Introduction à la programmation (Java)

(D’après des slides de Pierre Poulin); A ne pas distribuer

Attention:
un ; après le for( ), itère sur la condition, et ‘somme’ ne sera incrémentée qu’une seule fois

Boucle

Attention:
‘i’ n’est déclarée ici qu’à l’intérieur de la boucle for

- Pour traiter beaucoup de données en série
- Schémas
  - Boucle *for*

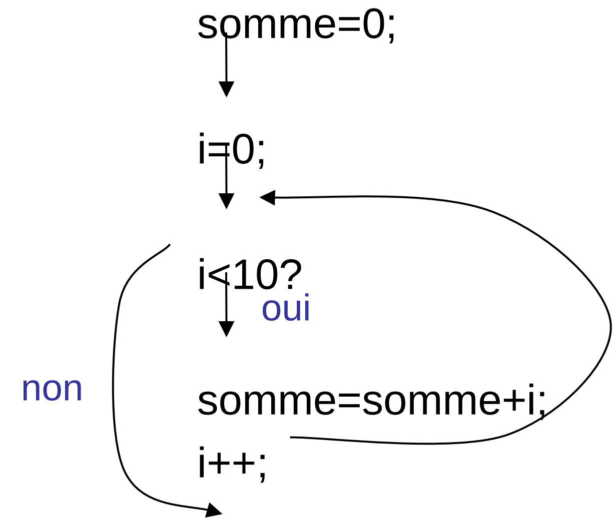
```
int somme = 0;
for (int i = 0; i<10; i++) somme = somme + i;
```

  - Boucle *while*

```
int somme = 0;
int i = 0;
while (i<10) { somme = somme + i;
               i++;
             }
```

- Que font ces deux boucles?

Schéma d’exécution



i:		0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10
somme:	0	0,	1,	3,	6,	10,	15,	21,	28,	36,	45,	sortie

Boucle

- do *A* while (*condition*)
  - Faire *A* au moins une fois
  - Tester la condition pour savoir s'il faut refaire *A*

```
int somme = 0;
int i = 15;
while (i<10) { somme = somme + i;
    i++;
   }
```

somme = 0

```
int somme = 0;
int i = 15;
do { somme = somme + i;
     i++;
     }
while (i<10)
```

somme = 15

Schéma d'exécution

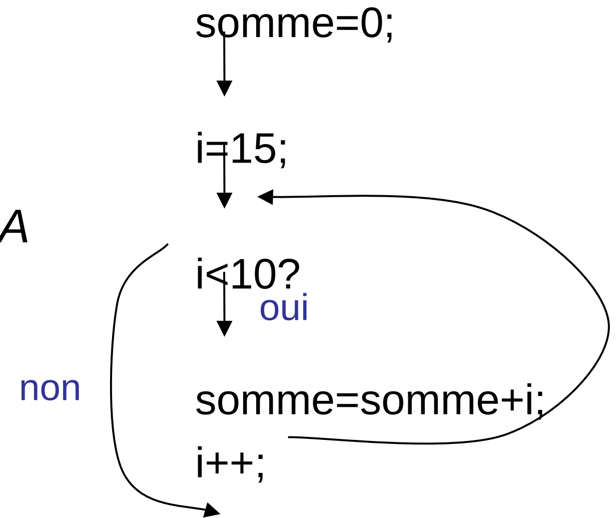
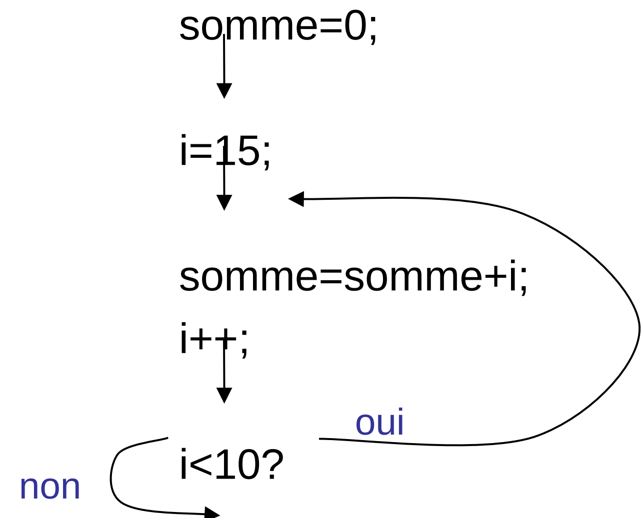


Schéma d'exécution



Exemple

- Calcul des intérêts
- Étant donné le solde initial, le solde souhaité et le taux d'intérêt, combien d'années seront nécessaires pour atteindre le solde souhaité
  - au lieu d'utiliser une formule, on simule le calcul
- Algorithme (pseudocode):
  1. ans = 0;
  2. WHILE solde n'atteint pas le solde souhaité
  3. incrémenter ans
  4. ajouter l'intérêt au solde

Programme

```
public void nombreAnnees (double balance, double targetBalance,
    double rate ) {
    int years = 0;
    while (balance < targetBalance) {
     years++;
     double interest = balance * rate;
     balance += interest;
    }
    System.out.println(years + " years are needed");
}
```

years = years + 1;

balance = balance + interest;

Appel de la méthode:

nombreAnnees(1000, 1500, 0.05)

Résultat:

56 years are needed

Factorielle

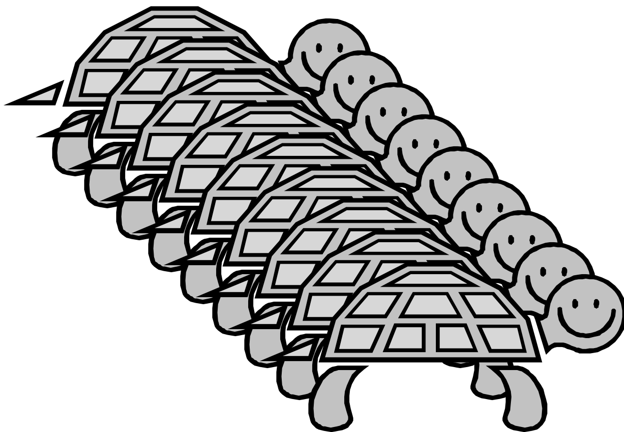
```
public class Factorielle
{
    public static double factorielle(int x) {
     if (x < 0) return 0.0;
     double fact = 1.0;
     while (x > 1) {
         fact = fact * x;
         x = x - 1;
     }
     return fact;
    }

    public static void main(String[] args) {
     int entree = Integer.parseInt(args[0]);
     double resultat = factorielle(entree);
     System.out.println(resultat);
    }
}
```

Si une méthode (ou un attribut, une variable) de la classe est utilisée par la méthode main (*static*), il faut qu'il soit aussi *static.*

Testons

Puis reprenons tranquillement

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

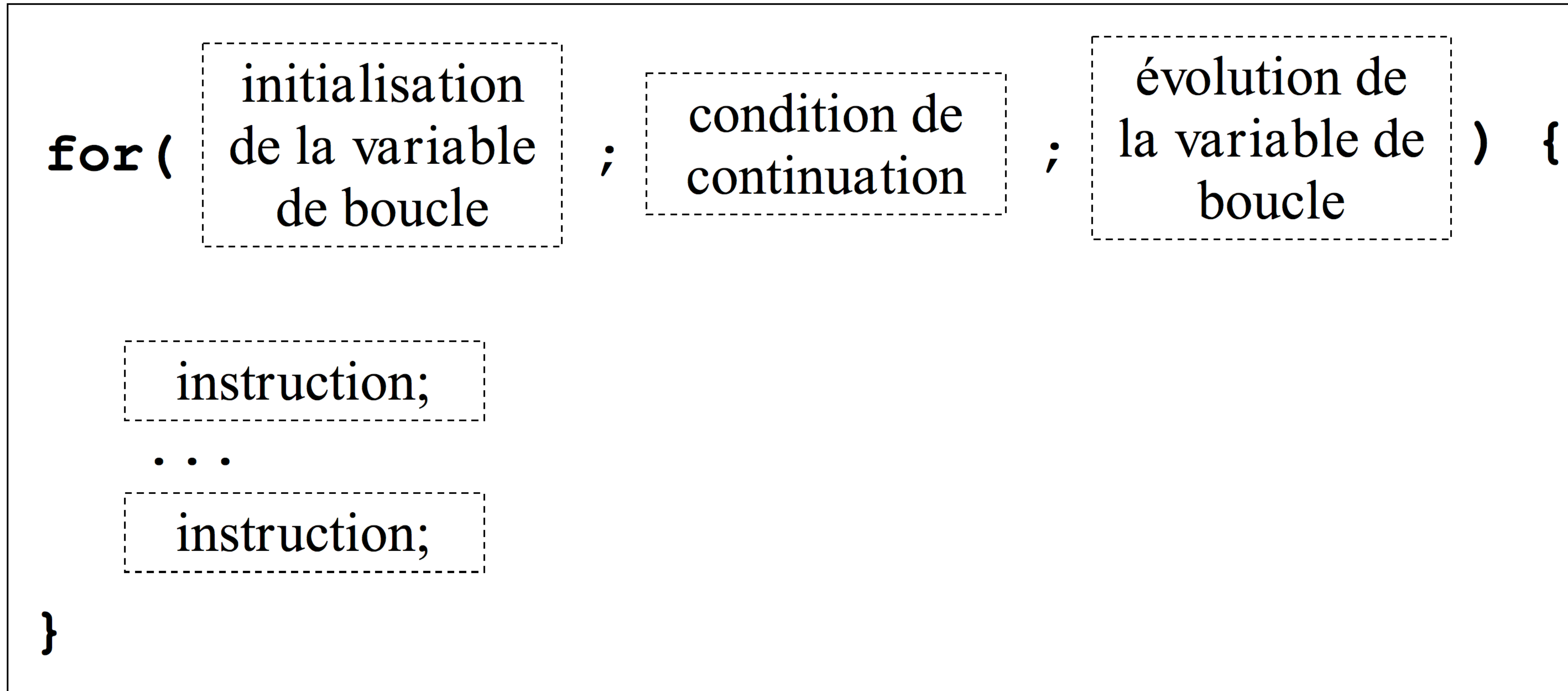
- Comme à la main : **1 * 2 * 3 * 4 * 5 * … * n** (→)
- Soit **res** le résultat entier du calcul, initialisé à **1** (produit vide).
- Pour tout **k** variant depuis **1** jusqu'à **n**, faire **res = res * k**

```
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 :



• 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;
}
```

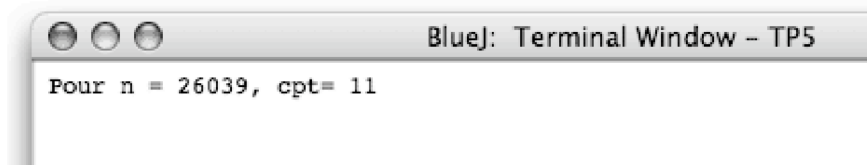
Le corps de boucle est vide ! Il n’y a que **d** qui monte…

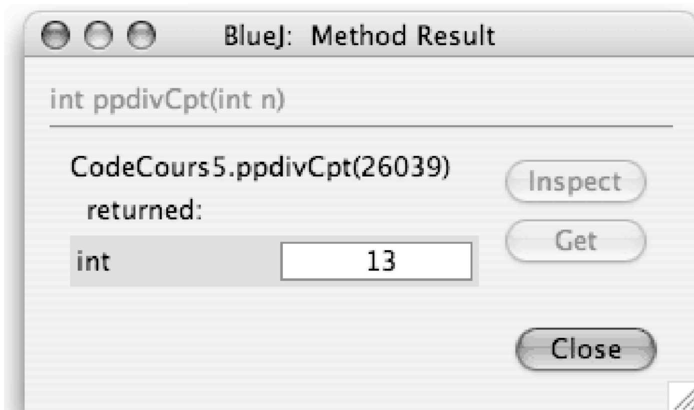
N.B. i) Le corps de boucle est optionnel.

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 :

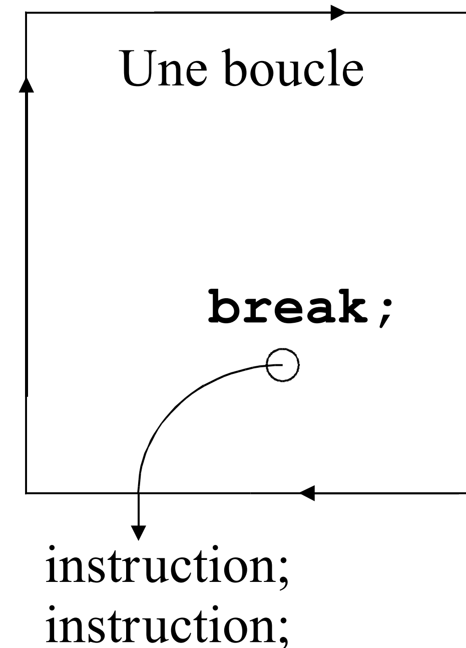
✓ Je sais que la boucle se terminera et que le **`return`** finira par avoir lieu ! En effet, dans le pire des cas, **`d`** sera égal à **`n`**.

✓ 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 !

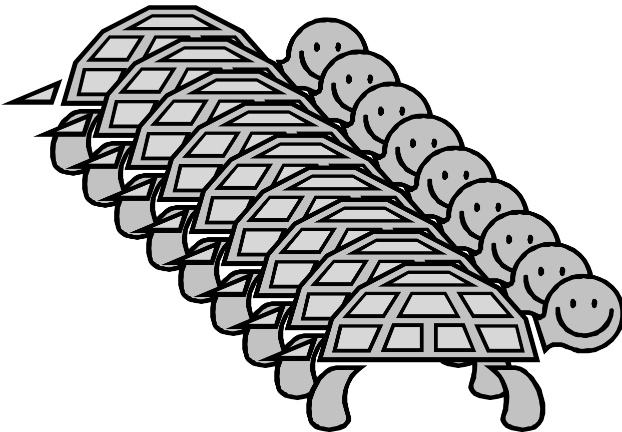
• Le problème cité aux pages 9 et 10 se transpose ipso facto à la boucle **`while`**, dont on peut sortir par un **`break`**.

• 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 ?…

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

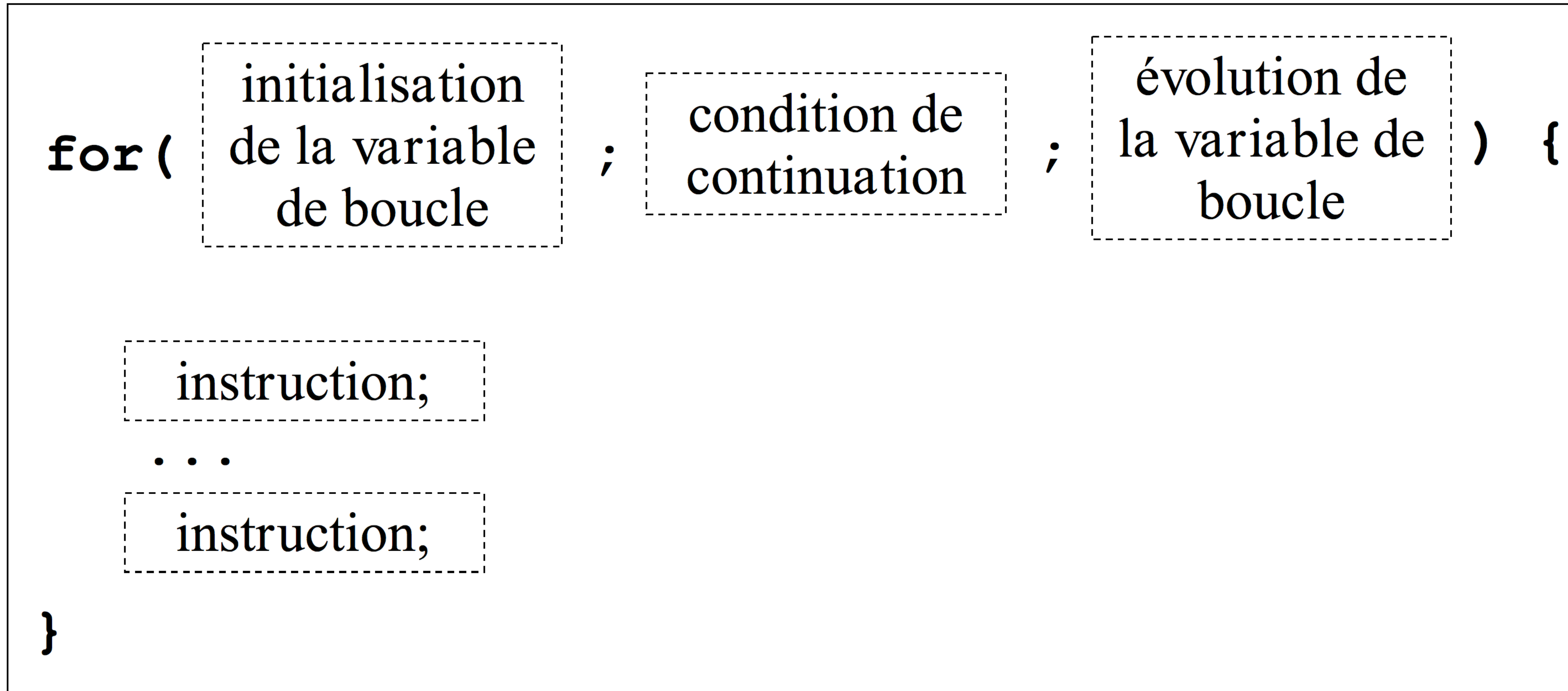
- Comme à la main : **1 * 2 * 3 * 4 * 5 * … * n** (→)
- Soit **res** le résultat entier du calcul, initialisé à **1** (produit vide).
- Pour tout **k** variant depuis **1** jusqu'à **n**, faire **res = res * k**

```
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 :



• 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;
}
```

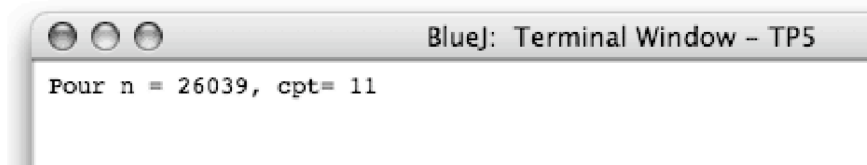
Le corps de boucle est vide ! Il n'y a que **d** qui monte…

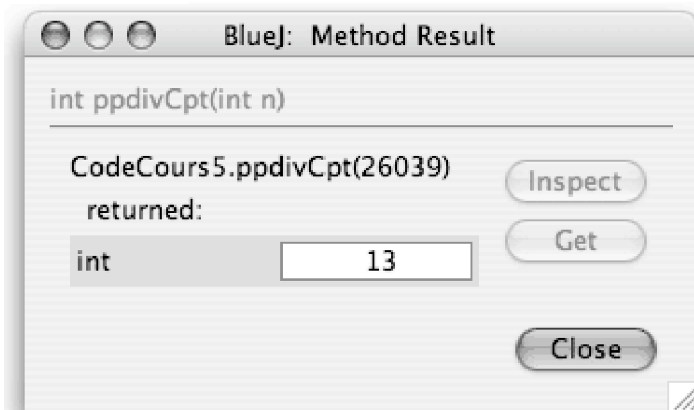
N.B. i) Le corps de boucle est optionnel.

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 :

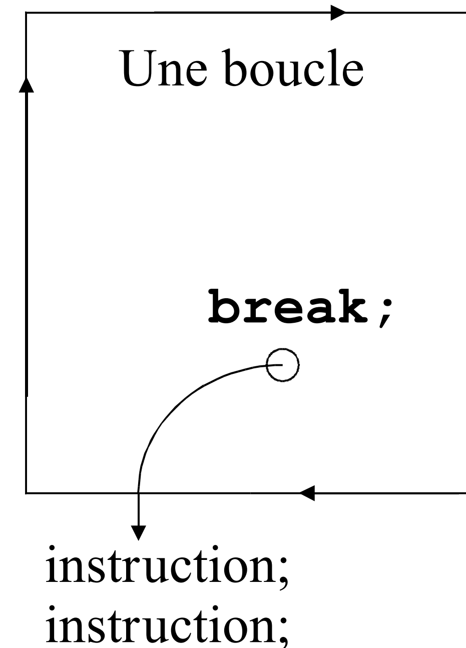
✓ Je sais que la boucle se terminera et que le **`return`** finira par avoir lieu ! En effet, dans le pire des cas, **`d`** sera égal à **`n`**.

✓ 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 ?…