# Université Paris 7 - Denis Diderot

# Licence Sciences et applications

# IF1 : Initiation à l'informatique

# 2008-2009

# 1-ère partie

# Objectifs et organisation

## Objectifs généraux :

- Connaissance élémentaire des ordinateurs
- Initiation à l'algorithmique et à la conception de programmes
- Apprentissage de la programmation impérative
- Initiation à l'approche objet
- Langage utilisé dans cet enseignement : JAVA

## Organisation générale :

- Cours en amphi
- Cours-TD : 2 heures par semaine
- TP :
  - ◇ une séance de 2 h chaque semaine
  - ◇ ET une séance de 2 heures une semaine sur deux

# Contrôle des connaissances :

Pa : Partiel
Ej : Examen de janvier
Er : Examen de rattrapage de juin
Td : Test obligatoire en Cours/TD
Tp : Test obligatoire en TP

Ecrit : NE = max (Ej, (Pa+Ej) / 2)
Cc = (Td + Tp) / 2
Note finale janvier : (3NE + Cc) / 4
Note finale session rattrapage :
max(Er, (3Er + Cc) / 4)

# Bibliographie


- Concepts fondamentaux de l'informatique,
  A. Aho, J. Ullman,
  Dunod, 1998. ISBN 2-10-003127-9
- The Java Tutorial, third edition,
  Mary Campione,Kathy Walrath, Alison Hulm
  Addison-Wesley, décembre 2000
  Disponible sur Internet à l'URL
  java.sun.com/docs/books/tutorial/books/3e/index.html
- Thinking in Java,
  Bruce Eckel
  Disponible sur Internet à l'URL
  http://www.mindview.net


Informations relatives au cours

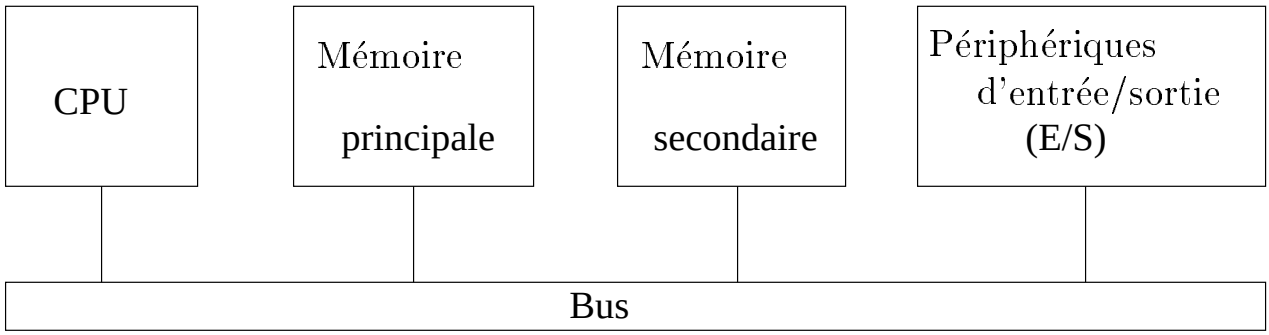
`http://www.pps.jussieu.fr/~rifflet/enseignements/IF1/`

Informations sur la classe Deug

`http://www.liafa.jussieu.fr/~yunes/deug/`

# Schéma général d'un ordinateur

Architecture de **Von Neumann** dont une réalisation a la forme suivante :




- Le bus est le lien de communication entre les éléments. Les données et les instructions transitent sur le bus.

- L'unité centrale de traitement
  CPU (*Central Processing Unit*)

  – Collection de circuits qui gère l'activité de l'ordinateur.
  – On l'appelle souvent le processeur.
  – Va chercher les instructions en mémoire principale et les exécute

- La mémoire principale
  - Stocke les programmes qui s'exécutent ainsi que les données qu'ils utilisent.
  - On la nomme aussi RAM
    ($Random\ Access\ Memory$)
  - Les données qui y sont inscrites disparaissent quand on éteint l'ordinateur (c'est une mémoire *volatile*)

- La mémoire secondaire
  - Formée par des disques de divers types ou des bandes magnétiques.
  - Elle est *permanente*, c'est-à-dire non volatile
  - Elle permet en général de stocker beaucoup plus de données que la mémoire principale
  - Le temps d'accès à une donnée y est beaucoup plus grand que dans le cas d'un accès en mémoire centrale
- Les périphériques d'entrée-sortie
  - Relient l'ordinateur au monde extérieur.
  - Par exemple : souris, clavier, écran, imprimante, réseau, . . .

## Quelques unités

- Dans un ordinateur, informations représentées par des suites d'informations binaires représentées de manière symbolique par "0" et "1", et appelés bits (*binary digits*).
- un octet (*byte*) = 8 bits
  Exemple : les suites de 8 bits 01000001 et 11100010 correspondent à des valeurs d'octets
- un mot = 4 ou 8 octets (32 ou 64 bits)
- Le code ASCII associe à chaque caractère un octet
  Exemple : 'A' est codé par01000001 et
  '1' par 00110001
  Un autre codage des caractères (Unicode) associe aux caractères sur 2 octets afin de permettre l'utilisation d'alphabets différents
- des abrévations courantes :
  - le kilo informatique est $= 2^{10} = 1024$
    On parle de Ko pour Kilo-octets
  - le Mega est $2^{20}$ et on parle de Mo (mega-octets)
  - Go (giga: $2^{30}$), To (tera: $2^{40}$)
  - on parle de milli ($10^{-3}$), de micro ($10^{-6}$), nano ($10^{-9}$) et pico ($10^{-12}$)
    (des millisecondes ou des microsecondes par exemple)
- Fréquence du processeur: exprimée en Hertz MHz (megaHertz), et GHz (GigaHertz)

Traitement Informatique : traitement automatisé de l'information avec des ordinateurs.

Face à un problème à résoudre, un informaticien procède en plusieurs étapes :

- Phase de *modélisation* : il s'agit de décrire l'information à traiter de sorte qu'elle soit compréhensible par une machine.
- Phase de mise au point d'un algorithme

# Exemple

Organisation d'examens en fin d'année.

- des étudiants doivent passer un ou plusieurs examens.
- chaque étudiant est inscrit à certains cours, mais pas forcément à tous, et doit passer les examens correspondants.
- si un même étudiant est inscrit dans deux cours donnés, les examens qu'il devra passer pour ces cours doivent être organisés dans des créneaux horaires différents.

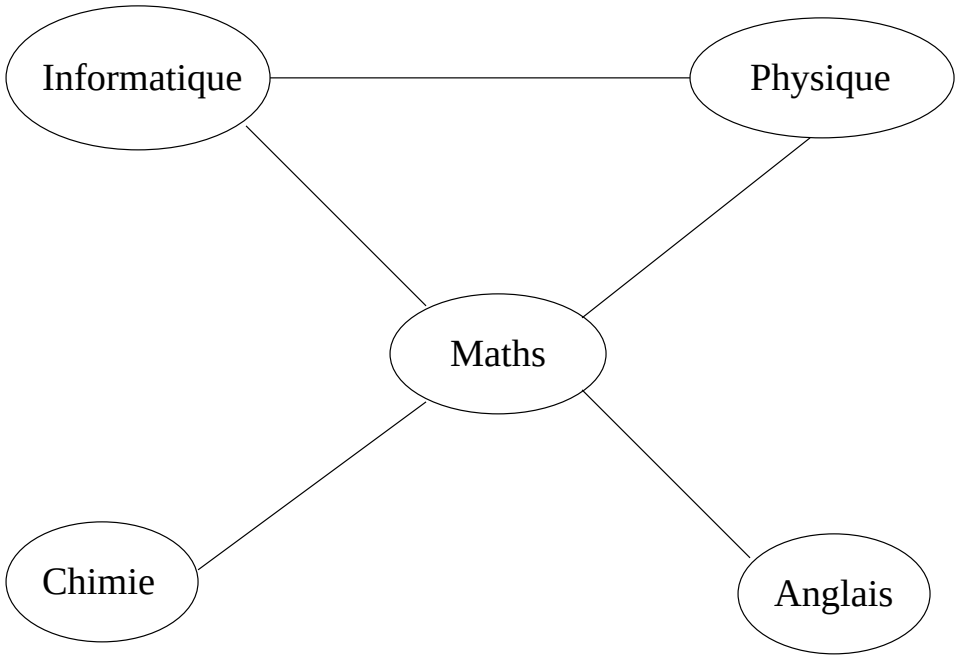
Comment arriver à écrire un programme informatique qui permette de résoudre ce problème ?

# Modélisation

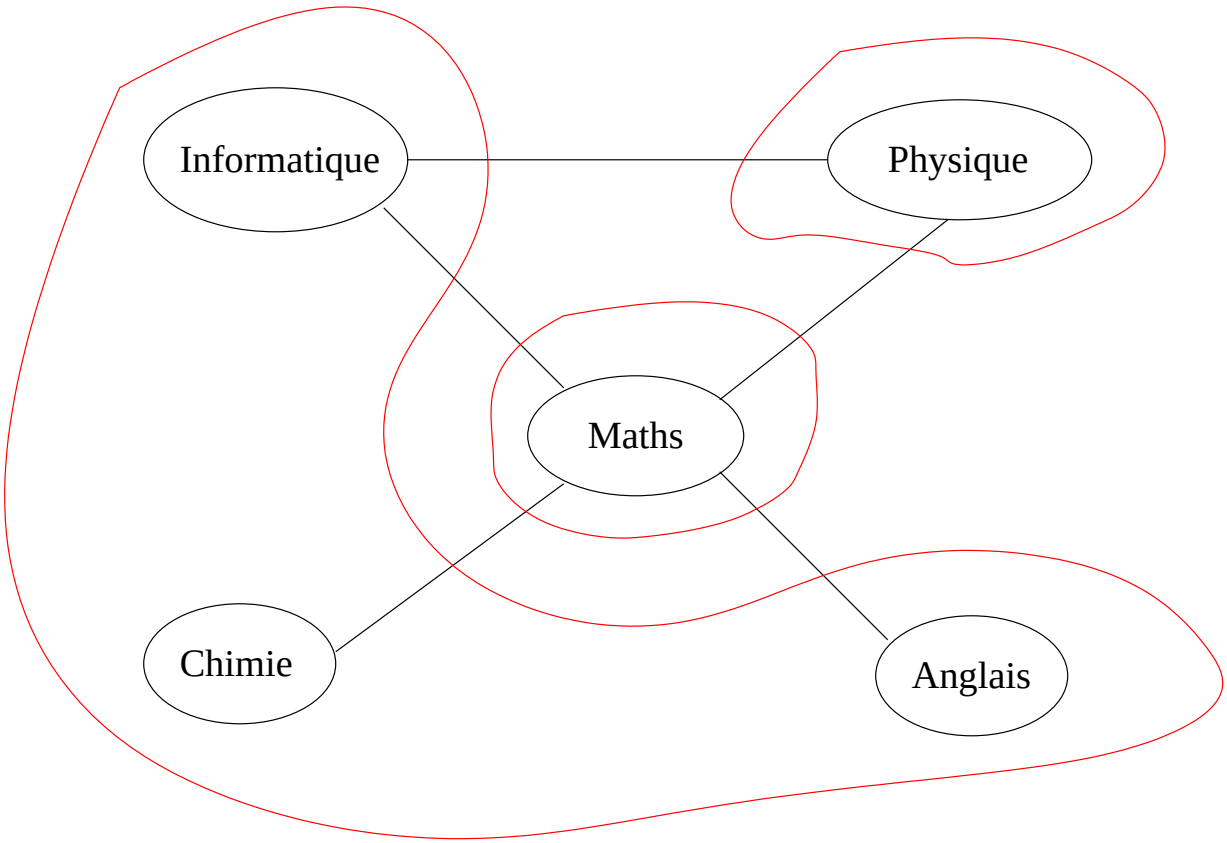
Supposons que l'on doive gérer un ensemble de 13 étudiants que l'on nommera A, B, ..., M, qui suivent les cours suivants :

- Informatique : {A, B, C, D, E, F}
- Mathématiques : {A, B, G, H, M}
- Physique : {B, C, D, F}
- Chimie : {G, H, I, J}
- Anglais : {K, L, M}

On va représenter chaque cours par un rond et on mettra un trait entre deux ronds s'il y a au moins un étudiant inscrit aux cours correspondants.



Une solution au problème initial consistera donc à allouer des créneaux horaires aux cours de sorte que des cours qui sont reliés par un trait ne soient pas dans le même créneau.




Une solution :

- Créneau horaire 1 : Informatique, chimie et anglais.
- Créneau horaire 2 : Mathématiques.
- Créneau horaire 3 : Physique.

# Algorithme

Il faut trouver une méthode générale (un algorithme) définissant une suite d'opérations permettant de résoudre le problème de manière automatique.

Dans l'exemple : très simple... mais on peut aisément imaginer un cas beaucoup plus complexe
(Paris 7 : 30 000 étudiants, 400 cours différents...).

Un algorithme pourrait par exemple consister à :

- choisir un ensemble de cours, le plus grand possible, tel qu'aucun étudiant ne soit inscrit à deux ou plus de ces cours.
- placer les examens qui correspondent à ces cours sur le premier créneau horaire disponible.
- puis recommencer le même travail avec les cours restants.

D'autres algorithmes seraient bien sûr possibles.

# Programme

Il reste ensuite à transformer l'algorithme choisi en programme

Pour cela, il existe de nombreux langages de programmation qui peuvent être compris par un ordinateur :

- FORTRAN,
- PASCAL,
- ADA,
- C,
- C++,
- ML, CAML
- LISP,
- JAVA, ...

Le langage que nous utiliserons est le langage **JAVA**

# Remarques

- Pour un même problème il existe en général plusieurs modélisations.
- Pour un même modèle, il existe en général plusieurs algorithmes.
- Pour un même algorithme, il existe en général plusieurs programmes.

L'objectif de l'informatique est de maîtriser cette chaîne :

Problème $\longrightarrow$ Modèle $\longrightarrow$ Algorithme $\longrightarrow$ Programme

Différents points sont à considérer :

- Comment représente-t-on et traite-t-on des données en machine ?
- Comment écrit-on un algorithme pour résoudre un problème ?
- Comment écrit-on un programme à partir d'un algorithme ?

# Programmation : les différents niveaux d'abstraction.

1. Matériel : logique numérique, microprogrammes
2. Langage machine : langage codé sous forme binaire et dépendant du processeur (instructions élémentaires permettant d'ajouter 2 entiers, déplacer une donnée, . . . )
3. Langage assembleur : représentation symbolique des instructions et données, généralement dépendant du processeur
4. Langage de programmation évolué : C, C++, Fortran, Java, Lisp, Caml, . . . ... A priori ils sont universels.

Exemple : Calcul de la surface d'un cercle de rayon 5 :

`surface =` $\pi$ `* 5`$^2$

- en langage bas niveau (type assembleur)

```
entrer 3.1415926535
multiplier par 5
multiplier par 5
mémoriser surface
```

- en langage C ou C++, voire en Java

```
surface = 3.1415926535 * 5 * 5;
```

- en langage Java

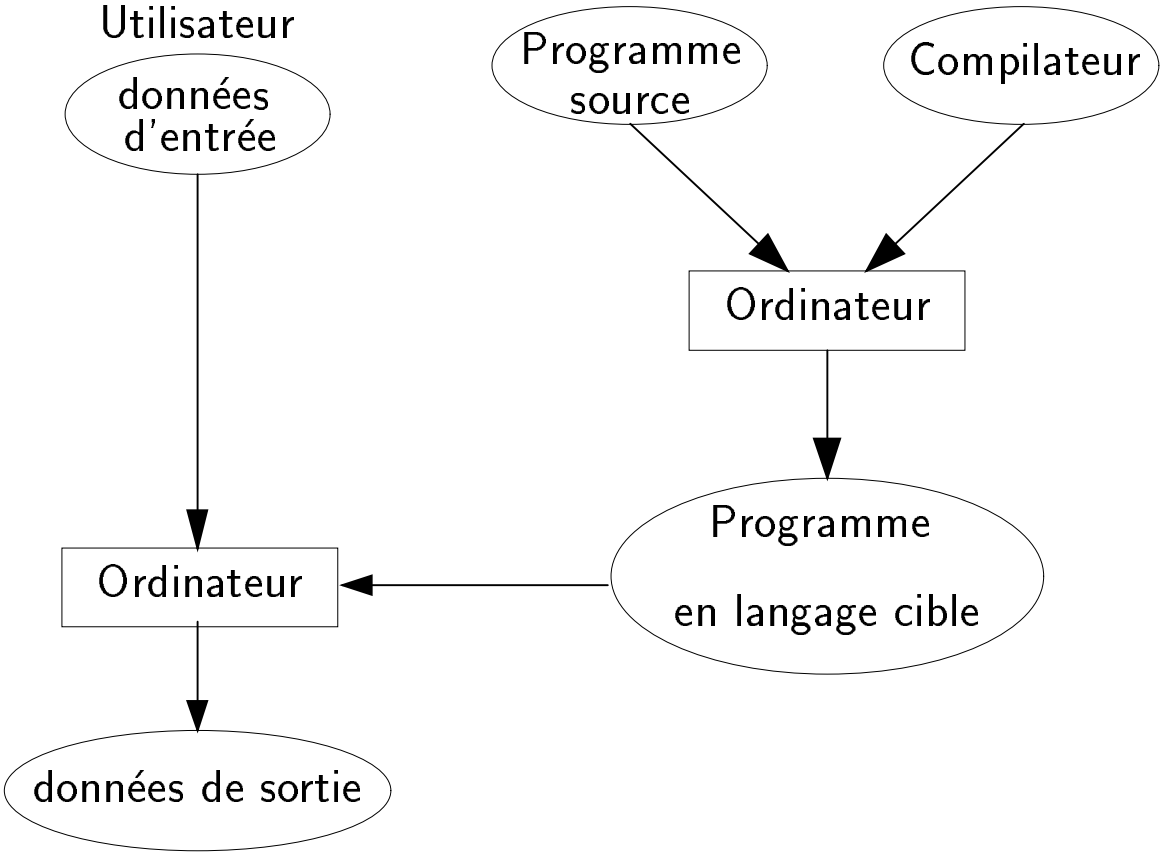
```
surface = Math.PI * 5 * 5;
```

ou `surface = Math.PI * Math.pow(5,2);`

# Cycle du programme

Édition dans un fichier, en utilisant un éditeur de textes

Compilation : un compilateur est un programme spécial qui traduit un programme écrit dans un langage source en un autre écrit dans un autre langage (langage cible)



Le langage cible peut être :

- le langage machine correspondant au processeur sur lequel le programme obtenu sera exécutable directement (il faut noter que ce n'est pas nécessairement le même processeur que celui sur lequel le compilateur est exécuté, on parle alors de compilation croisée)
- un langage intermédiaire exécutable par un programme particulier C'est le cas de Java où le compilateur génère ce qu'on appelle du $\underline{bytecode}$ interprété par une machine virtuelle. Le code obtenu est portable.

$Remarque$ : les programmes écrits dans certains langages sont directement interprétés à la volée sans phase de compilation (par exemple les shells d'Unix)

La chaîne de production peut produire :

- un programme exécutable correct
- un programme exécutable incorrect (provoquant des erreurs d'exécution ou donnant des résultats faux)
- des erreurs lors de la compilation

# Exemples de programmes JAVA

Affichage à l'écran d'un message : fichier `Hello.java`

```
import fr.jussieu.script.*;
class Hello {
   public static void main(String[ ] args) {
      // affichage de  Hello World!
      Deug.println("Hello World!");
   } // fin de la définition de main
} // fin de la définition de Hello
```

Compilation : le fichier `Hello.java` contient le source

```
--> javac Hello.java
-->
```

Exécution : un fichier `Hello.class` produit par le compilateur est supposé exister :

```
--> java Hello
Hello World!
-->
```

L'exécution n'est possible que si le fichier contient la définition d'une méthode (fonction) `main` avec les bonnes spécifications (`void`, `static`, `public` et un paramètre tableau de `String`))

# Exemple 2 : Puissance entière

Puissance entière : fichier `Puissance.java`

```
import fr.jussieu.script.*;
class Puissance {
  static int puissance(int x, int n){
    int i, val = 1;
    for(i = 1; i <= n; i++)
         val = val * x;  // val *= x;
    return val;
  }
  public static void main(String [ ]args){
    int nombre, exposant, res = 1;
    Deug.print("Donner un nombre entier ? ");
    nombre = Deug.readInt();
    Deug.print("Donner une puissance positive ou nulle ? ");
    exposant = Deug.readInt();
    if (exposant < 0){
        Deug.println("L'exposant doit etre positif");
        Deug.exit();
        }
    for(int i = 1; i <= exposant; i++)
         res = res * nombre;
    Deug.println("Valeur : " + res);
    Deug.println("Valeur : " + puissance(nombre, exposant));
    Deug.println("Valeur : " + Math.pow(nombre, exposant));
   }
}
```

Dans cet exemple, la puissance d'un nombre est calculée :

- une première fois, directement dans le corps du programme
- une deuxième fois, en faisant appel dans le corps du programme à une fonction écrite par le programmeur
- une troisième fois en faisant appel dans le corps du programme à une fonction fournie par le système (fonction `pow` de la classe `Math`, la valeur renvoyée étant alors un nombre réel)

Compilation : fichier source `Puissance.java`

```
--> javac Puissance.java
-->
```

Exécution : `Puissance.class` existant

```
--> java Puissance
Donner un nombre entier ? 3
Donner une puissance positive ou nulle ? 4
Valeur : 81
Valeur : 81
Valeur : 81.0
-->
```

# Erreurs à la compilation

Si on considère le programme suivant de calcul de surface d'un cercle :

```
import fr.jussieu.script.*;
class Erreurs1{
   public static void main( String [ ] args) {
       Deug.println(Math.PI * 5 * 5 );
       Deug.println(PI * Math.pow(5,2))
       }
}
```

Sa compilation (commande javac) va conduire à :

```
Erreurs1.java:5: ';' expected
       Deug.println(PI * Math.pow(5,2))
                                       ^

Erreurs1.java:5: cannot resolve symbol
symbol  : variable PI
location: class Erreurs1
       Deug.println(PI * Math.pow(5,2))
                    ^

2 errors
```

- ◇ manque d'un ; (point virgule) en fin d'instruction
- ◇ interprétation impossible de l'identificateur PI

**En conséquence, pas de nouveau fichier `.class` généré**

# Erreurs à l'exécution

Un premier exemple :

```
import fr.jussieu.script.*;
class Erreurs2{
   public static void main( String [ ] args) {
       int c, a = 3, b = 0;
       c = a / b;
       Deug.println("FIN");
       }
}
```

se compile correctement mais donne lieu à une exception lors de l'exécution du fait d'une tentative de division entière par zéro

```
--> ls Erreurs2*
Erreurs2.java
--> javac Erreurs2.java
--> ls Erreurs2*
Erreurs2.class       Erreurs2.java
--> java Erreurs2
  Exception in "main" java.lang.ArithmeticException:
                / by zero
   at Erreurs2.main(Erreurs2.java:5)
-->
```

Dans ce second exemple, une méthode `main` est définie mais ne répond pas à toutes les spécifications attendues par la machine Java (ici pas de paramètres) :

```
import fr.jussieu.script.*;
class MauvaisMain {
   public static void main( ) {
     //affichage de  Hello World!
     Deug.println("Hello World!");
     } // fin de la definition de main
} // fin de la definition de Hello
```

La compilation se passe normalement (fichier `MauvaisMain.class` construit par compilation du source `MauvaisMain.java` précédent), mais l'exécution provoque une exception :

```
--> javac MauvaisMain.java
--> ls MauvaisMain*
MauvaisMain.class      MauvaisMain.java
--> java MauvaisMain
  Exception in "main" NoSuchMethodError: main:
-->
```

# Description des algorithmes

Programme en JAVA parfois difficilement lisible :

⇒ on commence par écrire un algorithme avant d'écrire un programme.

⇒ on utilise pour cela un langage informel.

- pas de contraintes syntaxiques strictes (pour peu que l'expression en soit compréhensible et non ambigue)
  - expression de l'algorithme en langage naturel
- niveau d'abstraction adapté aux problèmes et aux connaissances

Exemple :

- lire les données
- trier les données
- afficher les données triées

# Exemple : Puissance entière

L'algorithme de calcul de $a^n$ peut s'exprimer de manière plus ou moins détaillée :

Première forme :

> multiplier $n - 1$ fois le nombre $a$ par lui-même,
> n étant la puissance

Deuxième forme :

- lire le nombre a et l'exposant n
- calculer les $n + 1$ premiers termes de la suite $R_i$ avec $R_0 = 1$, $R_{i+1} = R_i * a$
- le résultat est $R_n$

Troisième forme : plus détaillée, elle ressemble fort à un programme. Essentiellement elle exprime la manière dont on réalise $n$ fois une certaine séquence d'opérations (ce qu'on appelle une boucle)

La solution s'appuie sur le fait qu'il n'est pas nécessaire de conserver les valeurs de tous les $R_i$. L'indice $i$ peut être vu comme une indication temporelle pour les valeurs successives d'une même variable appelée ici $R$ :

```
lire le nombre réel a et la puissance entière n
vérifier que n est positif
      (sinon soit relire, soit s'arrêter)
initialiser une variable R à 1
pour i valant successivement 1, 2, ...  n faire
   affecter à R la valeur courante de R multipliée par a
afficher la valeur de R
```

# Exercices

On demande ici des descriptions informelles des algorithmes.

1. Décrire un algorithme qui lit une température en degré Celsius et affiche son équivalent en degré Fahrenheit. La formule utilisée pour la conversion est
   degré Fahrenheit=degré Celsius*1.8+32.
2. Décrire un algorithme qui calcule la somme et la moyenne de deux nombres.
3. Décrire un algorithme qui calcule la somme et la moyenne de trois nombres.
4. Décrire un algorithme qui calcule le maximum de deux nombres. Par rapport aux algorithmes précédents de quel nouveau type d'opération a-t-on besoin?
   Décrire deux algorithmes différents pour calculer le maximum de 4 nombres
5. Si on souhaite réaliser la recherche du maximum ou calculer la somme et la moyenne d'un nombre variable de valeurs, de quel(s) mécanisme(s) a-t-on besoin?
6. Décrire l'algorithme réalisant à la main le calcul du nombre $c$ somme de deux nombres $a$ et $b$ écrits en base $10$.
   On supposera que $a$ s'écrit $a_m a_{m-1} \ldots a_1 a_0$ et que $b$ s'écrit $b_m b_{m-1} \ldots b_1 b_0$ (les $a_i$ et $b_i$ sont des chiffres décimaux).

# Identificateurs

Pour désigner les éléments d'un programme (classes, variables, objets, fonctions, méthodes, constantes), donc donner un nom à quelque chose dans un algorithme ou un programme, on utilise un identificateur.

Identificateurs en Java

- identificateur = une **lettre** (le caractère _ est une lettre) suivie d'une suite de lettres ou de chiffres, de longueur quelconque
- minuscules et majuscules sont des lettres distinctes
  Ainsi `XaX` et `XAX` sont des identificateurs différents et il en est de même pour `X1` et `x1`
- exemples :

```
x, y, m1, ABC, Nombre_Max, ...
```

# La notion de type

Un type correspond à la définition de :

- un ensemble de valeurs
- un ensemble d'opérations applicables aux éléments de cet ensemble et la sémantique de ces opérations

En JAVA, on utilise le terme de classe,

- toute variable a un type
- toute expression a un type

Le typage impose des limites au mélange qu'on peut réaliser entre différentes valeurs

Le langage JAVA distingue

- les types primitifs (caractères, entiers, réels et booléens)
- les types références : les tableaux, les classes fournies par les nombreux paquetages (*packages*) Java dont le type (classe) String et les classes définies par les utilisateurs

# Les types primitifs

- le type booléen : `boolean`
  Valeurs `true` et `false`
  opérations : ET (`&&` ou `&`), OU (`||` ou `|`) et NON (`!`)
  Il existe aussi un opérateur OU exclusif (`^`)
- le type caractère : `char`
  En Java, caractères codés sur 2 octets (Unicode), ordre alphabétique respecté
- les types entiers :
  - `byte` : 1 octet
  - `short` : 2 octets
  - `int` : 4 octets
  - `long` : 8 octets
  - arithmétique : moins unaire (-), addition (+), soustraction (-), multiplication (*), quotient entier (/), reste (%)
  - comparaison : égal (==), différent (!=), plus grand (> ou >=), plus petit (< ou <=)
- les types réels :
  - `float` : 4 octets (précision simple)
  - `double` : 8 octets (double précision)
  - arithmétique : - unaire, +, -, *, quotient /
  - comparaison : ==, !=,>, >=, <, <=

# Constantes littérales des types primitifs

- `boolean` : `true` et `false`
- `char` : `'A'`, `'c'`, `'\n'`, `'\uec34'` (code Unicode hexadécimal commençant par `\u`)
- types entiers : l'écriture traditionnelle d'un entier est interprétée comme une valeur décimale de type `int`. Une telle écriture suivie de `l` ou `L` correspond à une constante de type long

  valeurs comprises entre $-2^{31}$ =-2147483648 et $2^{31}-1$ =2147483647

  2456, -234567 : valeurs décimales

  -0234567 : valeurs octales (matérialisées par le chiffre 0 au début après le signe)

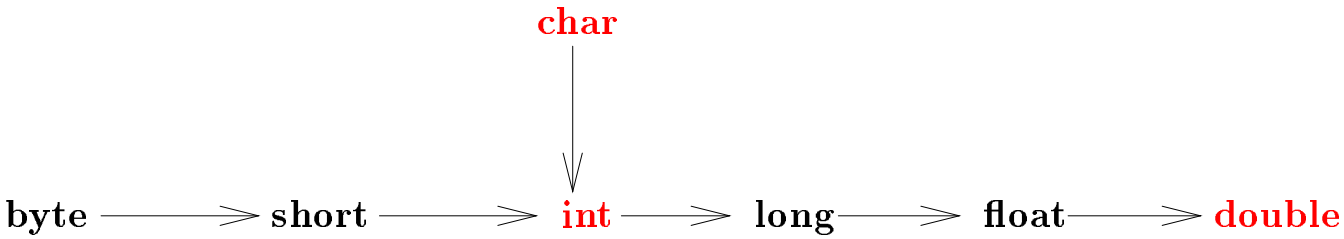
  -0x23456A7 : valeurs hexadécimales (matérialisées par 0x au début après le signe)
- types réels : nombre décimal traditionnel interprété de type `double` (de type `float` si suivi de `f` ou `F`)

  2456.17

  0.00245617

  23.345e-6

# Ordre des types primitifs




Cet ordre définit la compatiblité entre les types primitifs, en particulier les conversions implicites possibles.

Par exemple

- une valeur du type `int` peut être vue comme une valeur de type `long` mais pas le contraire
- une valeur du type `int` peut être vue comme une valeur de type `float` mais pas le contraire

Il détermine en particulier

- le type des expressions
- les opérations qui sont ou non possibles

Le type boolean est incompatible (incomparable) avec les autres types primitifs

# Variables

- une variable correspond à un emplacement en mémoire pouvant contenir une valeur d'un type donné : c'est le type de la variable qui permet l'interprétation correcte du contenu correspondant en mémoire
- une variable est désignée par un identificateur valide
- à une variable peut être affectée une valeur du type correspondant (ou d'un type compatible)

## Déclarations de variables

Toute variable doit être déclarée avant d'être utilisée :

- variables de types primitifs : la déclaration d'une variable de type primitif alloue la place en mémoire pour y stocker une valeur du type correspondant

```
/* déclarations multiples et une initialisation */
int entier1, entier2 = 13;
float reel1, reel2 = 123.45F;
```

- variables références : la déclaration d'une telle variable alloue la place pour stocker l'adresse en mémoire d'un objet du type correspondant. L'espace nécessaire pour l'objet devra être alloué avec l'opérateur `new`

# Affectation

Opération permettant d'attribuer à une variable d'un type donné une valeur du type correspondant :

- si $x$ est une variable de type entier, on peut lui affecter la valeur 5 mais pas la valeur 3.14
- l'affectation d'une valeur écrase (remplace) la valeur préalablement associée :

  la valeur correspondant à une variable peut donc changer au cours du temps (il est cependant possible de définir des constantes dont la valeur ne peut changer)
- notation dans les langages de description d'algorithmes :
  $\leftarrow$ (ou `<--` ou `:=`)
  Ainsi :

  ```
  x <-- 5
  ```

  se lit "x prend la valeur 5" ou "on affecte 5 à x".
  Après exécution de cette instruction et jusqu'à ce qu'une autre affectation soit réalisée sur $x$, la variable $x$ aura la valeur $5$

- en JAVA, l'opérateur d'affectation se note =

```
int x, y;
float f;
x = 34; y = -0x34;
f = 3.14f;
```

## Forme générale d'une affectation en JAVA

Une instruction permettant d'affecter une valeur à une variable a la forme générale suivante en JAVA :

<nomVariable> = <expression>;

Le caractère ; (point virgule) marque la fin de l'instruction (de manière générale il marque la fin de toute instruction)

Le type de la variable doit être compatible avec celui de l'expression

```
public static void main( String [ ] args) {
   int n; long m; float x; double y;
   n=1234; m=2345; x=234f; y=x;  // corrects
   n=m; x=3456.1;   // incorrects
   }
 }
```

## Définitions de constantes

Une constante est une variable portant le qualificatif `final`. Une telle variable ne peut être modifiée (une seule affectation possible lors de sa définition).

La compilation du programme :

```
class Constantes1{
  public static void main( String [ ] args) {
    final int i=3;
    i = 4;
  }
}
```

va signaler une erreur :

```
Constantes1.java:4: cannot assign a value
                          to final variable i
      i = 4;
      ^
1 error
```

# Expressions

- Une expression est composée de variables et/ou constantes combinées avec des opérateurs
- Une expression a un type et une valeur.
  Le type et la valeur de l'expression dépendent des éléments qui la composent (variables, constantes et opérateurs) :
  - ◇ si i et j sont entiers, i/j est une division entière alors que si l'un des deux ne l'est pas la division est réelle :

```
import fr.jussieu.script.*;
class Quotient{
   public static void main(String[ ] args){
    Deug.println(4/3); Deug.println(4/3.0);
   }
}
```

donne :

```
1
1.3333333333333333
```

  - ◇ `x+y/z` est une expression arithmétique dont le type dépend des types de `x`, `y` et `z`,
  - ◇ `(x>y) || !(x==y+1)` est une expression booléenne
- l'expression doit être syntaxiquement correcte :
  `3+4x-z` n'a pas de sens

- l'expression doit être valide pour le typage :
  `X - true` n'a pas de sens
- les règles de priorité permettent d'éliminer des parenthèses.

  COMPARER LES EXPRESSIONS:
  `x+y/z` et `(x+y)/z`
  `a-b-c` et `a-(b-c)`
- attention à la compatibilité des types :

  ```
  float f = 3.14;
  ```

  va produire une erreur de compilation :

  ```
  Incompatible type for =.
  Explicit cast needed to convert
                        double to float
  ```

  Il faut écrire (conversion explicite) :

  ```
  float f = (float)3.14;
  ```

De même il faut écrire

```
int n  = (int)3.14;
Deug.println(n); // on obtient 3
```

# Les expressions arithmétiques

- construites avec les opérateurs arithmétiques, des constantes littérales et des variables, des parenthèses pour forcer un ordre d'évaluation
- ordre de priorité des opérateurs
- à priorité égale, évaluation de gauche à droite.
  Ainsi :
  - ◇ a+b-c-d+e s'évalue comme (((a+b)-c)-d)+e
  - ◇ a/b/c*d/e*f s'évalue comme ((((a/b)/c)*d)/e)*f
- toute expression arithmétique a un type
  (le plus petit type entier pour une expression est `int`) :
  - si l'expression est une constante, le type est celui de la constante
  - si l'expression est composée, la sous-expression correspondant à l'opérateur de plus petite priorité est évaluée (le type le plus grand parmi les opérandes détermine celui de l'expression)

**Remarque**: avant de pouvoir utiliser une variable dans une expression, il est nécessaire qu'une valeur lui ait été affectée.

Le compilateur JAVA signale comme erreur les variables utilisées mais non initialisées.

La compilation du programme suivant:

```
import fr.jussieu.script.*;
class VariableNonInit {
   public static void main(String[ ] args) {
       int m, n = 3, p, q;
       p = m + n;
       Deug.println(q);
       }
}
```

donne:

```
Variable.java:5: variable m might not
                        have been initialized
          p = m + n;
Variable.java:6: variable q might not
                        have been initialized
          Deug.println(q);
2 errors
```

# Expressions et affectations

Lors d'une affectation à une variable d'une expression, l'expression est évaluée au moment de l'affectation. C'est la valeur de l'expression au moment de cette évaluation qui est affectée à la variable. Aucune trace de la manière dont cette valeur a été obtenue n'est conservée. Ainsi la séquence

```
  ...
x = 4;
y = 6;
z = x + y;
Deug.println(z);
y = 20;
Deug.println(z);
  ...
```

donne à l'exécution

```
10
10
```

La modification de la variable y après affectation de la variable z n'a aucun effet sur la valeur de la variable z

# Priorité des opérateurs JAVA

Le tableau ci-dessous contient les opérateurs principaux :

| Niveau | Opérateur |
|---|---|
| 1 | ++ -- |
| 2 | ! (*type*) |
| 3 | * / % |
| 4 | + - |
| 5 | < > <= >= |
| 6 | == != |
| 7 | & |
| 8 | ^ |
| 9 | \| |
| 10 | && |
| 11 | \|\| |
| 12 | = += -= *= . . . |

On utilise des parenthèses pour

- modifier la priorité,
- améliorer la lisibilité.

# Exercice

On considère des variables entières identifiées respectivement par un, deux, trois et cinq
(**il s'agit d'identificateurs, on aurait pu choisir d'autres noms tels que a, b, c et d par exemple**)

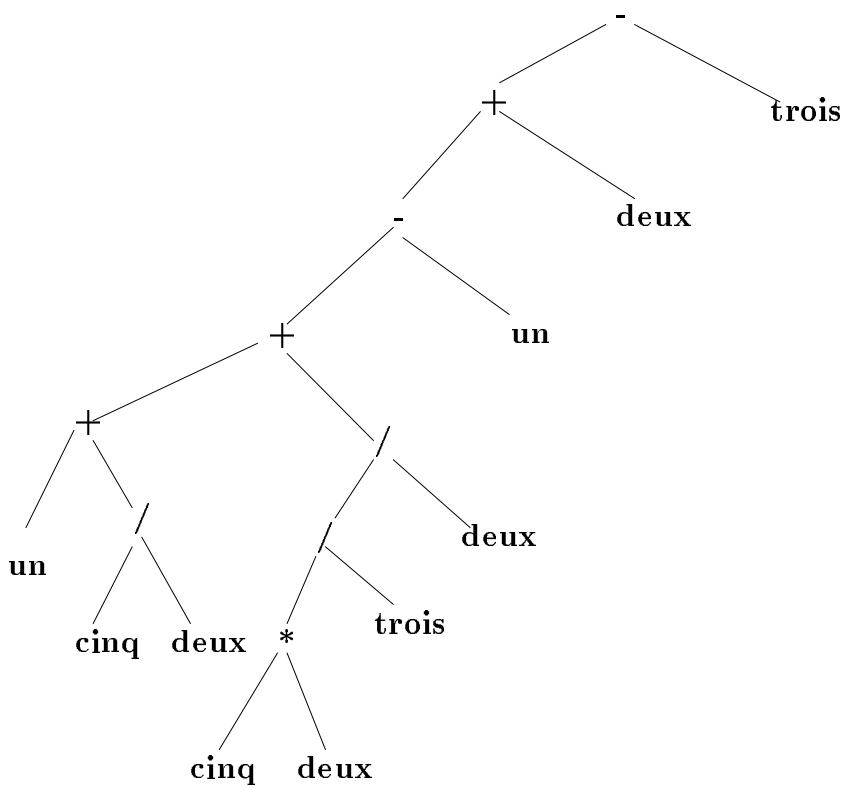
Soit l'expression $E_1$:
`un+cinq/deux+cinq*deux/trois/deux-un+deux-trois`

1. Cette expression est-elle syntaxiquement correcte?
2. En appliquant les règles de priorité de Java, transformer cette expression en expression totalement parenthésée
3. On suppose que `un`, `deux`, `trois` et `cinq` sont des variables de type `int` qui valent respectivement $1$, $2$, $3$ et $5$ quelle est la valeur de cette expression?
   (*la valeur de l'expression est 2*)
4. Mêmes questions avec l'expression $E_2$

`un+cinq/(deux+cinq)*deux/trois/(deux-un+deux)-trois`
(*la valeur de l'expression est -2*)

## Arbre associé à une expression

Il est commode d'associer à une expression. un arbre reflétant la manière dont elle est construite et donc dont elle sera évaluée.
Ainsi, pour $E_1$:




Exercice : Construire l'arbre associé à l'expression $E_2$

# Calculs impossibles et calculs incorrects

- en calcul entier:
  - ◇ une division par zéro va donner une erreur à l'exécution (exception en Java)
    java.lang.ArithmeticException: / by zero . . .
  - ◇ débordements lors d'opérations (overflow/underflow): les opérations sur les entiers 32 bits sont réalisées modulo $2^{32}$:

    ```
    int i = 2147483647;
    Deug.println(2*i); Deug.println(10*i);
    ```

    va donner les résultats suivants:

    ```
    -2
    -10
    ```

- en calcul réel:
  - ◇ un calcul réel est toujours possible: il existe des valeurs spéciales produites par certaines opérations:

    ```
    double f = 1.3, g = 0;
    Deug.println(f/g);
    Deug.println((-f)/g);
    Deug.println(g/g);
    ```

va afficher les résultats suivants:

```
Infinity
-Infinity
NaN
```

◇ problèmes de précision:

```
double d = 1.2345678912345678 9;
float f = 1.2345678912345678 9f;
Deug.println(d);
Deug.println(f);
```

va donner les résultats suivants:

```
1.234567891234568
1.2345679
```

Si on compare les valeurs des expressions
(0.3 - 0.2) et (0.2 - 0.1)
elles vont apparaître différentes :
La séquence

```
boolean b = ((0.3-0.2)==(0.2-0.1));
Deug.println(b);
```

affiche comme résultat

```
false
```

# Opérateurs booléens

Les opérations sur les booléens correspondent aux tables suivantes (les valeurs booléennes sont représentées symboliquement par Vrai et Faux):

| ET | Faux | Vrai |
|---|---|---|
| Faux | Faux | Faux |
| Vrai | Faux | Vrai |

conjonction (ET)

| OU | Faux | Vrai |
|---|---|---|
| Faux | Faux | Vrai |
| Vrai | Vrai | Vrai |

disjonction (OU)

| XOR | Faux | Vrai |
|---|---|---|
| Faux | Faux | Vrai |
| Vrai | Vrai | Faux |

XOR (OU exclusif)

| Faux | Vrai |
|---|---|
| Vrai | Faux |

négation

Expressions booléennes ou logiques : traduction en JAVA

Utilisation de tables de vérité pour évaluer les expressions booléennes.

| A | B | A et B | A ou B | non A | non B | non(A et B) | non(A ou B) | non(A) ou non(B) |
|---|---|---|---|---|---|---|---|---|
| Vrai | Vrai | Vrai | Vrai | Faux | Faux | Faux | Faux | Faux |
| Vrai | Faux | Faux | Vrai | Faux | Vrai | Vrai | Faux | Vrai |
| Faux | Vrai | Faux | Vrai | Vrai | Faux | Vrai | Faux | Vrai |
| Faux | Faux | Faux | Faux | Vrai | Vrai | Vrai | Vrai | Vrai |

# Exercices

1. On considère une expression booléenne construite avec $n$ variables. Combien la table de vérité de cette fonction contiendra-t-elle de lignes ?
2. Donner des expressions booléennes correspondant aux conditions suivantes (les identificateurs correspondent à des variables entières) et en donner les formes JAVA :

- x est supérieur à 3 et le quotient entier de la somme de x et de y par z est un multiple de 4
- si la somme de x et de y est un multiple de 5, alors x et y sont tous les deux des multiples de 5 et sinon (c'est-à-dire si la somme de x et de y n'est pas un multiple de 5), x et y sont des multiples de 4
- $n$ désigne une année bissextile (multiple de $4$, à l'exception des multiples de $100$ et pas de $400$)

3. On considère les intervalles `[a:b]` et `[c:d]` de nombres. Donner une expression booléenne vraie si les intervalles sont disjoints ou contenus l'un dans l'autre
4. Écrire la formule logique qui exprime le fait qu'un entier corresponde ou non à une année bissextile

5. Vérifier l'équivalence des expressions logiques suivantes (lois de Morgan):

- `NON(a OU b)` est équivalente à `(NON a) ET (NON b)`
- `NON(a ET b)` est équivalente à `(NON a) OU (NON b)`

6. Écrire les tables de vérité des expressions booléennes suivantes (le caractère `~` représente la négation) et en donner les expressions correspondantes de type `boolean` en JAVA:

- `a OU (~(b ET ~a) ET (a OU ~b))`
- `(a OU b) OU (~(b ET ~a) ET (a OU ~b))`
- `~((~a OU b) ET c) OU (a ET ~b)`

Au passage on notera que la deuxième est une tautologie (expression toujours vraie)

7. Exprimer chacune des expressions précédentes sous une forme dite normale disjonctive. Sous cette forme, une expression à $n$ variables $x_1, \ldots, x_n$, est équivalente à une expression écrite sous la forme d'une disjonction de $m$ termes

$$t_1 \text{ ou } t_2 \text{ ou } \ldots \text{ ou } t_m$$

où chacun des termes $t_i$ est une conjonction des $n$ variables $x_i$, certaines étant éventuellement sous leur forme complémentée

# Évaluation des expressions booléennes

L'évaluation des opérateurs `&&` pour ET et `||` pour OU est paresseuse:

- l'évaluation d'un `&&` s'arrête dès qu'on rencontre la valeur `false`
- l'évaluation d'un `||` s'arrête dès qu'on rencontre la valeur `true`

Il existe des opérateurs pour lesquels les deux opérandes sont toujours évalués (`&` pour ET et `|` pour OU)

L'opérateur `^` correspond à un OU disjonctif (XOR): si les deux R): si

# Expressions et chaînes

En Java:

- + est aussi l'opérateur de concaténation sur les chaînes de caractères:

  `"Deug" + " Mass" + " ou " + "Mias"`

  est la chaîne:

  *Deug Mass ou Mias*

- toute valeur en java a une représentation prédéfinie comme chaîne de caractères

  Dans le bout de programme:

```
int i=0;
char ch='A';
String st="i vaut "+i+" et ch vaut "+ch;
```

  la chaîne `st` est :

  *i vaut 0 et ch vaut A*

# Expressions, compléments

Une expression peut aussi contenir des appels de fonctions:

*Exemple:*
une affectation notée symboliquement:

$$d \leftarrow \sqrt{b^2 - 4ac}$$

dans la description d'un algorithme correspond en Java à l'instruction:

```
d = Math.sqrt(b*b - 4*a*c);
```

- Les fonctions et constantes mathématiques ont le préfixe `Math.`:

  `Math.sqrt`, `Math.cos`, `Math.sin`,
  `Math.log`, `Math.pow` . . .

  `Math.PI` et `Math.E` sont des constantes

  Les (valeurs renvoyées par les) fonctions mathématiques et les constantes mathématiques sont de type `double`

- Les fonctions ont éventuellement des paramètres. Exemples:
  - ◇ la valeur de `Math.sqrt(b*b - 4*a*c)` est du type `double` et correspond à la racine carrée de son paramètre qui est ici la valeur de l'expression
    ```
    b*b - 4*a*c
    ```
    `Math.sqrt` a un seul paramètre de type `double`
  - ◇ la valeur de `Math.pow(a,b)` est du type `double` et a la valeur de $a^b$.
    `pow` a deux arguments du type `double`
  - ◇ dans l'exemple de la classe `Puissance` la fonction de classe `Puissance` décrite précédemment (il s'agit donc `Puissance.puissance`)
    - renvoie une valeur de type `int`
    - a deux paramètres de type `int`

# Structure d'un "programme" Java

La structure générale simplifiée d'un programme Java est la suivante:

```
// importations
class MonProg{
   public static void main(String [ ] st){
   // liste de déclarations de variables
   // liste d'instructions
   }
}
```

- `MonProg` est le nom du programme (en fait de la classe): le fichier source a comme nom `MonProg.java`
- un fichier `MonProg.class` est créé si la compilation se passe sans erreur
- `main` contient le programme proprement dit, qui sera exécuté par ls commande `java MonProg`
  `main` **est un nom imposé**

- `import fr.jussieu.script.*;` est un exemple de clause d'importation qui permet d'utiliser les fonctions définies de `fr.jussieu.script`

# Structure d'un "programme" Java (suite)

*Remarques*:

- `public`, `static` et `void` dans la déclaration de `main` sont obligatoires
- Seront expliqués plus tard:
  - ◇ la signification `public`, `static` et `void`
  - ◇ la spécification `(String [ ] st)` permet de passer des paramètres au programme (on peut choisir un nom d'identificateur quelconque à la place de `st`)

```
import fr.jussieu.script.*;
class Hello2 {
  public static void main( String [ ] args) {
     Deug.println("Hello " + args[0] + " !");
     }
}
```

va produire:

```
--> java Hello2
Exception ... ArrayIndexOutOfBoundsException
      at Hello2.main(Hello2.java:3)
--> java Hello2 Claude Pierre
Hello Claude !
-->
```

  - ◇ MonProg peut également contenir d'autres déclarations de variables et de fonctions qui peuvent être utilisées ailleurs

# Structure d'un programme . . .

- *Liste de déclarations de variables* :
  On a déjà vu comment déclarer des variables:
  <Nom du type> <Nom de Variable> [= <Valeur >] ;
  *Remarque* : une déclaration n'a pas tout à fait le même sens pour les types références (c'est-à-dire non primitifs)
- Les instructions élémentaires sont de la forme :
  - ◇ *expression* ;
    Les plus (seules?) intéressantes sont celles qui réalisent des affectations:
    ```
    x = 3*(x+y)-5*(x+z/2);
    ```
  - ◇ d'autres instructions élémentaires permettent de réaliser des entrées-sorties par appels de fonctions (ou méthodes) de valeur `void` :
    ```
    Deug.println(x);
    ```

**Important** : les fonctions de type `void` ne renvoient pas de valeur (dans certains langages on parle de *procédures* ou *sous-programmes*). Les appels à ces fonctions ne peuvent être intégrés dans des expressions mais constituent de véritables instructions (il suffit de faire suivre un appel par ;)

## Blocs / listes d'instructions:

Plusieurs instructions peuvent être regroupées pour constituer un bloc délimité par les *accolades* { et }.

```
import fr.jussieu.script.*;
    ........
{ x = 3; y = x + z/2;
  Deug.println(y);
}
    ........
```

est un bloc constitué de 3 instructions.

Un bloc peut par ailleurs contenir des définitions de variables:

- une variable n'est visible que dans le bloc où elle est définie (et les éventuels sous-blocs qu'il contient), dans les instructions qui suivent sa définition
- JAVA interdit la réutilisation du nom d'une variable déjà visible dans le bloc

Bien que ce ne soit pas une nécessité en Java, les définitions de variables dans un bloc sont en général regroupées en tête de ce bloc

```
import fr.jussieu.script.*;
class Blocs{
   public static void main(String [ ] args){
    int n = 4; Deug.println(n);
       { int n = 5; int m = 6;
         Deug.println(n); Deug.println(m);
       }
    Deug.println(m);
    }
}
```

produit des erreurs à la compilation:

```
Blocs.java:5: Variable 'n' is already defined
                          in this method.
       { int n = 5;
             ^
Blocs.java:9: Undefined variable: m
    Deug.println(m);
                         ^
2 errors
```

*Remarque* : la définition d'une classe

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

constitue un bloc qui en tant que tel peut contenir la définition de variables qui peuvent porter différents qualificatifs (par exemple static ou final).

**Nous y reviendrons plus tard**, mais disons simplement que d'éventuelles variables qualifiées static en début de ce bloc constituent des variables globales visibles dans toutes les fonctions définies dans la classe.

```
import fr.jussieu.script.*;
class VariablesExternes{
 static int var = 100;
 public static void main( String [ ] args) {
  Deug.println(var);
  int var = 10; // définition de var locale
  Deug.println(var); // locale
  Deug.println(VariablesExternes.var); // globale
  VariablesExternes.var = 300;  // globale
  Deug.println(VariablesExternes.var); // globale
  Deug.println(var); // locale
 }
}
```

donne à l'exécution:

```
100
10
100
300
10
```

On note que l'identificateur var désigne deux variables (une variable globale et une variable locale): la définition de la variable locale (dans le main) masque la variable globale qu'on peut cependant encore désigner sous la forme VariablesExternes.var

On note aussi au passage que la définition des variables n'est pas nécessairement en début de bloc (la variable n'est visible que dans les instructions qui suivent sa définition).

## Affectation

On vient de voir la forme générale d'une affectation:

JAVA : <nomVariable> = <expression>;

Exemples:

| | |
|---|---|
| x <- 1.25 | x = 1.25; |
| i <- 5 | i = 5; |
| j <- 3*i*i+2*i+4 | j = 3*i*i+2*i+4; |
| x <- sin($\pi$/5) | x = Math.sin(Math.PI/5); |
| x <- (1+racine(5))/2 | x = (1+Math.sqrt(5))/2; |
| trouve <- (i<j) et (i>0) | trouve = (i<j) && (i>0); |

## Entrées / sorties

Pour *ce* cours, on peut utiliser les fonctions définies dans `fr.jussieu.script` pour *lire* des variables ou *afficher* des valeurs.

- Lire une variable: (en s'arrêtant sur le premier séparateur)
  - `Deug.readInt()`: est un appel à une fonction qui attend que l'utilisateur entre au clavier une valeur entière: la valeur retournée par cette fonction sera la valeur entière entrée.
    *Exemple:* `i = Deug.readInt();`
    est une instruction d'affectation qui affecte à la variable `i` (de type entier) la valeur entrée au clavier
  - `Deug.readChar()`: idem pour une valeur de type caractère
  - `Deug.readDouble()`: idem pour une valeur réelle
  - `Deug.readString()`: idem pour une chaîne de caractères ne contenant pas de séparateur
  - `Deug.readLine()`: lecture d'une ligne

- Afficher une valeur:

  `Deug.println(<Expression>);` : affichage suivi d'un caractère de fin de ligne

  `Deug.print(< Expression >);` : l'affichage n'est pas suivi d'un caractère de fin de ligne

  - `Deug.println(`*exp*`);` affiche sur l'écran une représentation de la valeur de l'expression $exp$ et passe à la ligne suivante:
    *Exemples:*

    `Deug.println(12+7);` affiche `19`

    `Deug.println(i);`

    ◇ si `i` est une variable de type entière valant $12$, affichera `12`

    ◇ si `i` est une variable de type double valant $12.4$, affichera `12.4`

    ◇ si `i` est une variable de type caractère représentant le caractère 'A', affichera `A`

    ◇ si `i` est une variable de type chaîne représentant *hello*, affichera `hello`

  - `Deug.println("valeur :("+i+","+j+")");` si `i` est une variable entière de valeur $1$ et `j` une variable réelle de valeur $2.3$ affichera

    ```
    valeur :(1,2.3)
    ```

# Exercices

1. Écrire un programme qui lit quatre entiers, affiche leur somme et leur moyenne réelle.
2. Écrire un programme qui lit une température en degré Celsius et affiche son équivalent en degré Fahrenheit.
   La formule est degré Fahrenheit=degré Celsius*1.8+32.
3. Écrire un programme qui calcule la somme entière et la moyenne réelle de quatre entiers
   - en lisant les 4 nombre avant de faire quoi que ce soit d'autre ;
   - en exploitant chaque nombre après l'avoir lu.

# Instructions conditionnelles

L'objectif des instructions de ce type est de permettre de sélectionner une séquence d'instructions sur la base de la satisfaction de certaines conditions

Ainsi par exemple:

- pour calculer la valeur absolue d'un nombre, il suffit de prendre l'opposé du nombre si celui-ci est négatif
- le nombre de jours du mois de février est $29$ ou $28$ selon que l'année est bissextile ou non
- on considère un système d'élection dans lequel, pour être élu, un candidat doit obtenir la majorité absolue des votants avec la contrainte supplémentaire au premier tour que le nombre de votants soit supérieur à 25%

# Si . . . alors . . .

On souhaite qu'une liste d'instructions ne soit exécutée que si une certaine *condition* est vraie

si *condition* alors
    *liste d'instructions*

où *condition* correspond à une expression booléenne.

**Problème**: pour lever toute ambiguïté, il faut délimiter clairement dans l'expression de l'algorithme la liste des instructions On peut par exemple

- encadrer cette suite d'instructions avec des mots particuliers (par exemple début et fin) ou des caractères particuliers
- marquer seulement la fin par un mot tel que finDeSi

**Ce qui est important est que l'écriture ne laisse subsister aucune ambiguïté**

## Exemple 1

calcul de la valeur absolue d'un nombre y

```
soit y un nombre réel
res <-- y
si y est négatif alors
    début res <-- (-y) fin
res contient le résultat
```

## Exemple 2

nombres de jours de février et de l'année

```
soit a une année
fevrier <-- 28
annee <-- 365
si a correspond à une année bissextile alors
    début annee <-- annee + 1
          fevrier <-- fevrier + 1
    fin
annee et fevrier contiennent finalement
          les nombres de jours recherchés
```

Traduction en Java

## **if** (***condition***) ***bloc***

Le bloc peut être réduit à une instruction unique (et les accolades peuvent alors être omises)

Le premier exemple donné se traduit ainsi en:

```
import fr.jussieu.script.*;
  class ValeurAbsolue{
    public static void main(String[ ] args) {
      float y, res;
      Deug.print("Donner un nombre reel : ");
      y = Deug.readFloat(); res = y;
      if (y < 0)
          res = -y; // une seule instruction
      Deug.print("la valeur absolue de ");
      Deug.println(y + " est " + res);
    }
 }
```

```
  --> java ValeurAbsolue
  Donner un nombre reel : -567.89
  la valeur absolue de -567.89 est 567.89
  --> java ValeurAbsolue
  Donner un nombre reel : -12345.789
  la valeur absolue de -12345.789 est 12345.789
```

Pour le second exemple on a d'abord mis en place la fonction testant si une année est bissextile:

```
/* fichier Bissextile.java */
class Bissextile {
  public static boolean bissextile(int annee) {
     boolean b;
     b = ((annee % 4) == 0)
          && ((annee % 100) != 0)
          || ((annee % 400) == 0);
     return b;
     }
}
```

Un fichier `Bissextile.class` a été construit en compilant le fichier précédent:

```
--> javac Bissextile.java
--> ls Bissextile*
Bissextile.class      Bissextile.java
-->
```

## Le programme principal:

```
/* fichier Jours.java */
import fr.jussieu.script.*;
class Jours{
  public static void main(String [ ] args) {
   int a, fevrier, annee;
   Deug.print("Donner une annee : ");
   a = Deug.readInt();
   if ( a < 0 )
     { Deug.println("annee incorrecte");
       Deug.exit();}
   fevrier = 28; annee = 365;
   if (Bissextile.bissextile(a))
      { fevrier ++; annee ++; }
   Deug.println("en " + a);
   Deug.print("   fevrier a "+fevrier+" jours");
   Deug.println(" et il y a "+annee+" jours");
  }
}
```

Après construction du fichier Jours.class
(commande javac Jours.java)

```
--> java Jours
Donner une annee : 2100
en 2100
 fevrier a 28 jours et il y a 365 jours
```

# Si . . . alors . . . sinon . . .

Ce type de construction vise à réaliser un traitement particulier si une certaine condition est vérifiée et un autre traitement si cette condition ne l'est pas

si *condition* alors
    *liste d'instructions 1*
sinon
    *liste d'instructions 2*

Là aussi l'écriture doit permettre de rendre non ambiguë la lecture. (utilisation de début/fin par exemple)

Exemple 3

calculer le maximum de deux nombres x et y

```
soit x, y et z des nombres
si x > y
   alors début z <-- x fin
   sinon début z <-- y fin
le résultat est z
```

Traduction en Java

**if** ***(condition)*** ***bloc1***
**else** ***bloc2***

Chacun des blocs peut être réduit à une instruction unique (et les accolades correspondantes omises).

**Cependant, pour éviter les ambiguïtés, il est préférable d'utiliser en toute circonstance des accolades pour constituer un bloc et ainsi délimiter les instructions à exécuter selon que la condition est vraie ou non.**

Le programme de maximum de deux nombres se traduit en:

```
import fr.jussieu.script.*;
class Maximum {
  public static void main( String [ ] args) {
    float x, y, z;
    Deug.print("entrer deux nombres reels : ");
    x = Deug.readFloat();
    y = Deug.readFloat();
    if (y < x) {z = x;}
    else {z = y;}
    Deug.println("le maximum est " + z);
  }
}
```

Le programme traitant les années bissextiles peut s'écrire:

```
import fr.jussieu.script.*;
class Jours2{
 public static void main(String [ ] args) {
  int a, fevrier, annee;
  Deug.print("Donner une annee : ");
  a = Deug.readInt();
  if ( a < 0 )
    { Deug.println("annee incorrecte");
      Deug.exit();}
  if (Bissextile.bissextile(a))
    { fevrier = 29 ; annee =366; }
  else
    { fevrier = 28 ; annee = 365; }
  Deug.println("en " + a);
  Deug.println(" fevrier a "+fevrier+" jours");
  Deug.println(" il y a "+annee+" jours");
  }
}
```

## Imbrication de conditionnelles

L'un quelconque des blocs d'instructions exécutées (voire les deux) peut contenir des instructions quelconques et en particulier des instructions conditionnelles

Cet usage nécessite de ne pas perdre de vue que

> **un else se rapporte toujours au dernier if qui n'a pas encore de else associé**

Dans de telles circonstances, afin de minimiser les risques d'erreur, il est recommandé de

- de structurer en bloc les différentes listes d'instructions
- de présenter le programme sous une forme indentée faisant apparaître l'association des if et des else

La forme la plus simple s'exprime en JAVA sous la forme suivante où
$\forall i$ (pour tout i), $\underline{condition_i}$ désigne une expression booléenne:

$$\begin{array}{l}
\underline{\text{if}}\ \underline{(condition_1)}\ \underline{bloc_1} \\
\underline{\text{else}}\ \underline{\text{if}}\ \underline{(condition_2)}\ \underline{bloc_2} \\
\qquad\qquad \ldots \\
\underline{\text{else}}\ \underline{\text{if}}\ \underline{(condition_n)}\ \underline{bloc_n} \\
\underline{\text{else}}\ \underline{bloc_{n+1}}
\end{array}$$

Dans une telle séquence,

- on exécute la séquence d'instructions $\underline{bloc_i}$ pour la plus petite valeur de $i$ pour laquelle $\underline{condition_i}$ est vraie (donc pour tout $j < i$, $\underline{condition_j}$ est fausse)
- on exécute $\underline{bloc_{n+1}}$ si toutes les conditions sont fausses

L'exemple suivant est une variation de l'exemple des années bissextiles. On souhaite distinguer différents types d'années par rapport à la propriété d'être bissextile:

```
import fr.jussieu.script.*;
class Jours3{
 public static void main(String [ ] args) {
   int a;
   Deug.print("Donner une annee : ");
   a = Deug.readInt();
   if ((a % 4) != 0)
      { Deug.println(a + " non bissextile");}
   else if ((a % 100) != 0)
      { Deug.println(a +
            " bissextile non seculaire");}
   else if ((a % 400) == 0)
      { Deug.println(a +
            " seculaire bissextile");}
  else { Deug.println(a +
             " seculaire non bissextile");}
  }
}
```

Ici on a systématiquement structuré les instructions en blocs (présence des accolades même avec une seule instruction)

La séquence suivante:

```
    ......
  a = Deug.readInt();
  x = 1;
  if (a >= 0 )
  if (a == 0) x = 2;
  else x = 3;
  Deug.println(x);
    ......
```

donne le résultat suivant:

| a | -1 | 0 | 1 |
|---|---|---|---|
| affichage | 1 | 2 | 3 |

La présentation ne change pas le comportement !

```
  a = Deug.readInt();
  x = 1;
  if (a >= 0 )
      if (a == 0) x = 2;
  else x = 3;
  Deug.println(x);
```

conduit au même résultat que précédemment.

Pour associer le else avec un if différent, il faut utiliser les accolades:

```
    ......
  a = Deug.readInt();
  x = 1;
  if (a >= 0)
      { if (a == 0) x = 2;}
  else x = 3;
  Deug.println(x);
    ......
```

donne

| a | -1 | 0 | 1 |
|---|---|---|---|
| affichage | 3 | 2 | 1 |

# Exercices

1. Ecrire un programme qui calcule le maximum de 2 nombres
2. Ecrire un programme qui calcule le maximum de 3 nombres
3. Ecrire un programme qui lit 2 entiers et affiche s'ils sont égaux ou pas
4. Ecrire un programme qui calcule le signe (i.e., -1, 0, ou 1) d'un produit de deux nombres entiers sans calculer le produit lui-même.

   Exemple: Le signe du produit de -4 et 7 est -1.
5. Ecrire un programme qui calcule le nombre et les solutions d'une équation du second degré
6. Ecrire un programme qui, après avoir lu un numéro de mois et une année, en affiche le nombre de jours (on supposera l'existence d'une fonction bissextile testant si une année est ou non bissextile)

7. Ecrire un programme qui, après avoir lu le nombre d'inscrits, le numéro de tour (1 ou 2), le nombre de votants et le nombre de suffrages obtenus de votants pour un candidat, détermine si celui-ci est ou non élu (on utilise la règle définie précédemment)
8. Ecrire un programme qui, après avoir lu un jour, un mois et une année calcule et affiche le jour suivant. Par exemple, après avoir lu
   - jour = 23, mois = 3, annee = 1988,
     il affichera 24/3/1988
   - jour = 28, mois = 2, annee = 2003,
     il affichera 1/3/2003
   - jour = 28, mois = 2, annee = 2004,
     il affichera 29/2/2004
   - jour = 31, mois = 12, annee = 2003,
     il affichera 1/1/2004

## Expressions conditionnelles

Il est fréquent de rencontrer des séquences telles que

`if (`*condition*`) x =` $expression_1$ `;`
`else x =` $expression_2$`;`

Il est possible de l'exprimer en JAVA sous la forme d'une expression particulière:

`x = (`*condition*`) ?` $expression_1$ `:` $expression_2$`;`

Ainsi la valeur absolue d'un nombre $x$ est obtenue par:

```
x = (x > 0) ? x : -x;
```

Dans de nombreux problèmes, on se trouve dans une situation dans laquelle

1. on évalue une expression
2. on sélectionne une séquence d'instruction particulière selon la valeur de l'expression

Il est courant de parler d'**aiguillage**

Nous nous contenterons de donner la syntaxe JAVA du mécanisme:

switch ( $expression$ ) {
    case $constante_1$ :
        . . .
    case $constante_{n1}$ :
        $listeInstructions_1$
    case $constante_2$ :
        . . .
    case $constante_{n2}$ :
        $listeInstructions_2$
    . . .
    case $constante_k$ :
        . . .
    case $constante_{nk}$ :
        $listeInstructions_k$
    default :
        $listeInstructions_{k+1}$
}

Il faut noter que:

1. les valeurs par rapport auxquelles la valeur de l'expression est testée sont des constantes
2. le principe de fonctionnement est qu'une fois que l'$expression$ contrôlant l'aiguillage a été évaluée, l'exécution se poursuit en séquence à partir du case correspondant à la valeur de l'expression. Pour sortir de l'aiguillage il est nécessaire d'utiliser l'instruction

   break;

   faute de quoi l'exécution se poursuivra en séquence

   Cette instruction termine, dans la plupart des applications, chacune des listes d'instructions (et est souvent omise dans la dernière)
3. default est l'étiquette par défaut (pour le cas où aucune valeur des cas ne soit satisfaisante)

## Les effets de l'omission des break;

Dans la séquence suivante, on a omis les break;:

```
switch(n) {
 case 2: case 4:
    Deug.println("pair");
 case 1: case 3:
    Deug.println("impair");
 default :
    Deug.println("bizarre");
}
```

Pour la valeur $1$ de $n$ on obtient:

```
impair
bizarre
```

et pour la valeur $2$:

```
pair
impair
bizarre
```

# Exemple

Jeu de dé: on gagne si la valeur obtenue est paire.

```
import fr.jussieu.script.*;
public class JeuDeDe {
  static int de(int valeur){
   int resultat;
   switch(valeur) {
      case 1: case 3: case 5: resultat=0; break;
      case 2: case 4: case 6: resultat=1; break;
      default: resultat = -1;
      }
   return resultat;
   }
   public static void main(String [ ] args){
    int valeur, res;
    Deug.print("Donner la valeur du dé : ");
    valeur = Deug.readInt();
    res = de(valeur);
    if (res < 0) Deug.println("Tricheur");
    else if (res == 0) Deug.println("Perdu");
    else Deug.println("Gagne");
   }
}
```

**Remarque** : dans ce qui suit (et dans les exemples et exercices dérivés) on ne considère pas les caractères accentués.

Dans l'exemple suivant, on lit un caractère et on imprime un message indiquant s'il s'agit:

- une voyelle majuscule (A E I O U Y)
- une consonne majuscule (les autres lettres majuscules)
- un caractère d'une autre nature

```
import fr.jussieu.script.*;
class Caracteres {
 public static void main( String [ ] args) {
  char c;
  Deug.print("Entrer un caractere : ");
  c =  Deug.readChar();
  if (c >= 'A' && c <= 'Z')
   { switch(c) {
     case 'A': case 'E': case 'I': case 'O' :
     case 'U': case'Y':
            Deug.println(c +
               " est une voyelle majuscule");
            break;
     default:
            Deug.println(c +
             " est une consonne majuscule");
     }  // pour le switch
   } // pour le if
   else { Deug.println(c +
             " n'est pas une lettre majuscule");
   }  // pour le else
 } // pour le main
} // pour la classe
```

# Exercice

Modifier l'exemple précédent de telle sorte qu'un message spécifique soit affiché selon que le caractère $c$ est:

- une voyelle minuscule (a, e, i, o, u ,y)
- une voyelle majuscule (A, E, I, O, U, Y)
- une consonne minuscule
- une consonne majuscule
- un chiffre
- un caractère d'une autre nature.

Les propriétés suivantes des caractères et de leur codage pourront être utiles:

- la différence de deux caractères donne leur distance. Ainsi `'D' - 'A'` a comme valeur 3.
- les codes des lettres majuscules ont des valeurs consécutives respectant l'ordre alphabétique et les codes des lettres minuscules ont des valeurs consécutives respectant l'ordre alphabétique. (le code d'une minuscule est plus grand que celui d'une majuscule). Il en résulte par exemple que
  Ainsi `'D' - 'd'` a même valeur que `'T' - 't'`.
- les codes des lettres majuscules ont des valeurs consécutives
- les codes des chiffres décimaux ont des valeurs consécutives et respectent l'ordre des chiffres.

# Fonctions

- Dans l'exemple du calcul de la puissance, la puissance est réalisée par une fonction:

  ```
  static int puissance(int x, int n)
  ```

  De cette façon:
  - la présentation du programme est meilleure: l'algorithme de calcul de la puissance est séparé de son utilisation (`main`)
  - on peut utiliser cette fonction `puissance` en dehors de cette classe (comme pour les fonctions de `Math`)

- En programmation objet et en Java on appelle en général une *fonction* une *méthode*.

# Exemple

Fichier `Puissance.java`:

```
import fr.jussieu.script.*;
class Puissance {
  static int puissance(int x, int n){
    int i, val = 1;
    for(i = 1; i <= n; i++)
         val = val * x;  // val *= x;
    return val;
  }
}
```

`puissance` est ici une méthode de classe de la classe `Puissance`

La commande `javac Puissance.java` générera un fichier `Puissance.class`

# Exemple (suite)

Fichier `Essai.java`:

```
import fr.jussieu.script.*;
class Essai {
 public static void main(String [ ] args){
  int nombre, exposant, res = 1;
  Deug.print("Donner un nombre entier ? ");
  nombre = Deug.readInt();
  Deug.print("Donner une puissance positive ou nulle ? ");
  exposant = Deug.readInt();
  if (exposant < 0){
      Deug.println("L'exposant doit être positif");
      Deug.exit();
      }
  res = 2*Puissance.puissance(nombre, exposant);
 }
}
```

Pour compiler et exécuter le programme:

```
--> javac Essai.java
--> java Essai
```

# Structure d'une méthode

Une méthode (fonction) se compose de:

- une en-tête de méthode qui précise le type de la valeur retournée par la fonction et le type et le nom de chacun des paramètres de cette fonction:
  `static int puissance(int x, int n)`

  - `puissance` est le nom de la fonction,
  - `int` est le type de la valeur retournée par cette fonction: un appel (une invocation) de `puissance` peut figurer dans une expression et sera interprété comme une valeur entière.
    Par Exemple:
    ```
    2*Puissance.puissance(nombre, exposant)
    ```
    est une expression de type entier.
  - `static` est un mot clé du langage (qui sera expliqué plus tard!)
  - `int x, int n` sont les deux paramètres de la fonction qui sont de types entiers.

- Un corps de méthode composé d'une liste de déclarations de variables et d'une liste d'instructions (un bloc)

*Remarque:* `main` est une méthode particulière.

# Appel/invocation de méthode

- A l'intérieur du corps de la méthode (fonction) les paramètres sont considérés comme des variables (locales)
  - qui ont le type de la déclaration des paramètres figurant dans l'en-tête de la fonction
  - qui seront initialisées au moment de l'appel de la méthode par les valeurs des expressions correspondantes:
    *Exemple:* Au moment de l'évaluation de

    ```
    res = 2*Puissance.puissance(nombre, exposant);
    ```

    - ◇ la *valeur* de la variable `nombre` du `main` initialise la variable `x` de `puissance`,
    - ◇ la *valeur* de la variable `exposant` du `main` initialise la variable `n` de `puissance`,
    - ◇ le contrôle d'exécution passe à la méthode `puissance`: les instructions de `puissance` sont exécutées jusqu'à l'instruction "`return val;`",
    - ◇ la *valeur* de la variable `val` de `puissance` est récupérée et multipliée par $2$, et le résultat est affectée à la variable `res` du `main`,
    - ◇ le `main` poursuit son exécution.

    Cette forme d'appel de méthode (fonction) s'appelle le **passage de paramètres par valeur**

- *(Pour l'instant)* Toutes les autres variables utilisées dans le corps de la méthode doivent être déclarées dans ce corps.

# Exercices:

1. Ecrire une méthode Java qui calcule le maximum de deux nombres
2. Ecrire une méthode Java qui calcule le maximum de trois nombres
3. Ecrire une méthode Java qu'on appellera `delta` qui, à partir des trois coefficients d'un trinôme du second degré à coefficients réels, calcule le discriminant de l'équation
4. Ecrire une méthode Java qui utilise la méthode `delta` définie précédemment pour déterminer le nombre de solutions d'une équation du second degré. Ecrire ensuite une méthode `main` qui lit les coefficients d'un trinôme du second degré et affiche le nombre de solutions réelles de ce trinôme.
5. Ecrire successivement des méthodes Java qui
   - ◇ renvoie la valeur true si un caractère donné $c$ est une lettre (minuscule ou majuscule)
   - ◇ renvoie la valeur true si la lettre transmise en paramètre est une majuscule et false si c'est une minuscule (on supposera que la fonction ne peut être appelée qu'avec une lettre);
   - ◇ on pourra imaginer toute autre méthode permettant de tester la nature d'un caractère.

6. Quelles sont les sorties du programme suivant:

```
import fr.deug.script.*;
class Main{
  public static void main(String [ ] st){
      int i = 10;
      int j =1;
      int k;
      k = i; i = j; j = k;
      Deug.println(" i="+i+" j="+j); } }
```

7. On crée maintenant une fonction et on obtient le programme:

```
import fr.deug.script.*;
class BadSwap{
 public static void badswap(int i, int j){
     int k;
     k = i; i = j; j = k;
 }
 public static void main(String [ ] st){
     int i = 10;
     int j = 1;
     badswap(i,j);
     Deug.println(" i="+i+" j="+j);
 }
}
```

Quelles sont les sorties de ce programme. Expliquez le résultat.