 1;
  }

  private static int fac (int n) {
    if (n <= 0) return 1;
    return n * fac(n - 1);
  }
  ..........
}
```

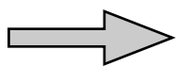
Comment accéder aux champs et méthodes de classe ?

- Dans le texte de la classe elle-même, directement :

```
int n = fac(5);
if (nbCercles > n) .....
```

- Bien entendu, on n'écrit pas n'importe quoi :

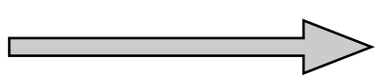
```
static void agrandir() {
  this.rayon = 5;
}
```



Error: non-static variable this cannot be referenced from a static context !

- On peut considérer que les champs et méthodes de classe sont communs à tous les objets de la classe. Les objets les connaissent. Mais il est fortement déconseillé d'y accéder en passant par les objets :

```
public void verifier() {
  int n = this.fac(5);
  if (this.nbCercles > n) ....
}
```



Le compilateur accepte, mais c'est très laid !

```
public class TestCercle {
  public void go() {
    Cercle c = new Cercle(...);
    int n = Cercle.fac(5);
    if (Cercle.nbCercles > n) ...
  }
}
```

- A partir du texte d'une autre classe, il faudra spécifier la classe d'origine du champ ou de la méthode de classe avec la **notation pointée**.

```
public class Cercle {
  private int rayon;
  private Point centre;
  private static int nbCercles = 0;

  public Cercle (int rayon, Point centre) {
    this.rayon = rayon;
    this.centre = centre;
    nbCercles = nbCercles + 1;
  }

  private static int fac (int n) {
    if (n <= 0) return 1;
    return n * fac(n - 1);
  }
  .........
}
```

	d'instance	*de classe*
champ	`objet.champ`	`Classe.champ`
méthode	`objet.méthode(...)`	`Classe.méthode(...)`

Exemple : la classe Math de l'API

- C'est une **classe sans objet**, tous ses champs et méthodes sont ***static*** !

Field Summary	
static double	**E** The double value that is closer than any other to *e*, the base of the natural logarithms.
static double	**PI** The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary	
static double	**abs**(double a) Returns the absolute value of a double value.
static float	**abs**(float a) Returns the absolute value of a float value.
static int	**abs**(int a) Returns the absolute value of an int value.
static long	**abs**(long a) Returns the absolute value of a long value.
static double	**acos**(double a) Returns the arc cosine of an angle, in the range of 0.0 through *pi*.
static double	**asin**(double a) Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.

```
double pi_sur_n = Math.PI / n;
if (Math.cos(pi_sur_n) < 0.3) ...
```