«Les ordinateur du futur ne pèseront pas moins d'une tonne et demi.»
Popular Mechanics, 1949.

1.) Commentaires.

Les commentaires de Java, comme ceux de la plupart des langages de programmation, n'apparaissent pas dans le programme exécutable. On peut donc ajouter autant de commentaires qu'on le souhaite sans craindre de gonfler la taille du code compilé. Java propose trois types de commentaires.

Le plus courant est `//`, qui débute un commentaire jusqu'à la fin de ligne :

```
// La culture c'est comme la confiture
// moins on en a, plus on l'étale.
System.out.println("La reine blanche comme lys");
System.out.println("Qui chantait à voix de sirène");
System.out.println("Berthe au grand pied, Bietris, Alis");
System.out.println("Haremburgis qui tint le Maine");
System.out.println("Et Jeanne la bonne Lorraine");
System.out.println("Qu'Anglais brûlèrent à Rouen...");
```

Lorsque les commentaires[1] sont trop longs, il est possible de les placer entre les symboles `/*` et `*/` :

```
/* ------------------------------------------------------------
                INSERTION DANS UN ARBRE BINAIRE
------------------------------------------------------------*/

public class ArbreB
{
...
}
```

Java propose un troisième type de commentaire utilisable pour la génération automatique de documentation (utilitaire JavaDoc). Ce type de commentaire commence par `/**` et se termine par `*/` :

```
/**
 * @(#)EX01.java   1.0 2000/01/28
 * Manipulation de l'environnement de développement
 * @see
 * @author     CAMOS-CNAM / LWH
 * @version    1.0
 */
```

Il faut absolument commenter les programmes car :

- Ils ne sont écrit qu'une seule fois
- Ils sont relus des dizaines de fois,
- Le commentaire est une re-formulation de la spécification du programme

Exemple de mauvais commentaire :

[1] Attention ! Contrairement à Caml, il n'est pas possible, en Java, d'imbriquer les commentaires.

```
int zoom=2 // zoom à 2
```

aucun renseignement sur le rôle de zoom et pourquoi 2.

Exemple de bon commentaire :

```
int zoom=2 // valeur par défaut du zoom au démarrage
```

2.) Types primitifs.

Java est un langage fortement typé. De plus, contrairement à Caml qui synthétise les types, Java demande que le type de chaque variable soit déclaré.

Les types primitifs sont aussi appelés types simples, standard, primaires, élémentaires ou de base.

Il existe huit types primitifs en Java.

2.1.) Les types primitifs numériques.

Il existe quatre types numériques entiers qui ont les caractéristiques suivantes :

`byte`	valeur codée sur 8 bits, de -2^7 à 2^7-1 (de -128 à +127)
`short`	valeur codée sur 16 bits, de -2^{15} à $2^{15}-1$ (de -36768 à +32767)
`int`	valeur codée sur 32 bits, de -2^{31} à $2^{31}-1$ (de - 2147483648 à +2147483647)
`long`	valeur codée sur 64 bits, de -2^{63} à $2^{63}-1$ (de -9223372036854775808 à +9223372036854775807)

Les constantes entières (par exemple "2121") sont le type int, à l'exception de celles suffixées par la lettre L (majuscule ou minuscule) qui sont de type long (par exemple "2121L").

Les entiers peuvent s'écrire en hexadécimal. Dans ce cas, ils seront préfixés par 0x. (par exemple "0x89A4" ou "0xCAFE"). Ils peuvent également s'écrire en octal. Dans ce cas, ils sont précédés de 0 (par exemple "01234" ou "077").

Qunad une constantes entière est affectée à une variable de type byte ou short, elle est automatiquement convertie dans le type de la variable sauf si la capacité d'acceuil de la variable est dépassée. Dans ce dernier cas, le compilateur renvoie un message d'erreur.

Les types numériques réels sont au nombre de deux :

`float`	valeur codée sur 32 bits.
`double`	valeur codée sur 64 bits.

Les constantes réelles sont par défaut de type double. Mais, si elles sont suffixées par la lettre F (majuscule ou minuscule) elles sont du type float.

Il est également possible de suffixé les constantes doubles par la lettre D.

Exemples de constantes réelles : "3.1", "6.", "0.123456", "-4E3", "-4E3D", "-4E3F" ...

Java réalise des conversion de type entre les différentes valeurs, à condition qu'il n'y ait pas de risque de perte d'informations. L'affectation d'une constante de type double à une variable de type float nécessite une conversion de type ou transtypage (*cast* en anglais).

2.2.) Le type primitif char.

Les caractères sont codés avec le standard Unicode[2]. Il est ainsi possible d'utiliser des caractères de n'importe quel alphabet (latin accentué, grec, cyrillique, arabe, hiragana etc...) ainsi que des symboles mathématiques et techniques. Les constantes de type caractère sont représentées entre deux apostrophes :

'a', 'A', '1', '0', 'ç', 'ù', '£', 'µ' etc...

Il existe très peu d'éditeurs de texte Unicode. C'est la raison pour laquelle Java résout cette difficulté en ayant recours aux séquences d'échappement Unicode qui n'utilisent que des caractères ASCII[3]. Pour un caractère Unicode représenté par les quatres chiffres hexadécimaux wxyz, la séquence d'échapement est \uwxyz.

Exemples :

Caractères	Séquence d'échappement Unicode
à	\u00E0
â	\u00E2
ç	\u00E7
è	\u00E8
é	\u00E9
ê	\u00EA
ë	\u00EB
î	\u00EE
ï	\u00EF
ô	\u00F4

2.3.) Le type primitif boolean

Le type boolean peut prendre deux valeurs : true et false. Il est employé dans des opérations de tests logiques utilisant les opérateurs relationnels supportés par Java.

Attention: le type boolean de Java ne peut être assimilée à un 0 ou 1 comme dans le cas des langages C ou C++.

[2]Le système Unicode code les caractères sur 2 octets car il est prévu pour gérer tous les caractères de toutes les langues écrites. Cela offre un jeu de 65536 caractères parmi lesquels environ 35000 sont utilisés actuellement. Cependant, bien qu'il soit théoriquement possible d'utiliser n'importe quel caractère dans une application Java, l'affichage d'un caractère dépend, en définitive, des possibilités de votre système d'exploitation et éventuellement de votre navigateur. Il n'est, par exemple, pas possible d'afficher du Kanji sur une machine équipée de la version US de Windows 95.

[3] Acronyme de "American Standard Code for Information Interchange". Code qui permet de restituer les lettres de l'alphabet romain, les chiffres et les principaux caractères spéciaux utilisés en informatique. Les caractères ASCII peuvent être sur 7 ou 8 bits.

3.) Variables.

3.1.) Déclaration.

Java, comme tous les langages fondés sur le Pascal, demande que le type de chaque variable soit déclaré. On déclare une variable en spécifiant d'abord son type puis son nom :

```
byte un_octet;
int un_entier;
long un_entier_long;
char un_caractère;
```

Chaque déclaration se termine par un point-virgule. Le point-virgule est nécessaire car une déclaration est une instruction Java complète.

Les règles à suivre pour un nom de variable sont :

- Un nom de variable doit commencer par une lettre.
- Cette lettre peut être suivie d'une séquence de lettre ou de chiffres.

Notez qu'en Java, le sens des termes "lettres" ou "chiffres" est beaucoup plus large que dans beaucoup d'autres langages. Une lettre est définie comme n'importe quel caractère Unicode représentant une lettre dans une langue quelconque. L'utilisateur français ou allemand peut donc très bien utiliser les caractères accentués. De même l'utilisateur russe peut utiliser des caractères cyrilliques.

3.2.) Valeurs par défaut.

Les variables de type primitif sont toujours initialisés. Le tableau suivant donne les valeurs par défaut :

Type de donnée	Valeur initiale
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0F
double	0.0D
char	\u0000
boolean	false

Cependant, c'est une bonne pratique de toujours initialiser soi-même les variables qu'on utilise.

3.3.) Initialisation.

Après avoir déclaré une variable, il est important de l'initialiser à l'aide d'une instruction d'affectation.

```
int ma_variable;    // Ceci est une déclaration
ma_variable = 5;    // Ceci est une affectation
```

Il est également possible d'initialiser la variable au moment de sa déclaration :

```
int ma_variable = 5;  // Déclaration et initilisation
```

Java permet de déclarer des variables n'importe où dans le code.

3.4.) Constantes nommées.

Les constantes nommées (aussi appelées constantes symboliques) sont déclarées avec le modificateur final :

```
final double PI =  3,14159265359; // définition du nombre PI
```

Elles ne peuvent plus être modifiées dans la suite du programme (il est donc important de les initialiser directement lors de leur définition). Par convention, les identificateurs de constantes sont entièrement en majuscules.

3.5.) Conversion de type.

Tableau des conversions sans pertes							
DE \ A	**byte**	**short**	**int**	**long**	**float**	**double**	**char**
byte	Oui	Oui	Oui	Oui	Oui	Oui	Oui
short		Oui	Oui	Oui	Oui	Oui	
int			Oui	Oui	Perte	Oui	
long				Oui	Perte	Perte	
float					Oui	Oui	
double						Oui	
char			Oui	Oui	Oui	Oui	Oui

Perte = Perte de précision.

Les autres conversions peuvent être effectuées sans perte d'information dans certaines conditions, généralement celles qui restreignent le type le plus large aux valeurs du type le plus étroit.

4.) Opérateurs.

4.1.) Opérateurs arithmétiques.

Opérateurs unaires			
Symbole	fonction	Champ d'appication	Exemples
+	plus unaire	Entiers et réels	+5
-	moins unaire	Entiers et réels	-5
++	pré ou post-incrémentation	Entiers et réels	y = x++; y = ++x;
--	pré ou post-décrémentation	Entiers et réels	y = x--; y = --x;
Opérateurs binaires			
Symbole	fonction	Champ d'appication	Exemples
+	addition	Entiers et réels	y = x + y;
-	soustraction	Entiers et réels	y = x - y;
*	multiplication	Entiers et réels	y = x * y;
/	division réelle	Réels	y = x / y;
/	division entière	Entiers	y = x / y;
%	reste de la division entière	Entiers	y = x % z;

4.2.) Opérateurs booléens.

Symbole	**Fonction**	**Exemples**
&	ET logique	a = b & c;
|	OU logique	a = b | c;
^	OU EXCLUSIF logique	a = b ^ c;
&&	ET LOGIQUE contitionnel	a = expr1 && expr2;
||	OU LOGIQUE conditionnel	a = expr1 || expr2;
!	négation unaire	a = !b;

4.3.) Opérateurs relationnels.

Symbole	**Fonction**	**Exemples**
>	supérieur	a = (b > c);
>=	supérieur ou égal	a = (b >= c);
<	inférieur	a = (b < c);
<=	inférieur ou égal	a = (b <= c);
==	égal	a = (b == c=;
!=	différent	a = (b != c);

4.4.) Opérateurs sur les bits.

Symbole	Fonction	Exemples
>>	décalage à droite signé	i>>4;
<<	décalage à gauche	i<<n;
>>>	décalage à droite non signé	i>>>2;
&	ET binaire	a = a & b;
|	OU binaire	a = a | b;
^	OU exclusif binaire	a = a ^ b;
~	complément à 1.	i = ~i;

```
i>>k; // decaler i vers la droite de k bits avec son signe
i<<k; // decaler i vers la gauche de k bits
i>>>k; // decaler i vers la droite de k bits sans signe
```

Exemples d'opérations sur les bits :

Expression	Représentation binaire
51966	1100 1010 1111 1110
21	0000 0000 0001 0101
51966 & 21	0000 0000 0001 0100
51966 | 21	1100 1010 1111 1111
51966 ^ 21	1100 1010 1110 1011
! 51966	0011 0101 0000 0001
21 << 2	
21 >> 2;	
21 >>> 2	

4.5.) Opérateurs d'affectation.

En plus de l'affectation classique (opérateur =), il existe d'autres opérateurs d'affectation. Par exemple :

Opération	Equivalence
a += b;	a = a + b;
a -= b;	a = a-b;
a *= b;	a = a * b;
a /= b;	a = a / b;
a %= b,	a = a % b;
a >>= b;	a = a >> b;
a <<= b;	a = a << b;
a &= b;	a = a & b;
a ^= b;	a = a ^ b;
a |= b;	a = a | b;

Il est possible de faire des affectations "en chaîne" telles que :

```
a=b=c=d=e=f=0;
```

4.6.) Opérateurs ternaire.

```
expression_test ? expression_1 : expression_2;
```

C'est une expression dont le résultat vaut expression_1 si expression_test est vrai et expression_2 sinon. Exemple :

```
a = (x > 10) ? "Supérieur" : "Inférieur";
```

4.7.) Opérateurs d'allocation : new.

Il s'agit de l'opérateur de création des objets et des tableaux. Exemples :

```
String s = new String("Rien ne m'est sûr que la chose incertaine...");
```

4.8.) Opérateurs instanceof.

Cet opérateur binaire prend un objet comme premier opérande et une classe comme second opérande. Le résultat vaut true si l'objet est une instance de la classe et false dans le cas contraire. Exemple :

```
String s = new("Et meure Pâris ou Hélène...");
if (s instanceof String)
{
   system.out.println("s est une chaîne");
}
```

Le petit programme suivant montre l'utilisation des opérateur mathématiques.

```
/**
 * Utilisation des opérateurs mathématiques
*/

import java.util.*;

public class MathOps {
  static void prt(String s) {
    System.out.println(s);
  }
  static void pInt(String s, int i) {
    prt(s + " = " + i);
  }
  static void pFlt(String s, float f) {
    prt(s + " = " + f);
  }
  public static void main(String[] args) {
    // Choix aléatoire de deux nombres entiers
    int i, j, k;
    // '%' limits maximum value to 99:
    j = (int)(Math.random() * 100);
    k = (int)(Math.random() * 100);
   // Opérations sur des entiers
   prt("Opérations sur les entiers j="+j+" et k="+k+".");
    i = j + k; pInt("j + k", i);
    i = j - k; pInt("j - k", i);
    i = k / j; pInt("k / j", i);
    i = k * j; pInt("k * j", i);
    i = k % j; pInt("k % j", i);
    j %= k; pInt("j %= k", j);
    // Choix aléatoire de deux nombres flottants
    float u,v,w;  // applies to doubles, too
    v = (float)(Math.random() * 100);
    w = (float)(Math.random() * 100);
   // Opérations sur les flottants
    prt("Opérations sur les flottants v="+v+" et w="+w+".");
    u = v + w; pFlt("v + w", u);
    u = v - w; pFlt("v - w", u);
    u = v * w; pFlt("v * w", u);
    u = v / w; pFlt("v / w", u);

    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
  }
}
```

5.) Instructions.

Le point virgule (;) marque la fin d'une instruction.

```
{
   int a = 1;
   int b = 2;
   a = a + b;
   b = b * a;
   a = b - a ; b = b * a;
   a =
        a + a;
}
```

Attention : le point virgule ne marque pas la fin de la ligne. Ainsi une instruction peut s'étendre sur plusieurs lignes ou une ligne peut abriter plusieurs instructions.

5.1.) Les blocs d'instructions

Comme nous allons le voir plus loin dans ce chapitre, il peut-être utile, selon les situations, de placer plusieurs instructions à un endroit précis. Or certaines constructions (instructions), n'accepte qu'une seule sous-instruction (comme, par exemple, pour le `if`). Pour palier à ce problème, une solution agréable à était utilisée (en fait, héritée de C) : on définit un bloc d'instruction comme étant une instruction.
Au point de vue de la syntaxe, un bloc d'instructions commence par une accolade ouvrante `{` et se termine par une accolade fermante `}`. Entre ces deux caractères, nous y trouvons les sous-instructions, qui sont séparées les unes des autres par des points-virgules. Le bloc peut contenir zéro (bloc vide), une (équivalent à une instruction unique) ou plusieurs sous-instructions.

```
{
String vers1 = "Gall, amant de la reine alla, tour magnanime";
String vers2 = "Galamment de l'Arène à la Tour Magne, à Nimes";
System.out.println(vers1);
System.out.println(vers2);
}
```

5.2.) La conditionnelle.

Syntaxiquement parlant, l'instruction conditionnelle s'introduit avec le mot clé `if` (attention, majuscules et minuscules sont différentes en Java : `IF` ou `If` ou bien encore `iF` sont des identificateurs différents). Il faut faire suivre cet identificateur d'un test (une expression booléenne) parenthésée, puis d'une instruction. Cette dernière sera exécutée uniquement si la valeur du test est vraie (`true`). Si ce n'est pas le cas, il est possible de rajouter le mot clé else suivi d'une instruction qui sera alors lancée.

```
{
   if (a == 0) b = 0; else
   {
        b = -5 / a;
        System.out.println("Le résultat est "+b);
   }
}
```

Notez bien que la partie associée à un résultat faux, est facultative. Autre détail important, même si votre test se réduit en la valeur d'une variable déclarée comme booléenne, il est obligatoire de la parenthèser.

5.3.) Les instructions répétitives.

Il existe quelques instructions qui permettent de mettre en place des boucles (ou des itérations, c'est la même chose) et d'en contrôler leurs exécutions. Trois styles de boucles sont utilisables, et deux instructions permettent de débrancher l'exécution à l'intérieur d'une boucle.

5.3.1.) LA BOUCLE *« POUR »*

La syntaxe est la suivante :

```
for (initialisation; expression booléenne; incrémentation)
   instruction
```

Exemples d'utilisation :

```
public class Essai {
static public void main(String argv[])
for(int i=1;i<9;i++) System.out.println("Mon " + i+" est une salade");
System.out.println("Mon tout est un célèbre écrivain anglais");
}
}
```

```
public class Compteur {
    public static void main(String argv[]){
    int somme=0;
    for(int indice=1;indice<=10;indice++)
    {
        somme += indice;
    }
    String str = "La somme est : ";
    System.out.println(str + somme);
    }
}
```

Il est possible de définir plusieurs variables dans une boucle "for" mais celles-ci doivent être du même type :

```
for (int i=1, j=i+10;i<5;i++,j=i*2)
{
    System.out.println("i= "+i+" j= "+j);
}
```

5.3.2.) LA BOUCLE *« TANT QUE »*

Cette instruction permet elle aussi de définir des boucles, mais la syntaxe diffère. Pour ceux qui se demandent pourquoi avoir plusieurs constructions possibles pour les boucles, je répondrais que selon les cas il y en a qui sont plus adaptées que les autres. La syntaxe du **while** est la suivante : l'instruction commence par le mot clé **while** suivie d'une expression booléenne parenthèsée et d'une instruction. L'expression sert à déterminer s'il faut encore effectuer une itération, ou bien passer à l'instruction suivant la boucle. L'instruction de la boucle, elle est exécutée à chaque étape de la boucle. Pour mieux voir les choses, reprenons l'exemple précédent (la somme des dix premiers entiers positifs), et réécrivons le avec le **while**. En voici le code.

```
public class Compteur {
    public static void main(String argv[]){
    int somme=0;
    int indice = 1;
    while(indice<=10)
    {
        somme += indice;
        indice ++;
    }
    String str = "La somme est : ";
    System.out.println(str + somme);
    }
}
```

Dans tous les cas, on retrouve l'initialisation de la variable **indice**, mais elle est située en dehors de la boucle, on retrouve aussi l'incrémentation, mais celle-ci est maintenant placée dans la partie des choses à faire à chaque tour de boucle. Le résultat lui reste strictement le même.

5.3.3.) LA BOUCLE DO WHILE

Cette instruction a un fonctionnement quasi identique à la précédente. La différence essentielle réside dans le fait que l'on commence par calculer l'expression correspondante au test ou bien que l'on commence à exécuter l'instruction. Au point de vue de la syntaxe, on introduit l'instruction par le mot clé **do** que l'on fait suivre d'une instruction (le corps de la boucle, qui sera la première chose qui sera lancée), puis du mot clé **while**, lui-même suivit d'une expression parenthèsée. Cette dernière constitue la condition de rebouclage (la valeur **false** assurant la fin de la boucle). A titre d'exemple voici encore un programme calculant la somme des dix premiers entiers positifs.

```
public class Compteur {
    public static void main(String argv[]){
    int somme=0;
    int indice = 1;
    do
    {
        somme += indice;
        indice ++;
    } while(indice<=10)
    String str = "La somme est : ";
    System.out.println(str + somme);
    }
}
```

5.3.4.) L'INSTRUCTION CONTINUE

L'instruction **continue** permet d'arrêter le traitement pour l'itération en cours, et d'en recommencer immédiatement une nouvelle. Passons tout de suite à un exemple, histoire de mieux comprendre les choses. Supposons que l'on ait besoin de calculer la somme des dix premiers entiers positifs, excepté pour l'entier 5. Un code Java possible est alors ...

```
public class Compteur {
    public static void main(String argv[]){
    int somme=0;
    int indice = 0;
    while(indice<10)
    {
         indice ++;
         if (indice == 5) continue;
         somme += indice;
    }
    String str = "La somme est : ";
    System.out.println(str + somme);
    }
}
```

Si vous exécutez ce code, le résultat affiché vaudra 50 (soit 55 moins la valeur du cinquième entier, soit 5). Il faut bien comprendre deux choses. Premièrement, quoi que l'on puisse mettre après le **continue** , ce code ne sera pas exécuté. Deuxièmement, un **`continue`** doit forcément être placé dans une branche d'un test conditionnel, sans quoi ca n'a pas de sens. Pour mieux comprendre ce dernier point, méditez sur le bout de programme suivant.

```
public class Baisse_des_impots {
    public static void main(String argv[]){
    int somme=0;
    int année = 1900;
    while(année<2500)
    {
         année ++;
         continue;
         // Le code suivant ne sera jamais exécuté.
         System.out.println("Les impôts baisseront en "+année);
    }
  }
}
```

On peut faire encore plus compliqué avec cette instruction. Elle peut permettre de reprendre l'exécution en un point précis du programme. De tels points d'un programme sont nommés des labels (ou étiquettes). Au point de vue de la syntaxe, on définit un label par un identificateur immédiatement suivit d'un deux-points (par exemple, `nom_de_label:`).

```
public class Continue {
public static void main(String argv[]){
int a, b=0, s=0;

label1: while(b<5)
       {
          a=0;   b++;
          while(a<5)
         {
           a++;
           if (a==3) continue label1;
           else s++;
         }
       }
System.out.println("Somme = " +s);
  }
}
```

5.3.5.) L'INSTRUCTION BREAK.

Cette instruction permet donc aussi de faire un débranchement au sein d'une boucle, mais contrairement à l'instruction précédente, l'exécution se poursuit non pas par une nouvelle itération, mais par l'instruction suivant la boucle. Elle sert donc purement et simplement à stopper la boucle de façon définitive. Le programme suivant illustre mes propos.

```
public class Break {
  public static void main(String argv[])
  {
        int somme=0, indice=1;
        while(indice<=10)
        {
          if (indice==5) break;
          somme += indice++;
        }
        System.out.println("Le résultat vaut : "+somme);
  }
}
```

Le résultat vaut bien évidement 10. Dernière remarque au sujet de cette instruction : on peut aussi interrompre une boucle et reprendre au label spécifié :

```
public class Break {
  public static void main(String argv[])
  {
        int somme=0, indice=1;
        etiquette1:
        while(indice<=10)
        {
          if (indice==5) break etiquette1;
          somme += indice++;
        }
        System.out.println("Le résultat vaut : "+somme);
  }
}
```

6.) Etude de la classe Number.

La classe Number se trouve dans le package java.lang.

Elle hérite de la classe java.lang.Object

Il s'agit d'une classe publique.

Il s'agit d'une classe abstraite qui n'est donc pas instanciable.

Cette classe abstraite est la super-classe de `Byte`, `Double`, `Float`, `Integer`, `Long`, et `Short`.

Constructeur
Number( ) Constructeur par défaut.

Methodes	
`byte`	**byteValue ( )** Renvoie la valeur du nombre comme un « byte ».
`abstract double`	**doubleValue ( )** Renvoie la valeur du nombre comme un « double ».
`abstract float`	**floatValue ( )** Renvoie la valeur du nombre comme un « float ».
`abstract int`	**intValue ( )** Renvoie la valeur du nombre comme un « int ».
`abstract long`	**longValue ( )** Renvoie la valeur du nombre comme un « long ».
`short`	**shortValue ( )** Renvoie la valeur du nombre comme un « short ».

7.) Etude de la classe Integer.

La classe Integer se trouve dans le package java.lang.

Elle hérite de la classe java.lang.Number

Il s'agit d'une classe publique.

C'est une classe finale. Elle n'est donc pas héritable.

La class Integer est une classe enveloppe (wrapped classe) qui permet de manipuler le type primitif int comme un objet. Un objet de type Integer contient un champ de type int.

Champs	
`static int`	**MAX_VALUE** La plus grande valeur pour le type int.
`static int`	**MIN_VALUE** La plus petite valeur pour le type int.

Constructeurs	
Integer(`int value`) Construit un nouvel objet Integer à partir d'un int.	
Integer(String `s`) Construit un nouvel objet Integer à partir d'une chaîne.	

Methodes	
`byte`	**byteValue()** Renvoie la valeur en tant que type « byte »
`int`	**compareTo(Integer `anotherInteger`)** Comparaison de deux entiers.
`int`	**compareTo(Object `o`)** Comparaison avec un autre objet..
`static Integer`	**decode(String `nm`)** Transforme une chaîne en un entier. Accepte le format décimal, hexadécimal et octal.
`double`	**doubleValue()** Renvoie la valeur en tant que type « double »
`boolean`	**equals(Object `obj`)** Comparaison avec un autre objet. Renvoie vrai si obj est un objet Integer ayant la même valeur.
`float`	**floatValue()** Renvoie la valeur en tant que type « float »
`int`	**hashCode()** Renvoie un hascode pour cet entier.
`int`	**intValue()** Renvoie la valeur en tant que type « int »
`long`	**longValue()** Renvoie la valeur en tant que type « long »
`static int`	**parseInt(String `s`)** Transforme l'argument en un entier signé en base 10.
`static int`	**parseInt(String `s, int radix`)** Transforme l'argument en un entier signé en base radix.
`short`	**shortValue()** Renvoie la valeur en tant que type « short »
`static String`	**toBinaryString(`int i`)** Retourne une chaîne représentant l'entier en base 2.
`static String`	**toHexString(`int i`)** Retourne une chaîne représentant l'entier en base 16.
`static String`	**toOctalString(`int i`)** Retourne une chaîne représentant l'entier en base 8.
`String`	**toString()** Retourne une chaîne représentant l'entier.
`static String`	**toString(`int i`)** Retourne une chaîne représentant l'entier passé en paramètre.
`static String`	**toString(`int i, int radix`)** Retourne une chaîne représentant l'entier dans la base spécifiée en second argument.
`static Integer`	**valueOf(String `s`)** Renvoie un nouvel objet Integer initialisé avec la valeur passée en argument.
`static Integer`	**valueOf(String `s, int radix`)** Renvoie un nouvel objet Integer initialisé avec la valeur passée en argument (dans une base donnée).

8.) Etude de la classe Float.

La classe Float se trouve dans le package java.lang.

Elle hérite de la classe java.lang.Number

Il s'agit d'une classe publique.

C'est une classe finale. Elle n'est donc pas héritable.

La class Float est une classe enveloppe (wrapped classe) qui permet de manipuler le type primitif float comme un objet. Un objet de type Float contient un champ dont le type est float.

Champs.	
`static float`	**MAX_VALUE** La plus grande valeur pour le type float.
`static float`	**MIN_VALUE** La plus petite valeur pour le type `float`.

Constructeurs	
Float(`double value`) Construit un nouvel objet Float à partir d'un type `double`.	
Float(float value) Construit un nouvel objet Float à partir d'un type `float`.	
Float(String s) Construit un nouvel objet Float à partir d'un type `String`.	

Methodes	
`byte`	**byteValue ( )** Returns the value of this Float as a byte (by casting to a byte).
`int`	**compareTo(Float anotherFloat)** copares two Floats numerically.
`int`	**compareTo(Object o)** Compares this Float to another Object.
`double`	**doubleValue()** Returns the double value of this `Float` object.
`boolean`	**equals(Object obj)** Compares this object against some other object.
`static int`	**floatToIntBits(float value)** Returns the bit represention of a single-float value.
`float`	**floatValue()** Returns the float value of this `Float` object.
`int`	**hashCode()** Returns a hashcode for this `Float` object.
`static float`	**intBitsToFloat(int bits)** Returns the single-float corresponding to a given bit represention.
`int`	**intValue()** Returns the integer value of this Float (by casting to an int).
`boolean`	**isInfinite()** Returns true if this Float value is infinitely large in magnitude.
`static boolean`	**isInfinite(float v)**

	Returns true if the specified number is infinitely large in magnitude.
`boolean`	**isNaN()** Returns true if this `Float` value is Not-a-Number (NaN).
`static boolean`	**isNaN(float v)** Returns true if the specified number is the special Not-a-Number (NaN) value.
`long`	**longValue()** Returns the long value of this Float (by casting to a long).
`static float`	**parseFloat(String s)** Returns a new float initialized to the value represented by the specified `String`, as performed by the `valueOf` method of class `Double`.
`short`	**shortValue()** Returns the value of this Float as a short (by casting to a short).
`String`	**toString()** Returns a String representation of this Float object.
`static String`	**toString(float f)** Returns a String representation for the specified float value.
`static Float`	**valueOf(String s)** Returns the floating point value represented by the specified String.

9.) Le standard Unicode.

Il est intéressant d'étudier le standard Unicode pour voir que la création d'un code de représentation des caractères n'est pas simplement technique mais également conceptuel et linguistique.

Les principales caractéristiques d'Unicode sont :

- la capacité d'encoder tous les caractères du monde ;
- le codage sur 16 bits ;
- un mécanisme d'extension (UTF-16) pour coder un million de caractères supplémentaires si nécessaire

Les alphabets traités sont :

- Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, modern Korean Hangul, Chinese/Japanese/Korean (CJK) ideographs. Et bientôt: Ethiopic, Canadian Syllabics, Cherokee, additional rare ideographs, Sinhala, Syriac, Burmese, Khmer, and Braille.
- Ponctuation, diacritiques, mathématique, technique, flèches, dingbats
- Signes diacritiques de modification de prononciation (n + `~' = ñ)
- 18000 codes en réserve

9.1.) Eléments de texte, caractères et glyphes

D'un point de vue conceptuel il n'est pas évident de définir ce qu'est un caractère. Un élément de texte peut être composé de plusieurs caractères. Par exemple, en espagnol le double l "ll" compte comme un seul élément, comme s'il s'agissait d'une lettre spéciale. Les concepteurs d'Unicode ont choisi de définit des éléments de code (caractères) plutôt que des éléments de texte.

- p.ex. l'élément de texte "ll" est donc traité comme deux codes: `l' + `l'
- chaque lettre majuscule et minuscule est un élément de code

D'autre part il faut distinguer le caractère (la valeur d'un code) et son affichage sur l'écran ou sur le papier. On a donc une notion de caractère abstrait, par exemple :

- "LATIN CHARACTER CAPITAL A"
- "BENGALI DIGIT 5"

et une notion de glyphe qui est la marque faite sur un écran ou sur papier pour représenter visuellement un caractère, par exemple

sont des glyphes qui représentent le caractère "LATIN CHARACTER CAPITAL A". Unicode ne definit pas les glyphes et ne spécifie donc pas la taille, forme, orientation des caractères sur l'écran.
Il existe également une notion de caractères composites (p.ex. â) qui est formé de

- une lettre de base (qui occupe un espace) "a"
- un ou plus marques (rendus sur le même espace) "^"

Unicode spécifie

- l'ordre des caractères pour créer un composite
- la résolution des ambiguïtés
- la décomposition des caractères précomposés
- "ü" peut être encodé par le code U+00FC[4] (un seul caractère de 16-bits)
- ou bien décomposé en U+0075 + U+0308 ("u"+"¨").
- l'encodage en un seul caractère assure la compatibilité avec le standard ISO-Latin-1.

9.2.) Définition des codes

Lors de l'attribution des codes aux caractères on a veillé à assurer l'inclusion de standards précédents (0 .. FF = Latin-1). De plus on a défini une notion de script qui est un système cohérent de caractères utilisés par plusieurs langues. Ceci afin d'éviter les duplications. Par exemple, le chinois, le japonais et le coréen utilisent tous le même script (nommé CJK) car ils ont plusieurs milliers de caractères en commun.

Comme on peut s'y attendre, un texte est une séquence de codes correspondant à l'ordre de frappe au clavier des caractères. Cependant toutes les langues ne s'écrivent pas dans la même direction, il existe donc des caractères spéciaux de changement de direction.

L'attribution des codes obéit aux principes suivants :

- Un nombre de 16 bits est assigné à chaque élément de code du standard. Ces nombres sont appelés les valeurs de code
- U+0041 = nombre hexadécimal 0041 = décimal 65 représente le caractère "A" .
- Chaque caractère reçoit un nom
  - U+0041 s'appelle "LATIN CAPITAL LETTER A."
  - U+0A1B s'appelle "GURMUKHI LETTER CHA."
- Des blocs de codes de taille variable sont alloués aux scripts en fonction de leur nombre de caractères. L'espace des codes est actuellement aloués selon la séquence suivante : [standard ASCII (Latin-1)] - [Greek] - [Cyrillic] - [Hebrew] - [Arabic] - [Indic] - [other scripts] - [symbols and punctuation] - [Hiragana] - [Katakana] - [Bopomofo] - [unified Han ideographs] - [modern Hangul] - [surrogate characters] - [reserved for private use (never used)] - [compatibility characters].
- L'ordre "alphabétique" à l'intérieur d'un script est si possible maintenu

[4] La notation U+dddd signifie qu'il faut lire le nombre dddd comme un code Unicode en base 16.

9.3.) Base de donnée de codes

Il existe une base de données de codes qui définit, outre le code et le nom de chaque caractère, d'autres attributs utiles pour le traitement des textes. Pour chaque caractère on a les quatorze champs :

0	Code value
1	Character Name.
2	General Category
3	Canonical Combining Classes
4	Bidirectional Category
5	Character Decomposition
6	Decimal digit value : lorsque le caractère représente un chiffre, p.ex. le "V" romain
7	Digit value
8	Numeric value
9	"mirrored" (pour l'écritures bidirectionnelles)
10	Unicode 1.0 Name
11	10646 Comment field
12	Upper case equivalent mapping
13	Lower case equivalent mapping
14	Title case equivalent mapping

Quelques exemples:

Code value	0041
Character Name.	LATIN CAPITAL LETTER A
General Category	Lu
Canonical Combining Classes	0
Bidirectional Category	L
Character Decomposition	
Decimal digit value	
Digit value	
Numeric value	
mirrored	N
Unicode 1.0 Name	
10646 Comment field	
Upper case equivalent mapping	
Lower case equivalent mapping	
Title case equivalent mapping	0061

Code value	005E
Character Name.	CIRCUMFLEX ACCENT
General Category	Sk
Canonical Combining Classes	0
Bidirectional Category	ON
Character Decomposition	<compat> 0020 0302
Decimal digit value	
Digit value	
Numeric value	
mirrored	N
Unicode 1.0 Name	SPACING CIRCUMFLEX
10646 Comment field	
Upper case equivalent mapping	
Lower case equivalent mapping	
Title case equivalent mapping	

9.4.) Formes d'encodage

Il faut faire la différence entre la définition des codes Unicode pour la représentation des caractères et la manière dont ces caractères sont stockés sur les supports physiques (p.ex. dans des fichiers). Il existe deux modes principaux d'encodage :

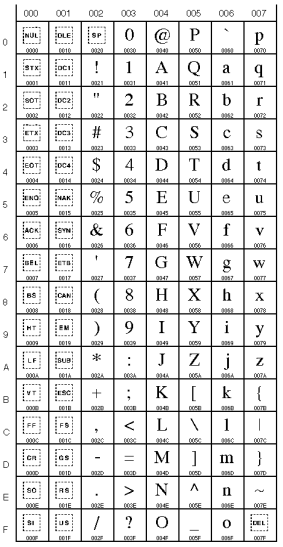
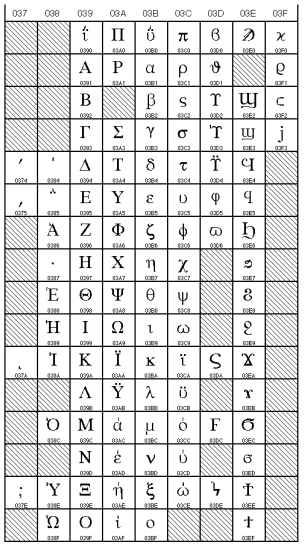
UTF-16
- caractères 16-bits
- paires de 2 x 16-bits pour extension

UTF-8
- codage à longueur variable
- 0x00 .. 0x7F ==> 1 byte (même codes que ASCII)
- 0x80 .. 0x3FF ==> 2 bytes
- 0x400 .. 0xD7FF, 0xE000 .. 0xFFFF ==> 3 bytes
- 0x10000 .. 0x10FFF ==> 4 bytes
- conversion sans perte vers et de UTF-16

9.5.) Le type caractère

Tout ce que nous venons de présenter montre que le type caractère n'est pas aussi simple qu'il en a l'air au premier abord. Il existe de nombreuses opérations, parfois complexes, sur ce type. Si l'on considère que les deux opérations de base sur les caractères sont l'encodage et le décodage (passer d'un caractère à l'entier correspondant et réciproquement), on peut énumérer d'autres opérations telles que :
- trouver l'équivalent majuscule d'un caractère
- trouver l'équivalent minuscule d'un caractère
- encoder un caractère sous forme UTF-8
- composer un caractère à partir d'un caractère de base et de marques
- etc.

Si l'on prend le standard Unicode, ces opérations nécessitent une consultation de la base de données des caractères ainsi que l'application de règles de compositions non triviales.

Latin de base (: [standard ASCII (Latin-1)])	
Cyrillique ([Cyrillic])	

10.) Exercices.

EXERCICE 2.1 :

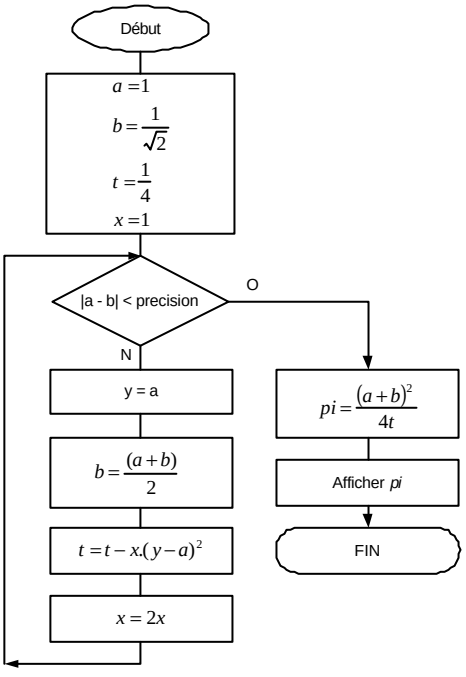
L'âge d'Yvonne, exprimé en années, est un multiple de 7, et si l'on inverse les deux chiffres qui le composent, cela la rajeunit de 18 ans. Ecrire un programme vous permettant de retrouver l'âge d'Yvonne.

```
public class Yvonne {
   public static void main(String argv[]){
        for (int i=3;i<15;i++) {
             /* L'âge d'Yvonne est un multiple de 7 */
             int age = i * 7;
             int prem = age / 10; /* premier chiffre */
             int second = age % 10; /* Deuxième chiffre */
             int inverse = second * 10 + prem;
             if ((age - inverse) == 18)
                  System.out.println("Yvonne a "+age+" ans.");
        }
   }
}
```

EXERCICE 2.2 :

Pi est le nom de la seizième lettre de l'alphabet grec, et est définit comme étant le quotient de la circonférence d'un cercle par son diamètre. L'appellation pi vient du grec *peripheria* qui désigne la circonférence d'un cercle. Ce nombre a de nombreuses propriétés qui ont fasciné les Grecs qui l'ont étudié.

Calculez pi en utilisant l'algorithme de Gauss-Legendre décrit ci-contre.



EXERCICE 2.3 :

Quels sont les résultats affichés par le programme suivant :

```
public class Mystère {
    public static void main(String argv[]){
    int a=0;
    int b=0;
    a = b ++;
    System.out.println("1) a = " + a + " et b = "+b);
    a = ++ b;
    System.out.println("2) a = " + a + " et b = "+b);
    }
}
```

EXERCICE 2.3 :

Calculez logarithme naturel de 2 à l'aide du développement suivant :

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \frac{1}{9} - \frac{1}{10} + \frac{1}{11} \ ...$$

```
public class Log2 {
   public static void main(String argv[]){
   int diviseur=0;
   char signe = '+';
   double result = 0;
   int itérations = 10000;
   while (diviseur < itérations) {
        diviseur ++;
        if (signe == '-') {
             signe = '+';
             result = result - (1.0/diviseur);
        } else {
             signe = '-';
             result = result + (1.0/ diviseur);
        }
   }
   System.out.println("Ln 2 = "+result);
   }
}
```

EXERCICE 2.3 :

Il est possible de calculer le quotient de 2 nombres entiers avec un nombre illimité de décimales. voici la marche à suivre:

Calculer et afficher la partie entière du quotient (devant la virgule) grâce à la division entière et traiter la partie décimale de la manière répétitive suivante:

- multiplier le reste de la division précédente par 1000(pour des tranches de 3 chiffres)
- calculer/afficher la partie entière du quotient

Ecrire un programme qui affiche le résultat du quotient $\frac{100}{93}$ avec environ 500 décimales.

```
public class Division {
   public static void main(String argv[]){
        double nb = 100;
        int    ent = 0;
        int    reste = 0;
        ent  =  (int)nb / 93;
        reste = (int)nb % 93;
        System.out.print(ent+",");
        for (int i=0;i<180;i++) {
             nb = reste * 1000;
             ent  =  (int)nb / 93;
             reste = (int)nb % 93;
             String s = Integer.toString(ent);
             if (ent < 100) s = "0"+s;
             if (ent < 10) s = "0"+s;
             System.out.print(s+" ");
        }
   }
}
```