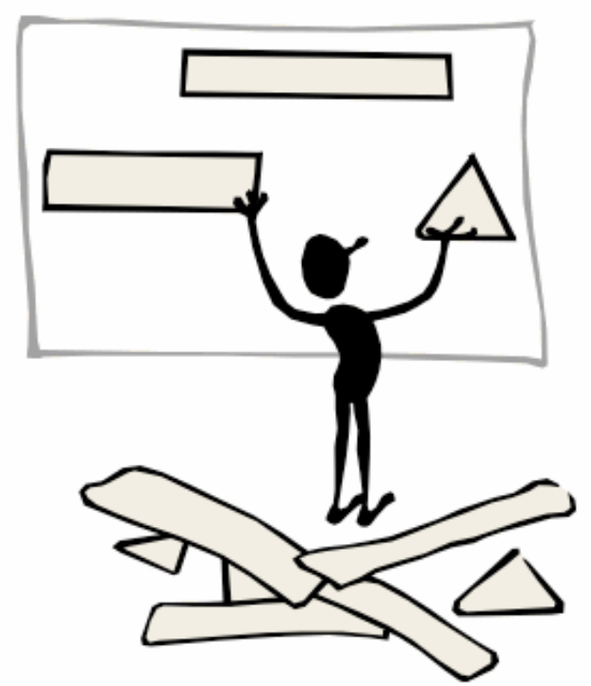
Programmation en Java 2008-2009
Frédéric Mallet & Jean-Paul Roy
L1 Info&Math - Faculté des Sciences de Nice
http://deptinfo.unice.fr/~roy

Programmer ses propres classes



Rappels : objets et classes (vocabulaire)

- Le langage Java organise ses données sous la forme :
  - de **données primitives** (`int`, `float`, `boolean`...)
    Par exemple `56.021` *est une donnée de type* `float`.
  - d'**objets**. Le type d'un objet est sa **classe**.
    Par exemple "`Hello !`" *est un objet de type* `String`

- Le nom d'un type primitif ou d'une variable débute par une **minuscule**.
Le nom d'une classe débute par une **Majuscule** !

- Un objet possède des **champs**. L'ensemble des valeurs de ses champs à un moment donné décrit **l'état** de l'objet : `x et y dans la classe Point.`

- La *construction* d'un objet passe par l'opérateur `new` suivi d'un **constructeur** de la classe de l'objet : `Point p = new Point(15,23);`

- Les **méthodes** d'une classe permettent d'envoyer un **message** à un objet de cette classe : `int x = p.getX();`

Field Summary

int	**x** The *x* coordinate.
int	**y** The *y* coordinate.

Constructor Summary

Point() Constructs and initializes a point at the origin (0, 0) of the coordinate space.
Point(int x, int y) Constructs and initializes a point at the specified (*x*, *y*) location in the coordinate space.
Point(Point p) Constructs and initializes a point with the same location as the specified Point object.

Method Summary

boolean	**equals**(Object obj) Determines whether or not two points are equal.
Point	**getLocation**() Returns the location of this point.
double	**getX**() Returns the X coordinate of the point in double precision.
double	**getY**() Returns the Y coordinate of the point in double precision.
void	**move**(int x, int y) Moves this point to the specified location in the (*x*, *y*) coordinate plane.

etc.

(dé)Construction de la classe Point

- Comment aurions-nous pu définir la classe `Point` si elle n'avait existé dans l'API Java ?

- Nous aurions déclaré qu'il s'agit d'une **classe** Java :

```
class Point {
    ....
    ....
}
```

← *le texte de la classe*

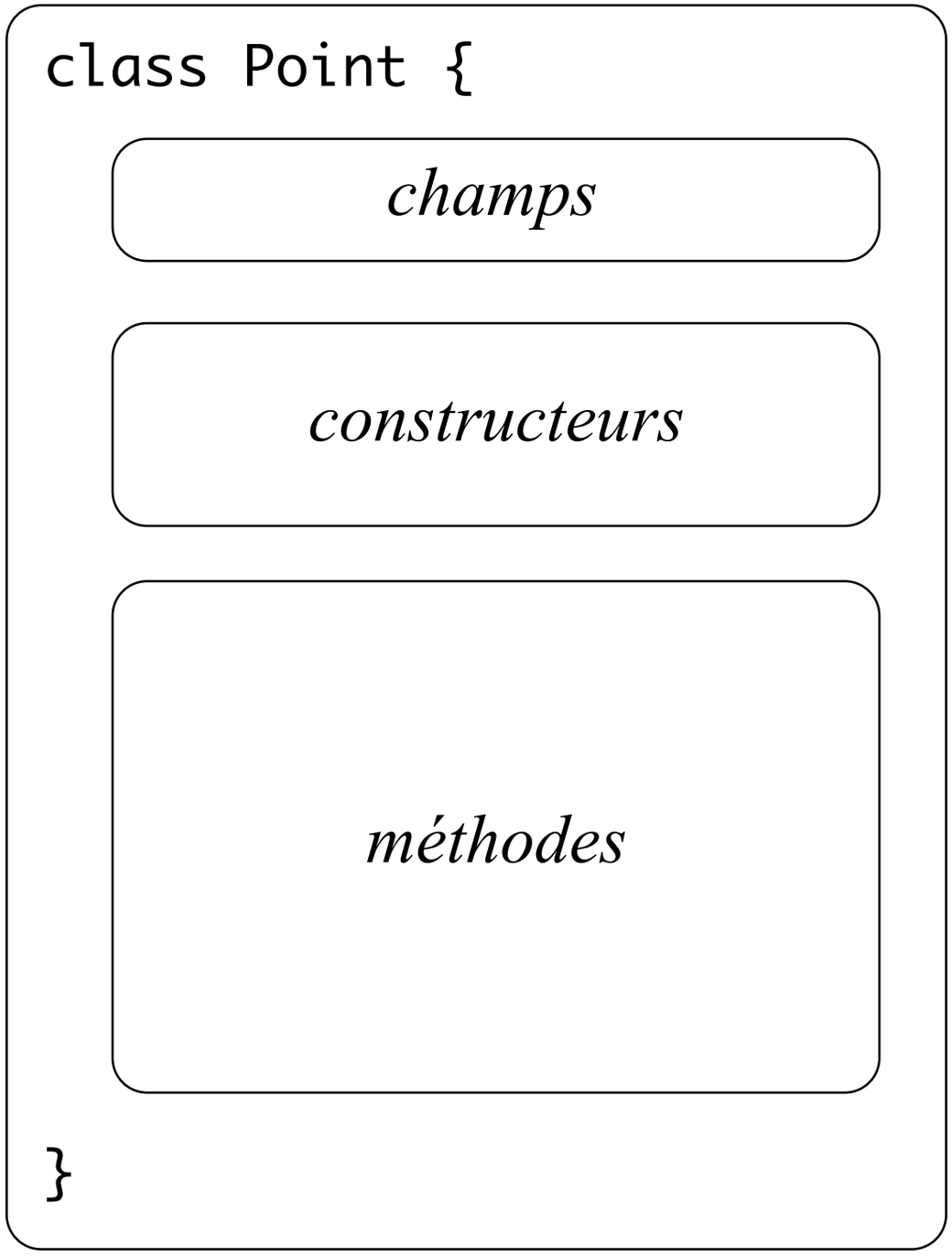
- Le texte de cette classe comprendra plusieurs sections, comme dans l'API :

① Les champs

② Les constructeurs

③ Les méthodes

- **Architecture d'une classe** :



- Retenez bien cet **ordre** dans lequel on définit les composants de la classe. Nous allons décrire les caractéristiques d'un objet de la classe.

① Les champs

• Je photographie un point à un instant donné. De quoi est-il composé ? Qu'est-ce qui résume son état ?

REPONSE : ses deux coordonnées x et y !

• Un objet point aura donc **deux champs** x et y. Java opte pour des entiers, dans le but de dessiner sur la grille des pixels de l'écran. De plus le graphisme entier est plus rapide que le graphisme flottant...

• Puisqu'il s'agit de simuler, nous faisons le même choix et déclarons donc deux champs entiers :

les champs ⟶

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

  <constructeurs>

  <méthodes>

}
```

② Les constructeurs

• Lorsque je vais construire un nouveau point, de quelle information vais-je disposer ?

- rien du tout : je construit le point *(0 ; 0)* par défaut avec un constructeur sans paramètre !

- les coordonnées *(a ; b)* du point à construire. Je les passe à un constructeur à deux paramètres.

- un point *p* déjà construit, dont je veux une copie (clone) : je passe ce point existant à un constructeur à un paramètre. Etc.

• Vous voyez qu'il peut y avoir plusieurs manières de construire un nouveau point, donc **plusieurs constructeurs**. C'est leur domaine de définition qui les distingue !

• **Un constructeur n'est pas une fonction (méthode)**, il n'y a pas de résultat, même *void* ! Il a le **même nom que celui de la classe** !

les champs ⟶

les 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;
  }

  <méthodes>
}
```

- On voit clairement que **le rôle d'un constructeur est d'initialiser les champs** de l'objet en construction.

- C'est new qui créé l'objet !...

③ Les méthodes

• Les **méthodes** constituent le savoir-faire d'un objet. Tous les objets d'une même classe ont les mêmes méthodes.

• Les méthodes permettent au programmeur de parler à un objet en lui envoyant un **message**.

• L'objet qui reçoit un message va **réagir** : effectuer un calcul, modifier son état, écrire sur l'écran, etc.

• Et **il retournera ou non un résultat** à celui qui a envoyé le message !

• Exemples de messages à envoyer à un point p :

p, donne-moi ton abscisse.
p, donne-moi ton ordonnée.
p, as-tu les mêmes coordonnées que le point q ?
p, déplace-toi d'un vecteur (dx,dy) !
p, donne-moi ton état sous la forme d'une chaîne.

- Traduction en Java :

p, donne-moi ton abscisse.	`p.x` *ou* `p.getX()`	*getX : void → int*
p, donne-moi ton ordonnée.	`p.y` *ou* `p.getY()`	*getY : void → int*
p, as-tu les mêmes coordonnées que le point q ?	`p.estEgalA(q)`	*estEgalA : Point → boolean*
p, déplace-toi d'un vecteur (dx,dy) !	`p.move(dx,dy);`	*move : int × int → void*
p, donne-moi ton état sous la forme d'une chaîne.	`p.toString()`	*toString : void → String*

- Programmation :

les champs ⟶

les constructeurs ⟶

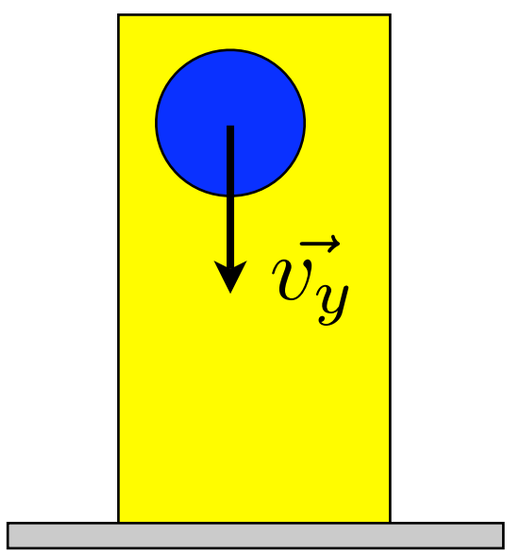
les méthodes ⟶

- *La méthode* `toString()` *résume l'état de l'objet dans une chaîne de caractères.*
Le mot ***public*** *est là car Java en a déjà une par défaut...*

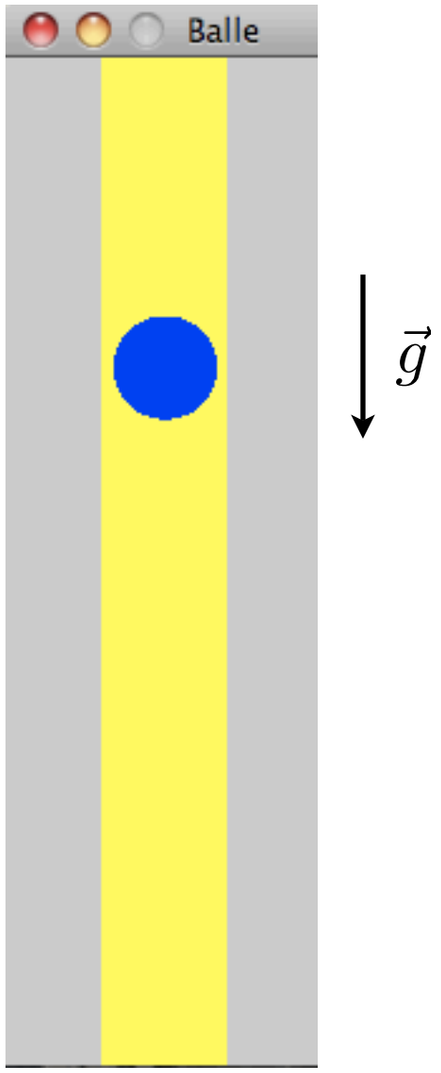
```
class Point {
  int x, y;
  <constructeurs>

  int getX() {
    return x;
  }
  int getY() {
    return y;
  }
  void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
  boolean estEgalA(Point q) {
    return x == q.x && y == q.y;
  }
  public String toString() {
    return "Point[x=" + x + ",y=" + y + "]";
  }
}
```

Application

Modélisation d'une balle dans un champ de pesanteur



• Une balle de masse m = 1 est lachée à la verticale. On veut modéliser son mouvement sous l'action de la gravité.



• La **balle** sera un **objet avec 3 champs** entiers :

- son ***rayon*** r (pour le dessin).
- sa ***position verticale*** y.
- sa ***vitesse verticale*** vy.

constructeur →

```
class Balle {
  int r, y, vy;

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

  <méthodes>
}
```

- **Méthodes** de la classe `Balle` : que doit savoir faire une balle ?

  - Elle doit savoir s'afficher graphiquement dans le canvas.

$$afficheToi : void \rightarrow void$$

  - Elle doit savoir représenter textuellement son état courant.

$$toString : void \rightarrow String$$

  - Elle doit savoir faire un pas élémentaire dans l'unité de temps.

$$bouge : void \rightarrow void$$

- La méthode `toString` est usuelle, comme dans la plupart des classes. On s'en sert souvent lors de la mise au point pour connaître l'état de la balle lors d'un mauvais fonctionnement : `println(balle)`.

- La méthode `dessineToi` n'a pas de résultat : elle va modifier l'écran...

- Programmation :

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

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

  void dessineToi() {
    ellipse(width / 2, y, 2 * r, 2 * r);
  }

  public String toString() {
    return "Balle[r=" + r + ",y=" + y + ",vy=" + vy + "]";
  }

  <méthode bouge>
}
```

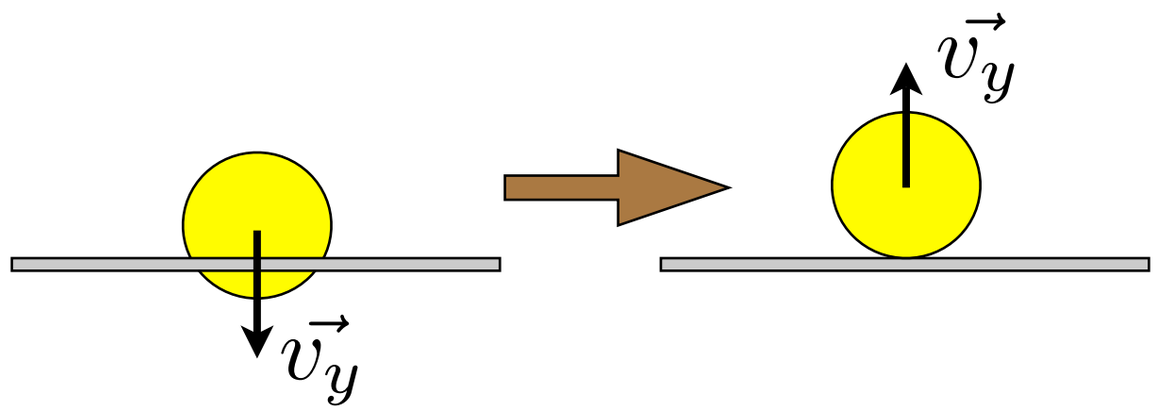
j'utilise Processing...

N.B. *Lorsqu'une classe Java n'utilise pas les primitives Processing, on dit que c'est une classe en* ***Java pur****... Ce n'est pas le cas de la classe* `Balle` *!*

• A chaque unité de temps (changement d'image), la balle va faire un pas de longueur sa vitesse courante. Suite à l'effet de la pesanteur, sa vitesse est ensuite augmentée d'une quantité constante (on prend g = 1).

```
void bouge() {
  y = y + vy;          // mise à jour de la position
  vy = vy + 1;         // mise à jour de la vitesse
}
```

• Problème : la balle tombe bien en accélérant, dépasse le bas de la fenêtre et s'enfonce dans le sol. Il faut **gérer le rebond** !



• Or il y a rebond lorsque :

$y + r \geq height$

On laisse venir le rebond, et on force alors la balle à revenir au niveau du sol, en inversant la direction de sa vitesse.

```
void bouge() {
    y = y + vy;
    if (y + r > height) {
      y = height - r;
      vy = -vy;
    }
    vy = vy + 1;        // gravité = 1
  }
```

Mais le mouvement est perpétuel ?!

- Notre modélisation est incomplète, car le mouvement est... perpétuel !

- Dans la réalité, la balle rebondit de moins en moins haut, sous l'effet des frottements (air, contact mou avec le sol).

- Rajoutons donc une composante de **friction à chaque rebond**. La friction est simplement un coefficient multiplicatif $f \in [0,1]$ que l'on applique à la vitesse. Pour $f = 0$, on a un choc ***mou*** (chewing-gum envoyé sur un mur) et pour $f = 1$, le choc est parfaitement ***élastique*** (c'était le cas jusqu'à présent).

- La friction étant un **nombre approché**, la vitesse devient un nombre approché, et la position aussi !

- Nous sommes conduits à revoir le texte de la classe...

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

  Balle(int rInit, float fInit, float yInit, float vyInit) {
    r = rInit;
    friction = fInit;
    y = yInit;
    vy = vyInit;
  }

  void bouge() {
    y = y + vy;
    if (y + r > height) {         // collision avec le sol ?
      y = height - r;
      vy = -vy;
      vy = vy * friction;         // coefficient de friction
    }
    vy = vy + 1;                  // gravité = 1
  }

  void dessineToi() { ... }

  public String toString() { ... }

}
```

```
Balle b = new Balle(20, 0.9, 40, 0);
```

- Le **programme principal** qui va utiliser la classe `Balle` comprendra les méthodes `setup()` et `draw()` :

```
Balle b;                  //  b est utilisée dans les deux fonctions

void setup() {
  size(50,400);
  fill(0,0,255);     //  couleur de remplissage de la balle
  noStroke();        //  on ne dessine pas le contour de la balle
  b = new Balle(20, 0.9, 40, 0);
  println("Balle initiale : " + b);
}
```

```
void draw() {
  background(255,255,0);      //  on efface le fond d'écran
  b.dessineToi();             //  on dessine la balle courante
  b.bouge();                  //  et on fait avancer la balle d'un pas
}
```

- Au final, ce programme Processing comprendra deux fichiers qui seront placés dans un dossier (les programmeurs parlent d'un **projet**).

Modéliser avec des objets

• Une **modélisation à objets** va décrire les objets d'un domaine, leurs propriétés (champs, méthodes) et éventuellement comment ils sont reliés entre eux.

• On doit penser aux objets comme ayant une **identité propre**. Ceci signifie par exemple que deux objets avec des champs égaux restent cependant deux objets distincts.

• Les grands concepts de la **programmation orientée objets** sont déjà présents dans le langage SIMULA (1967). Puis SMALLTALK (1972) popularisa ces idées dans les laboratoires de recherche Xerox, où les ingénieurs d'Apple puisèrent l'interface graphique du Macintosh.

• Les langages Java et C++ se disputent la première place des langages à objets. Mais **quasiment tous les langages modernes ont des objets** !