Introduction à la programmation avec Java

UFR Sciences de Nice

Licence Math-Info 2006-2007

Module L1I1

Frédéric MALLET

Jean-Paul ROY

Où en sommes-nous ?

✔ Nous savons rédiger le texte d'une **classe d'objets**, avec dans l'ordre : ses *champs*, ses *constructeurs*, ses *méthodes*.

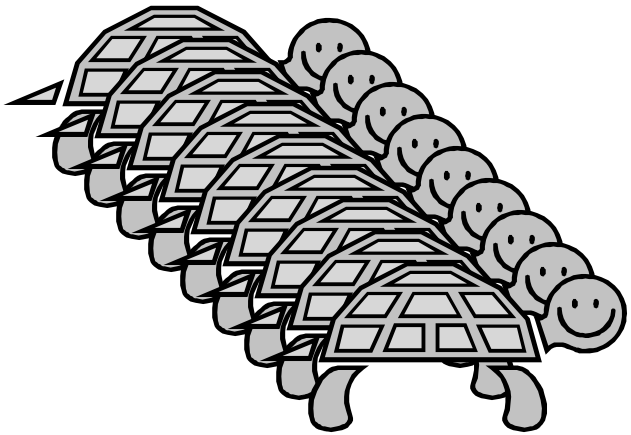
✔ Nous pouvons exprimer qu'une méthode a ou n'a pas de *résultat*.

✔ L'utilisation d'une *conditionnelle* **if** n'a plus de secrets pour nous.

✔ Nous commençons à acquérir des capacités d'*abstraction* et de *modularité*.

✔ Nous savons faire la différence entre une *méthode d'instance* et une *méthode de classe* (*statique*).

Les algorithmes itératifs



Qu'est-ce qu'un calcul « répétitif » ?

- Comment calculer la factorielle **`n!`** d'un entier **`n ≥ 0`** ?

SOLUTION 1 : par une récurrence (comme en maths)

$$n! = \begin{cases} 1 & \text{si } n=0 \\ n*(n-1)! & \text{sinon} \end{cases}$$

`n! = si n==0 alors 1 sinon n*(n-1)!`

```
static int fact(int n) {
  if (n == 0)
    return 1;
  return n*fact(n-1);
}
```

fact est une fonction récursive

Stratégie très intéressante que nous développerons au cours 9…

SOLUTION 2 : par une itération

- Comme à la main : **`1 * 2 * 3 * 4 * 5 * … * n`**
- Soit **`res`** le résultat entier du calcul, initialisé à **`1`** (produit vide).
- Pour tout **`k`** variant depuis **`1`** jusqu'à **`n`**, faire **`res = res * k`**

```
public static int factFor(int n) {
  int res = 1;
  for(int k=1; k<=n; k=k+1) {
    res = res * k;
    System.out.println("k=" + k + ",res=" + res + "\n");
  }
  return res;      // ici k n'est plus visible !
}
```

k	res
1	1
2	2
3	6
4	24
5	120

La boucle **`for`** contrôle automatiquement l'évolution de sa variable **`k`**. La variable **`res`** est modifiée par une instruction spécifique.

- La syntaxe (forme grammaticale) de cette boucle **`for`** est :

```
for( initialisation de la variable de boucle ; condition de continuation ; évolution de la variable de boucle ) {

    instruction;
    ...
    instruction;

}
```

- Autre exemple : calcul de la somme $1^2 + 3^2 + 5^2 + \ldots + 99^2$

```
int res = 0;
for(int k=1 ; k < 100 ; k = k+2) {
  res = res + k * k;
}
System.out.println("La somme vaut " + res);
```

• Le nombre de « tours de boucle » n'est pas forcément connu à l'avance !

• Soit à calculer le plus petit diviseur **`d ≥ 2`** d'un entier **`n ≥ 2`**.

· Rappel : d divise n ⇔ **`n % d == 0`**

· Idée : je pars de **`d = 2`**, et je monte tant que **`d`** ne divise pas **`n`**.

Ai-je forcément une solution ? Avec les nombres premiers ?

```
public static int ppdiv(int n) {
  assert(n>=2);
  int d;
  for(d=2 ; n % d != 0; d = d + 1) {  }
  return d;
}
```

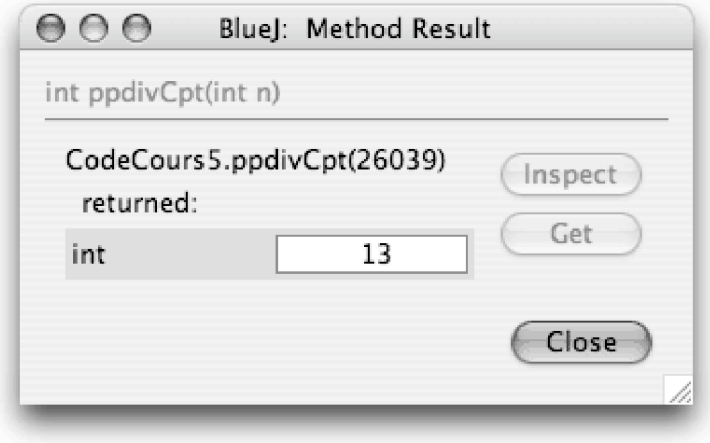
Le corps de boucle est vide ! Il n'y a que **`d`** qui monte…

N.B. i) Le corps de boucle est optionnel.

ii) La variable **`d`** est déclarée en-dehors de la boucle **`for`** car on en a besoin après la fin de la boucle !

• Comment faire afficher ce nombre de « tours de boucle » si l'on en a besoin ? Facile : il suffit de prendre un COMPTEUR **`cpt`** :

```
public static int ppdivCpt(int n) {
  assert(n>=2);
  int d, cpt = 0;
  for(d=2 ; n % d != 0; d = d + 1) {
    cpt++;
  }
  System.out.println("Pour n=" + n + ", cpt=" + cpt);
  return d;
}
```



• Petit problème au passage, lié à paranoïa sécuritaire de Java…
Le code suivant ne compilerait pas :

```
public static int ppdivBug(int n) {
  assert(n>=2);
  for(int d=2 ; d <= n ; d = d + 1) {
    if (n % d == 0) {
      return d;
    }
  }
}
```

Missing return statement !

EXPLICATION :

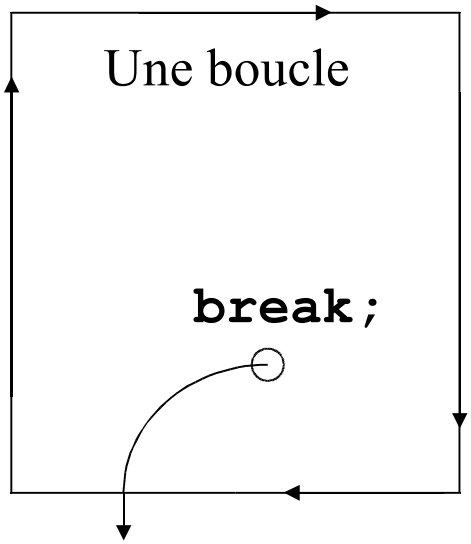
- Je sais que la boucle se terminera et que le **`return`** finira par avoir lieu ! En effet, dans le pire des cas, **`d`** sera égal à **`n`**.

- Mais le compilateur Java n'en est pas sûr. Il veut avoir la certitude que toute sortie de fonction se fait sur un **`return`** !
Le **`if`** le laisse ici dans un état perplexe, et il râle…

- Il faut effectuer le **`return`** *après* la boucle !

- Donc sortir de la boucle (la « casser ») dès que l'on rencontre un diviseur de **`n`**. C'est l'instruction **`break`**…

L'exécution de l'instruction **`break`** dans une boucle provoque immédiatement la sortie de la boucle. Les instructions qui suivent la boucle seront alors exécutées…




```
public static int ppdivBreak(int n) {
  int d;
  for(d=2 ; d <= n ; d = d + 1) {
    if (n % d == 0) {
      break;    // je continue sur la ligne du return !
    }
  }
  return d;
}
```

La boucle `while` (tant que…)

- C'est une boucle traditionnelle en algorithmique.

FONCTION ppdiv(entier n) : entier
On suppose que $n \geq 2$
Soit d = 2
Tant que d ne divise pas n :
    d devient d+1
Le résultat est d.

```
public static int ppdivWhile(int n) {
  assert(n>=2);
  int d = 2;
  while (n % d != 0) {
    d = d+1;
  }
  return d;
}
```

Il est essentiel de s'assurer que la boucle termine dans tous les cas !

• Choisir entre les fonctions des pages 7, 10 et 11 est affaire de goût et de classicisme…

• En général, la boucle **`for`** a une meilleure sécurité que la boucle **`while`**… sauf si on l'utilise comme une boucle **`while`**, sans garantir que l'on puisse majorer le nombre de tours de boucle !

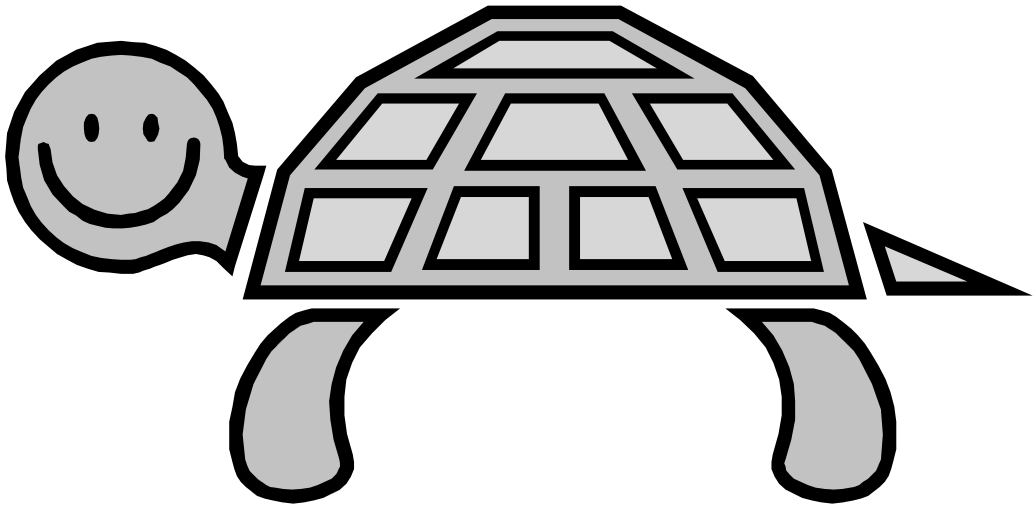
• Le problème cité aux pages 9 et 10 se transpose ipso facto à la boucle **`while`**, dont on peut sortir par un **`break`**.

• Nous y reviendrons régulièrement, puisque vous allez pratiquer les boucles pendant plusieurs années !

• L'essentiel de la stratégie lorsqu'on veut programmer une boucle tient dans ces mots :

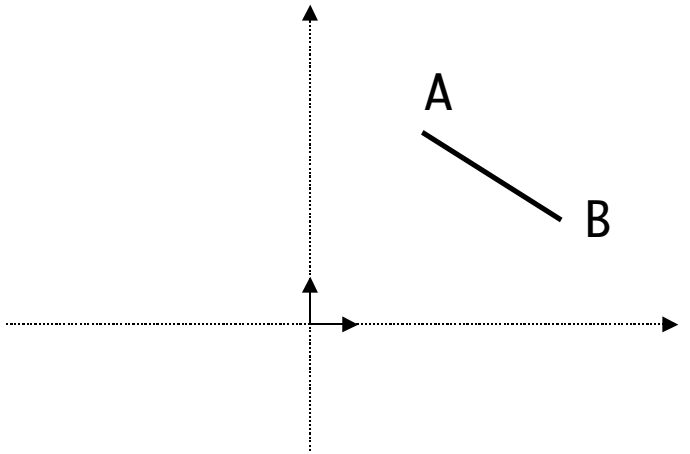
> Si je suis en plein milieu du calcul, de quelles variables dois-je disposer pour continuer, et que représentent-elles exactement ?…

Le Graphisme de la Tortue



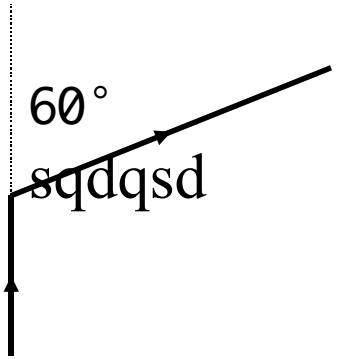
Il y a deux sortes de graphismes

• Le graphisme cartésien, dans un repère (orthonormé). Un point est repéré par ses coordonnées globales. On sait dessiner un point A(x,y), et un segment AB joignant deux points.




```
traceSegment(A,B);
```

• Le graphisme polaire, sans repère global. On sait seulement se diriger vers l'avant ou l'arrière, et tourner à gauche ou à droite. C'est un graphisme local.



translations + rotations :
déplacements plans

```
avance(50);
tourneDroite(60°);
avance(100);
```

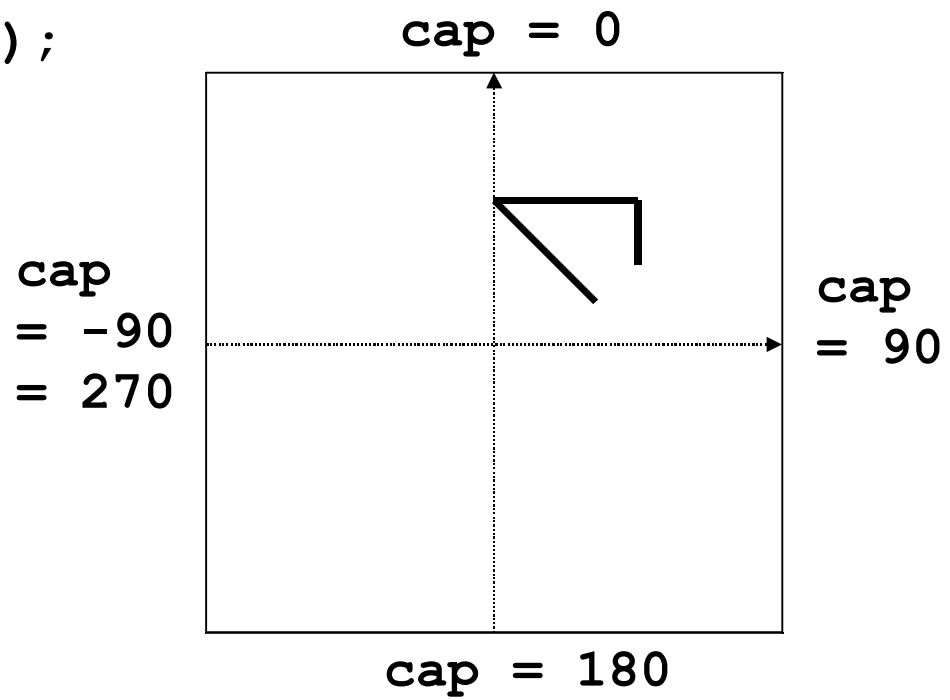
Le graphisme polaire de la tortue

• Vous devez intégrer **`turtle.jar`** et avoir **`Turtle.java`** dans votre répertoire. Ces deux fichiers vous sont fournis sur le Web.

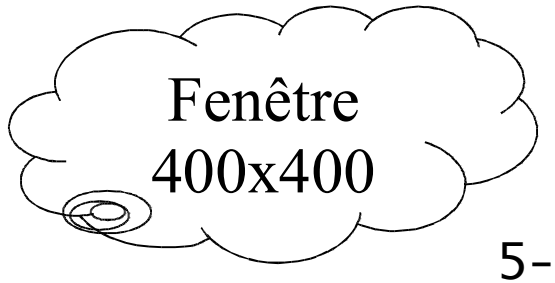
```
public class Turtle {
  public Turtle(double x, double y) {...}
  public Turtle() { this(0,0); }
  public void forward(double distance) {...}
  public void back(double distance) {...}
  public void right(double angle) {...}
  public void left(double angle) {...}
  public void penUp() {...}
  public void penDown() {...}
  public void clear() {...}
  public void setPenColor(String color) {...}
}
```

• Le constructeur **`Turtle(x,y)`** initialise une nouvelle tortue au point (x,y), regardant vers le Nord (cap = 0°), crayon baissé.

```
Turtle t = new Turtle(100,50);
t.forward(50);
t.left(90); // degrés !
t.setPenColor("red");
t.forward(100);
t.penUp();
t.left(135);
t.forward(100);
t.penDown();
t.setPenColor("blue");
t.forward(100);
```



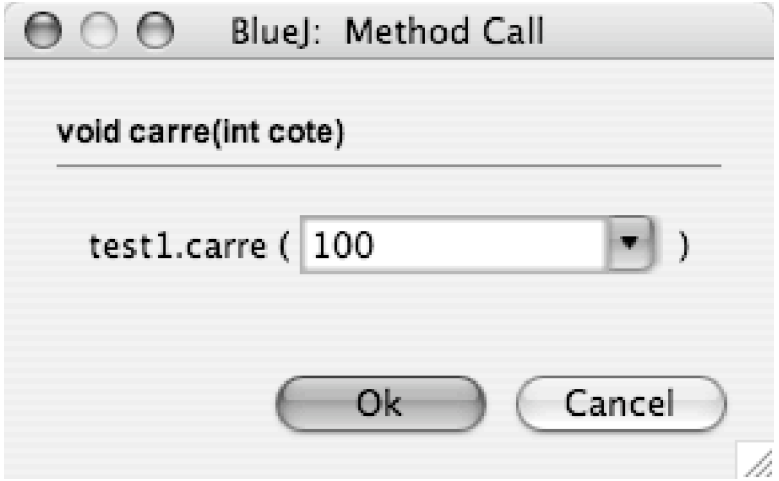
"blue", "red", "green", "black"

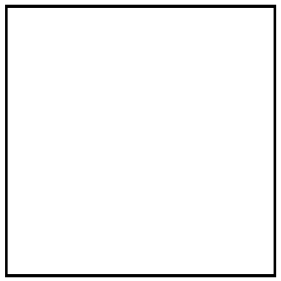


• Dans une classe **`TestTurtle`**, écrivons des « méthodes tortue ».

Un carré

```
class Test {

   private Turtle lea;

   /**
    * Dessin d'un carré
    * @param c longueur du côté
    */
   void carre(int c) {
     lea.forward(c);
     lea.right(90);
     lea.forward(c);
     lea.right(90);
     lea.forward(c);
     lea.right(90);
     lea.forward(c);
     lea.right(90);
   }
}
```





• Oui, bien sûr, mieux vaut écrire une *boucle* :

```
public void carre(int c) {
  for(int k=1 ; k<=4 ; k++) {
    lea.forward(c);
    lea.right(90);
  }
}
```

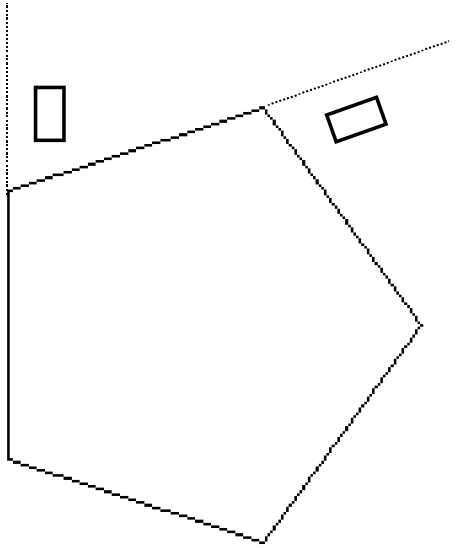
J'exécute 4 fois la même chose !

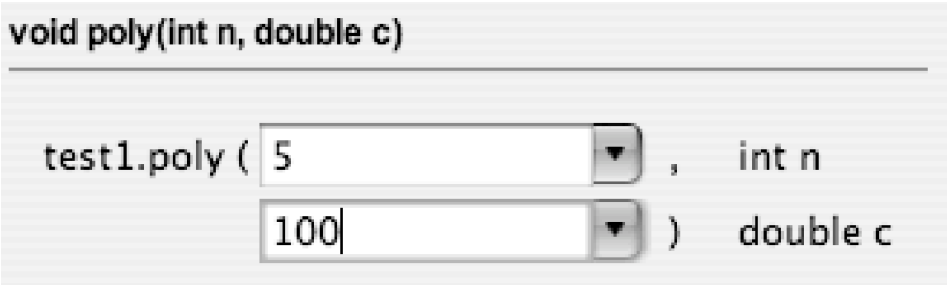
```
public void carre(int c) {
  int k = 1;
  while (k <= 4) {
    lea.forward(c);
    lea.right(90);
    k++;
  }
}
```

Un polygone régulier à n côtés

```
/**
  * Dessin d'un polygone régulier
  * @param n le nombre de côtés
  * @param c la longueur d'un côté
  */
```

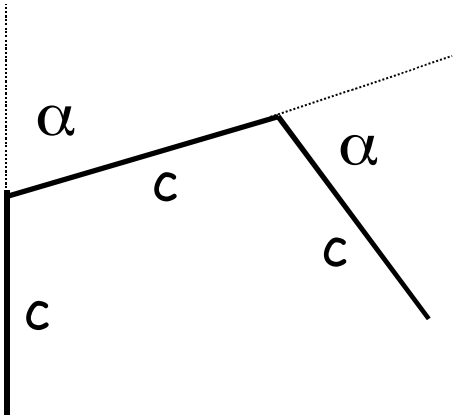
```
public void poly(int n, int c) {
  assert(n>2);
  double □ = 360.0/n;
  for(int i=0; i< n; i++) {
    lea.forward(c);
    lea.right(□);
  }
}
```





Une autre vision d'un polygone

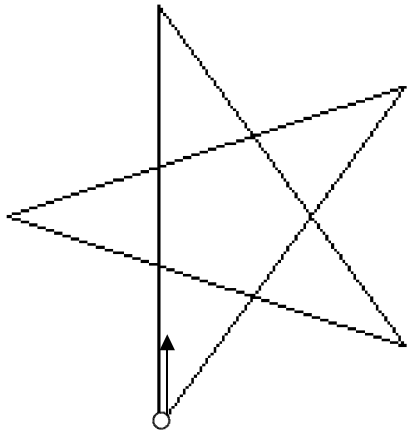
- Un polygone est finalement donné par :
  - la longueur c d'un côté
  - l'angle α dont on tourne à chaque sommet.

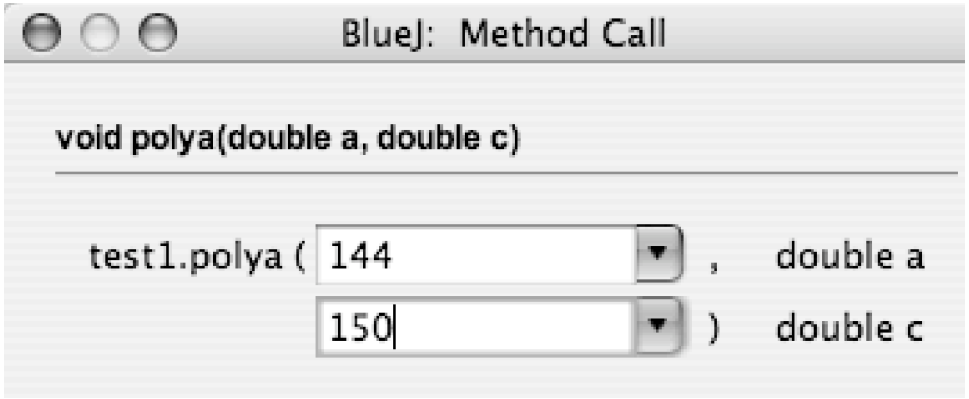



```
public void polya(int □, int c)
{
   while(true) {
     lea.forward(c);
     lea.right(□);
   }
}
```

- Hélas cette méthode ne termine pas. Il faut forcer la « BlueJ Virtual Machine » à quitter (par ex. en cliquant sur la case de fermeture).

• Le polygone obtenu n'est pas forcément convexe. Il peut être « croisé », par exemple si **`a = 144°`** :





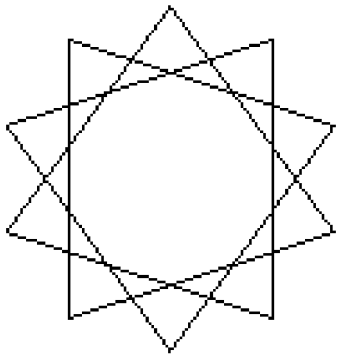
• On obtient ici un polygone croisé à **`5`** côtés qui continue à être tracé alors que la tortue est repassée par le point de départ ! Comment la stopper ?…

• Lorsqu'elle repasse par son point de départ, la tortue a tourné **`5`** fois d'un angle **`a = 144°`**, donc d'un angle total **`at = 720°`** qui est un multiple de **`360`**.

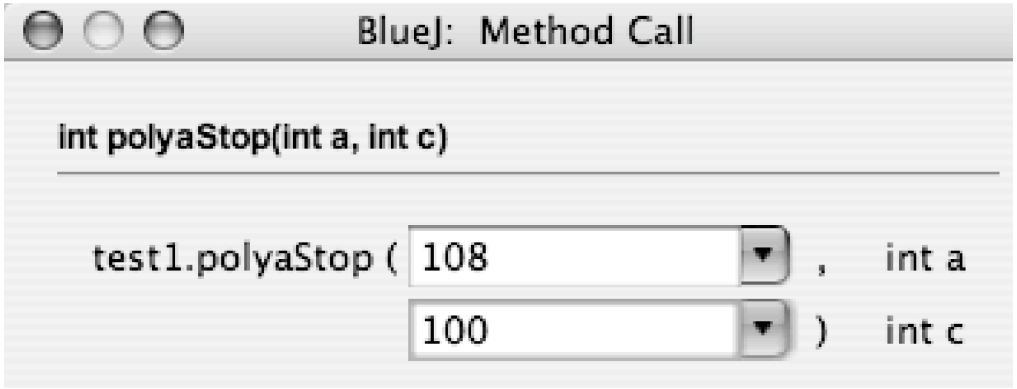
• Et de manière générale :

`at` = (nombre de côtés) × **`a`** = (un entier) × **`360`**

• Ceci donne un moyen de la stopper : dès que l'angle total dont elle a tourné devient un multiple de **360**.



```
public int polyaStop(int a, int c) {
  lea.forward(c); lea.left(a);
  int at = a, n = 1;
  while (at % 360 != 0) {
    lea.forward(c);
    n++;
    lea.left(a);
    at += a;
  }
  return n;
}
```



La boucle « **do**...**while**... »

• Le programme précédent n'est pas très élégant, il duplique la séquence d'instructions :

`lea.forward(c); lea.left(a);`

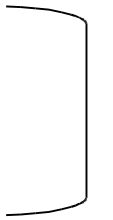
une fois avant la boucle, et une autre fois dans la boucle !

• EXPLICATION : la boucle **`while`** teste avant d'agir !

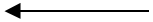
• La boucle **`do`**...**`while`**... agit puis teste ensuite s'il faut continuer.

On est donc sûr que l'action sera exécutée au moins une fois !

```
public int polyaStopDoWhile(int a, int c) {
  int at = 0, n = 0;                    // ce qui est plus naturel !
  do {
       lea.forward(c);
       n++;
       lea.left(a);
       at += a;
  } while (at % 360 != 0);
  return n;
}
```



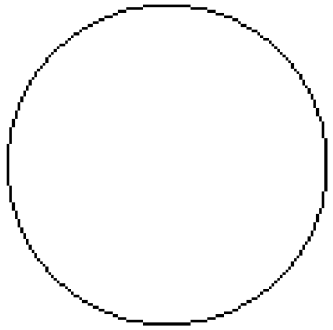
action



test de sortie

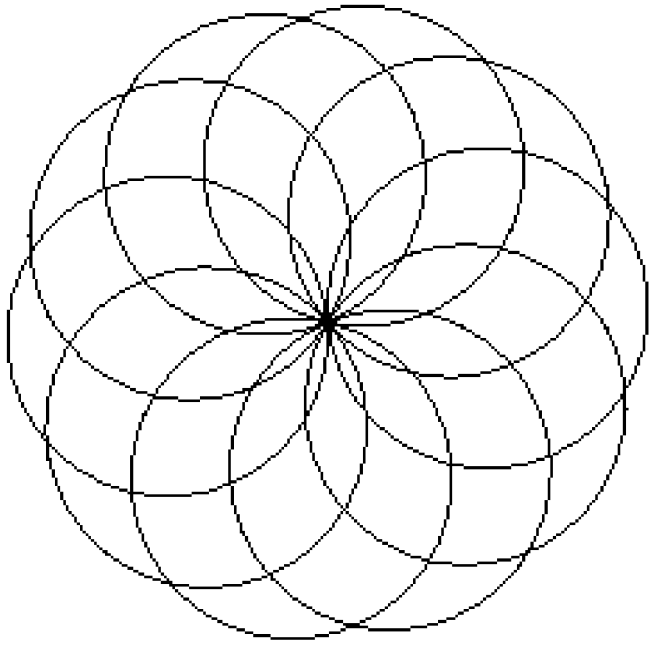
Les cercles

• On approche un cercle par un polygone régulier convexe ayant beaucoup de sommets. Bons résultats à partir de 36 sommets :



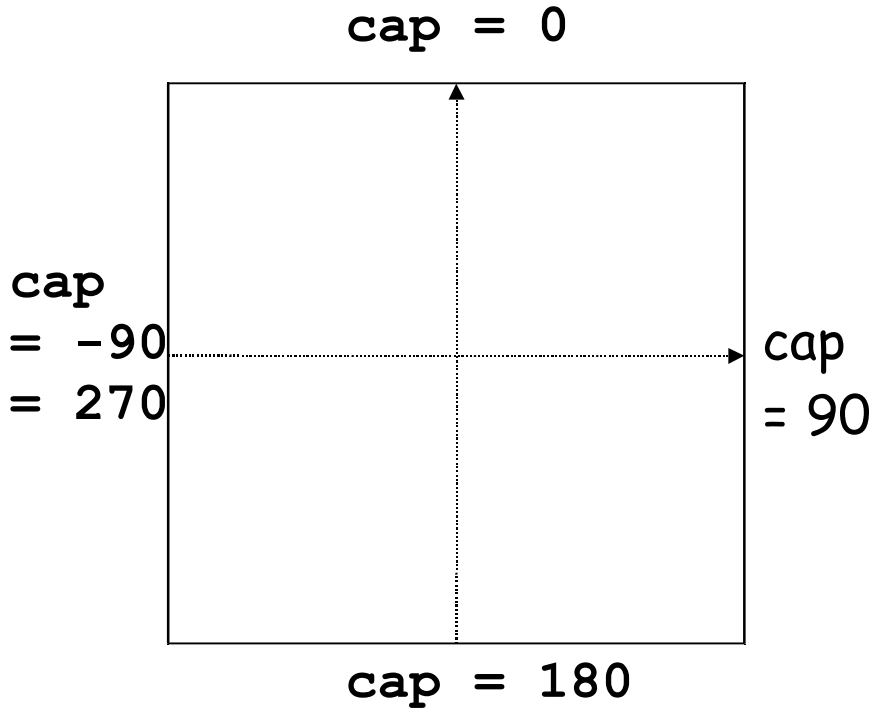
• On peut bouger la tortue entre deux tracés :

```
for(int i=0; i<10; i++) {
   this.poly(36,10);
   lea.right(36);
}
```



Le graphisme cartésien de la tortue

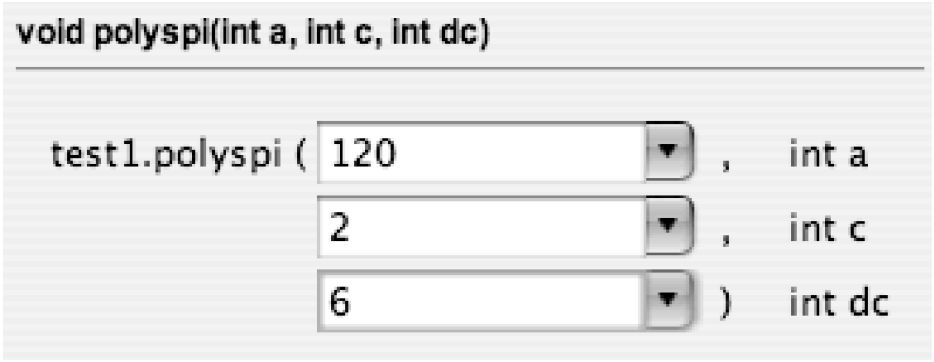
• Le fichier **`Turtle.java`** donne aussi accès à quelques primitives liées à un repère absolu Oxy, origine au centre de la fenêtre et axes usuels.

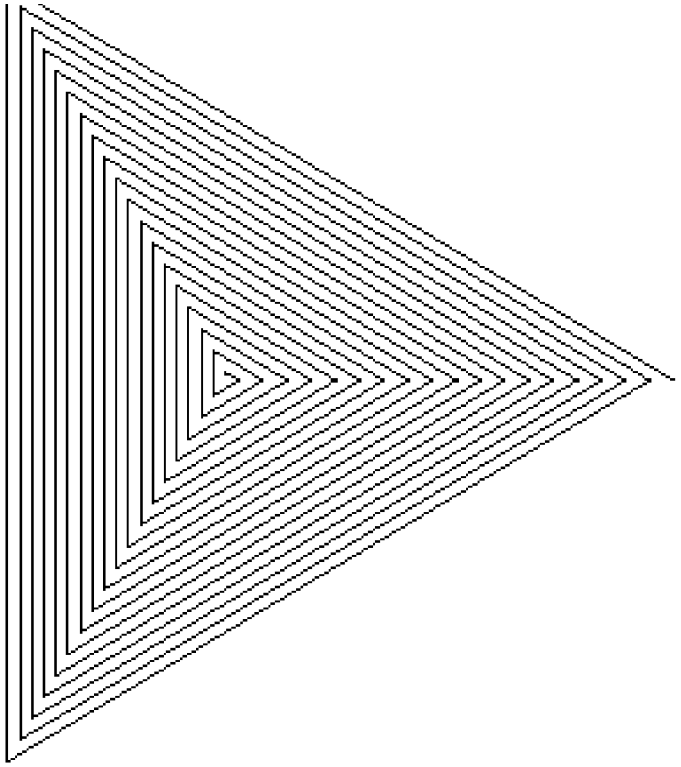



```
public double getX()
public double getY()
public double getHeading()

public void setPosition(double x, double y)
public void setHeading(double cap)

public void toward(double x, double y)
```

Les spirales

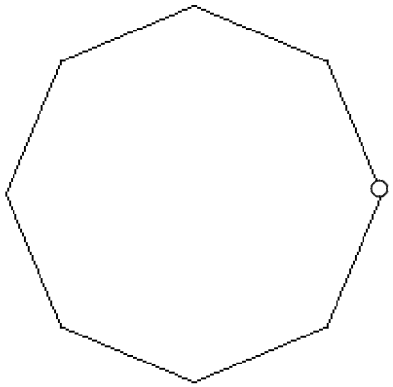
- Pourquoi avancer d'une longueur constante ?



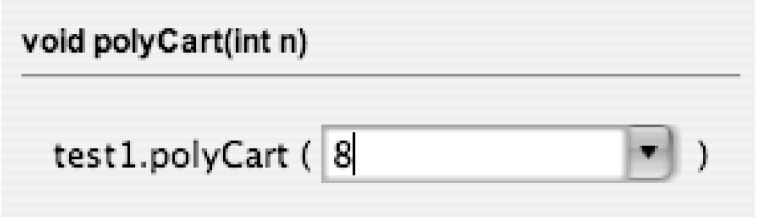


```
public void polyspi(int a, int c, int dc) {
   while (Math.abs(lea.getX()) < 190
            && Math.abs(lea.getY()) < 190) {
      lea.forward(c);
      lea.right(a);
      c = c + dc;
   }
}
```

Un polygone régulier à n côtés



- En cartésien, il faut connaître les coordonnées de chaque sommet. Inscrivons-le dans un cercle de centre `O` et de rayon `100`. Les sommets sont liés aux racines n-èmes complexes de `1`.

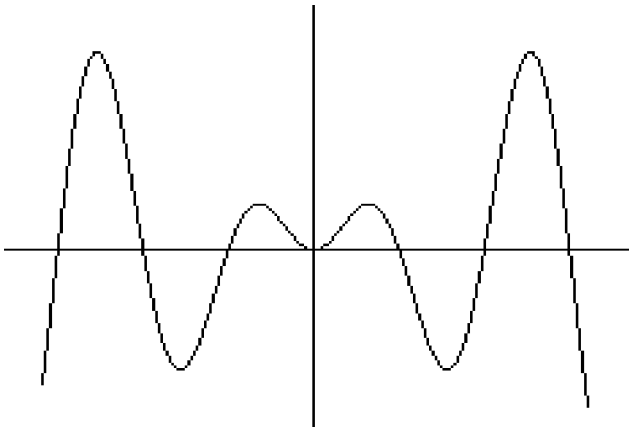


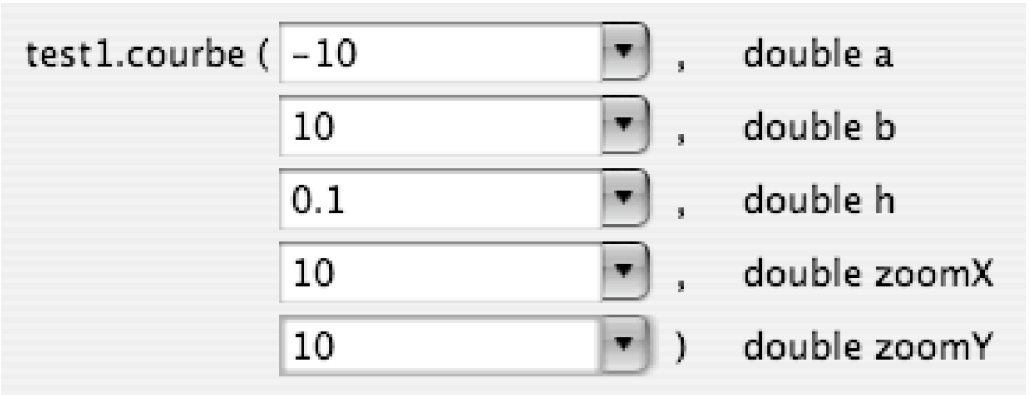

```
public void polyCart(int n) {
  assert(n>2);
  double a = 2 * Math.PI / n;
  lea.penUp(); lea.setPosition(100,0); lea.penDown();
  for(int i=1; i<=n; i++) {
    lea.setPosition(100 * Math.cos(i * a),
                    100 * Math.sin(i * a));
  }
}
```

Comment tracer une courbe y = f(x) simple ?

- Simple : continue sur [a,b], sans variations trop brutales.

```
public static double f(double x) {
   return x * Math.sin(x);
}
```





```
public void courbe(double a, double b, double h,
                   double zoomX, double zoomY) {
  for(int i=0; i< 4; i++) {      // les deux axes
    lea.forward(200); lea.back(200); lea.right(90);
  }
  double x = a, y = f(a);        // le point de départ
  lea.penUp();
  lea.setPosition(zoomX * x, zoomY * y);
  lea.penDown();
  while (x < b) {
    x += h;
    y = f(x);
    lea.setPosition(zoomX * x, zoomY * y);
  }
}
```

- Tracer une courbe consiste donc à dessiner une ligne brisée (un polygone ouvert).
- Les paramètres **`zoomX`** et **`zoomY`** gouvernent les échelles.

Simulation d'une « promenade aléatoire »

- On se propose de simuler la trajectoire suivie par un homme ivre faisant **n** pas de longueur **1** mais changeant aléatoirement de direction à chaque pas, et calculer le carré $\mathbf{d}^2$ de la distance finale **d** à l'origine.

```
public double promenade(int n) {  // promenade de n pas
  double x0 = lea.getX(), y0 = lea.getY();
  for(int i=0; i<n; i++) {
     lea.forward(1);
     lea.left(rand.nextInt(360));
  }
  double dx = lea.getX() - x0, dy = lea.getY() - y0;
  return dx * dx + dy * dy;
}
```

Comment faire un diaporama ?

• Pour faire afficher plusieurs programmes graphiques avec un laps de temps entre deux, écrivons une méthode **`delay(…)`** :

```
/** delay(n) ne fait rien pendant n millisecondes */
public static void delay(int n) {
   long t0 = System.currentTimeMillis();
   while (System.currentTimeMillis() - t0 < n) {}
}
```

• Et une méthode se chargera de faire défiler les images :

```
public void diaporama() {
  lea.clear();                          // efface la fenêtre
  this.courbe(-10,10,0.1,10,10);        // affiche un dessin
  delay(2000);                          // attend 2 secondes
  lea.clear();                          // efface la fenêtre
  this.promenade(1000);                 // affiche un autre dessin
}                                       // etc.
```

L'origine du graphisme tortue : LOGO

- Le graphisme de la tortue est apparu avec le langage LOGO vers 1968, au MIT A.I. Lab (*Artificial Intelligence Laboratory, Massachusets Institute of Technology*), dans une équipe dirigée par Seymour Papert.
- Destiné aux enfants pour en étudier les mécanismes d'apprentissage et la spatialisation.
- Apprécié des instituteurs pour sa facilité d'utilisation.

```
POUR CARRE :C
REPETE 4 [AVANCE :C DROITE 90]
FIN
```

Comparez avec le programme Java page 18…