Benoît Lemaire 2021
https://benoitlemaire.wordpress.com

Initiation à l'informatique et à l'algorithmique (LICENCE MIASHS 1)

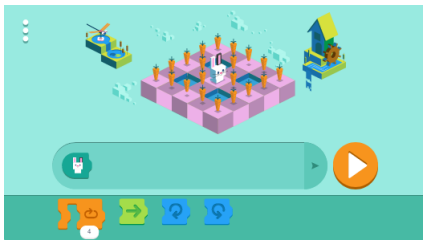
Ce document synthétise le cours dispensé à Grenoble en première année de licence MIASHS (Mathématiques et Informatique Appliquées aux Sciences Humaines et Sociales). Pour un bon apprentissage, le cours et ce document doivent être obligatoirement accompagnées d'un entraînement régulier devant l'ordinateur, par exemple en cherchant à programmer les problèmes proposés en TD. Le site web mentionné en entête contient les exercices de TD, de TP et d'examens des années précédentes.

1. Premiers éléments de programmation Java

Un programme informatique est une suite d'*instructions* que l'ordinateur va exécuter les unes après les autres, un peu comme une recette de cuisine. Il faut bien distinguer deux temps : la création du programme par le programmeur (vous !) et l'exécution du programme par l'ordinateur. C'est exactement comme pour les recettes de cuisine : le grand chef écrit la recette, et le commis suit la recette pas à pas. Vous allez être cette année le grand chef et vous utiliserez l'ordinateur pour exécuter vos programmes.

Évidemment, il faut indiquer les instructions à l'ordinateur dans un certain langage. Tant que le programme n'est pas syntaxiquement correct, l'ordinateur ne pourra pas exécuter le programme. Pour poursuivre l'analogie, le commis ne pourra pas réaliser la recette si elle n'est pas écrite dans le langage qu'il comprend. S'il est écrit « battre les œufs et marlier le poidure », le commis refusera de continuer. Il vous faudra passer du temps pour maîtriser la syntaxe du langage de programmation et écrire dans un langage compréhensible par l'ordinateur. Mais ce n'est pas parce que le programme est syntaxiquement correct que son exécution va correspondre à ce que vous vouliez faire. Vous pouvez avoir une recette de cuisine parfaitement écrite en français, mais dont la réalisation est catastrophique...

Pour bien comprendre cette distinction entre programme et exécution, je vous suggère de jouer avec ce doodle Google[1], un petit jeu en ligne dans lequel vous devez donner des instructions à un lapin pour qu'il mange toutes les carottes. Il ne faut pas faire déplacer le lapin



1 https://www.google.com/doodles/celebrating-50-years-of-kids-coding?doodle=32615474&domain_name=google.com&hl=fr

comme dans un jeu classique, mais trouver la bonne séquence d'opérations qui pourra être *exécutée* dans un second temps. Ca devient vite assez complexe...

Dans ce cours, nous allons étudier le langage Java, mais beaucoup de notions abordées sont les mêmes dans d'autres langages comme C, C++, Python, Perl, etc. Nous en profiterons aussi pour étudier comment l'ordinateur représente les nombres entiers, les négatifs, les réels, les textes. Mais nous allons tout de suite commencer par le langage Java pour que vous ayez rapidement la satisfaction de pouvoir écrire vos propres programmes. Nous ferons de temps en temps des digressions sur les concepts clés de l'informatique.

Un programme est donc une suite d'instructions. La plupart des programmes nécessitent des instructions d'*entrée-sortie (input/output* en anglais ou I/O), afin de pouvoir saisir des données au clavier (entrée) mais aussi afficher à l'écran les résultats du programme (sortie). On parle aussi d'instructions de *lecture* et *d'écriture*.

entrée-sortie

Écriture

En java, pour afficher une *chaîne de caractères*, c'est-à-dire une suite de caractères, on utilise l'instruction `System.out.println`. Par exemple :

Sortie ou écriture

```
System.out.println("Bonjour tout le monde !");
```

Notez bien le point-virgule nécessaire à la fin de chaque instruction, ainsi que les guillemets pour encadrer la chaîne. On peut donc écrire notre premier programme qui va afficher la table de multiplication de 5 :

```
System.out.println("1*5=5");
System.out.println("2*5=10");
System.out.println("3*5=15");
System.out.println("4*5=20");
System.out.println("5*5=25");
```

Ce programme ne fonctionne pas tel quel, il faudra lui donner un nom et dire que c'est le programme principal, mais nous verrons cela un peu plus tard.

Lecture

Pour aller plus loin, il faut permettre à l'utilisateur de taper (on dit saisir aussi) des données au clavier. Par exemple, on aimerait faire un programme qui affiche une table de multiplication en particulier, et pas toujours la table des 5. On utilise pour cela une instruction de *lecture*. Cette instruction va interrompre le programme et attendre que l'utilisateur ait saisi un nombre, une lettre, son nom, etc. Il va donc nous falloir *stocker* une valeur dans une *variable*. Idéalement, cette instruction devrait ressembler à quelque chose comme :

Entrée ou lecture

```
x = lireUneValeurAuClavier ;
```

Par exemple, pour demander à l'utilisateur de nous donner le rayon d'un cercle et afficher la surface de ce cercle, on aurait besoin de ce schéma (ce n'est pas du Java!) :

```
rayon = lireUneValeurAuClavier ;
afficher("votre cercle a une surface de" + π * rayon * 2)
```

Nous verrons plus tard pourquoi cette instruction de lecture est malheureusement un peu compliquée en Java. Il faut commencer par ajouter cette ligne au tout début du programme (on verra la signification plus tard) :

```
import java.util.Scanner;
```

Ensuite, on définit, *une fois seulement*, un Scanner qui va nous permettre de lire au clavier :

```
Scanner sc = new Scanner(System.in);
```

On peut alors lire notre chaîne de caractères, en la stockant dans une variable que nous appelons ici x :

```
x = sc.nextLine();
```

Pour lire un nombre entier et non pas des caractères, on remplace cette ligne par :

```
x = sc.nextInt();
```

Pour lire un flottant (un réel), on utilise :

```
x = sc.nextFloat();
```

ou, pour un flottant de grande capacité :

```
x = sc.nextDouble();
```

Premier programme Java

Nous pouvons maintenant écrire notre premier programme Java pour afficher le périmètre et la surface d'un cercle dont l'utilisateur nous donne le rayon.

```
import java.util.Scanner;
public class PremierProgramme {
   public static void main(String[] args) {
        double rayon,per,surf;
        Scanner s=new Scanner(System.in);
        System.out.println("Quel est le rayon de votre cercle ?");
        rayon=s.nextDouble();
        per=2*rayon*Math.PI;
        surf=rayon*rayon*Math.PI;
        System.out.println("Le perimetre est "+per+" et la surface est "+surf);
   }
}
```

La première ligne est nécessaire à la lecture depuis le clavier. Elle indique juste qu'il faut utiliser une *bibliothèque* (appelée parfois aussi *librairie* après mauvaise traduction de l'anglais) particulière, c'est-à-dire un autre programme déjà existant.

La seconde ligne définit une *classe*, que l'on a appelé ici PremierProgramme. En deuxième année, on verra qu'un programme peut être constitué de plusieurs classes. Cette année, nous n'utiliserons toujours qu'une seule classe. Attention à l'accolade après le nom de la classe et à l'accolade fermante à la fin. Entre les deux, vous pouvez voir qu'on *indente* les lignes, c'est-à-dire **qu'on les décale à droite à chaque nouvelle ouverture d'accolade**. C'est extrêmement important pour que le programme soit lisible. Prenez l'habitude dès maintenant d'indenter les programmes, il y aura des points en moins aux évaluations si vous indentez mal.

classe

indentation

La troisième ligne indique à l'ordinateur où est le début du programme. C'est un peu évident ici que le programme doit commencer au début, mais on verra par la suite que ce n'est pas toujours le cas. On indique donc la fonction principale qui s'appellera toujours *main* (principal en anglais). Les autres mots-clés de cette ligne seront explicités plus tard.

main

Ensuite, on indique que l'on va utiliser trois variables, `rayon`, `per` et `surf`, et que ce sont des nombres réels (indiqué par le mot-clé *double*). Nous allons détailler cela dans la prochaine partie. Viennent ensuite des instructions d'affichage à l'écran, de lecture du rayon, de calcul du périmètre et de la surface et l'affichage du résultat. Remarquez que la valeur π est connue de Java sous la forme `Math.PI`. Les prochaines pages vont être consacrées à bien comprendre toutes ces instructions.

2. La notion de variable

Une variable permet, comme en mathématiques, de représenter une information que l'on appelle une **valeur**. La variable a donc un **nom**. Il lui est aussi associé une place dans la mémoire de l'ordinateur. Mais une variable peut représenter des choses très différentes : l'âge d'un individu, son nom, la liste de tous ses numéros de téléphones favoris, etc. Java étant un langage *typé*, on va donc devoir indiquer à l'ordinateur le type de la variable pour qu'il puisse réserver la place adéquate dans sa mémoire. Une variable a donc un **type**.

variable

type

Avant d'utiliser une variable, on doit donc indiquer à l'ordinateur son nom et son type. On dit que l'on *déclare* la variable. On indique d'abord son type, puis son nom. C'est exactement ce que l'on a fait dans le programme précédent avec

déclaration

```
double rayon,per,surf;
```

on a dit que les trois variables étaient de type double. On aurait pu aussi écrire trois lignes :

```
double rayon;
double per;
double surf;
```

Le type double représente un nombre réel. Mais il en existe bien d'autres, qui occupent des espaces différents dans la mémoire de l'ordinateur : des entiers, des caractères, des chaînes de caractères, etc. Avant de les détailler, il nous faut comprendre comment l'ordinateur représente les informations dans sa mémoire.

3. Représentation des informations dans l'ordinateur

Le codage binaire

Dans un ordinateur, on a besoin de représenter différents types d'information, des images, des sons, des vidéos, des textes, mais elles se ramènent toutes à des nombres. Ainsi,

- les images sont découpées en pixels et la couleur de chaque pixel est codée par un nombre ;
- les sons sont échantillonnés, découpés tous les X millisecondes et l'information dans chaque tranche de temps est codée numériquement ;
- les vidéos sont des suites d'images, donc des suites de nombres ;
- les textes sont des suites de caractères, chacun représenté par un code numérique.

bit

Il faut donc représenter des nombres. Dans un ordinateur, l'unité minimale de représentation de l'information correspond à un fil sur lequel il y a du courant ou il n'y a pas de courant. On peut le représenter par oui/non, vrai/faux ou, plus généralement, par 0 ou 1. On appelle cela un bit (**bi**nary digi**t**). Un bit d'information peut permet de coder deux informations. Par exemple, on pourrait convenir que 0 représente « il ne pleut pas » et que 1 représente « il pleut ». Deux bits d'information permettent de coder 4 informations (codées 00, 01, 10, 11). Par exemple, on pourrait convenir que 00 représente mon premier frère, 01 représente ma sœur, 10 représente mon second frère et 11 représente moi-même. Mais il va falloir un codage plus complexe si on veut représenter des lettres, des couleurs, etc. Continuons... trois bits permettent de coder 2^3=8 informations (codées 000, 001, 010, 011, 100, 101, 110, 111).

octet

On groupe souvent les bits par 8 et on appelle cela un octet (*byte* en anglais). Cela permet de représenter 256 informations (codées 00000000, 00000001, 00000010, 00000011, 00000100, ..., 11111111) puisque 2^8=256. Un octet permet donc de représenter tous les symboles de l'alphabet occidental et ce codage qui a été fait dans les années 1960 et qui est encore utilisé aujourd'hui. Par exemple, le G est codé par 01000111 et le '+' est codé par 0101011. On reviendra sur ce codage plus tard.

 Attention, ne pas confondre bit (O ou 1) et *byte* (octet) !

byte

Ensuite, on a les différents multiples : kilo-octet (1Ko=1024 octets), méga-octet (1Mo=1024Ko), giga-octet (1Go=1024Mo), téra-octet (1To=1024Go), etc. On a utilisé 1024 plutôt que 1000 parce que ce nombre est une puissance de 2 et que cela est bien pratique dans le monde binaire. L'inconvénient est qu'un Go n'est pas exactement un milliard d'octets, mais 1024×1024×1024=1073741824 ! Vous pouvez vous en rendre compte en comparant la taille mémoire de votre ordinateur, exprimée en octets et en Go.

Comment maintenant représenter des nombres avec ce code composé uniquement de 0 et de 1 ? On pourrait représenter un nombre comme 28 par 28 fois le nombre 1, mais ce ne serait pas très efficace. Avant de voir quelles solutions les informaticiens ont imaginées, on peut étudier rapidement quelques systèmes de représentations des nombres.

Le codage des Babyloniens

Les Babyloniens ont créé, il y a près de 4000 ans, deux signes, le clou et le chevron. Jusqu'à 9, les chiffres étaient représentés par le nombre de clous correspondant :

1 : **I** ; 2 : **II** ; 3 : **III** ; 4 : **IIII** ; 5 : **IIIII**... 9 : **IIIIIIIII**

ensuite ils utilisaient l'autre symbole, le chevron, qui représente le 10 :

10 : <
11 : <**I**
12 : <**II**
13 : <**III**...
20 : <<
21 : <<**I**
22 : <<**II**...
46 : <<<<**IIIIII**
59 : <<<<<**IIIIIIIII**

ensuite, ils réutilisaient le symbole I pour représenter 60 :

60 : **I**
91 : **I**<<<**I**
137 : **II**<**IIIIIII** = 60*2 + 17

On est en base 60. Cette base a traversé les siècles et est encore utilisée chez nous : une heure vaut 60 minutes, un tour complet vaut 360° = 6 x 60°, etc.

L'inconvénient est qu'on n'a pas de moyen de distinguer le 1 et le 60. En fait, il manque le zéro qui permet d'indiquer les positions non occupées. Il sera imaginé plus tard par les indiens et transmis par les arabes. D'ailleurs, le mot chiffre vient de l'arabe *Sifr* (le vide). L'autre inconvénient est que cela produit un code dont la taille varie beaucoup au fur et à mesure qu'on progresse dans les nombres.

Bases 10, 2, 16...

Notre système est lui en base 10, avec un symbole différent pour chaque chiffre, mais le mécanisme est le même : on a 9 symboles plus le zéro. Quand on a épuisé les symboles, on met le premier symbole et le zéro et on recommence :

10, 11, 12, 13... 19, 20, 21...

On pourrait faire la même chose dans d'autres bases. Par exemple, en base 4, on n'aurait que 3 symboles plus le zéro : 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, 110, 111, 112...

Après 3, on a épuisé tous les chiffres, donc on marque un 1 qui représente 1 paquet de 3 unités et un 0 pour représenter 0 unités. Vous pouvez voir si vous avez compris en visionnant la vidéo qui décrit le système utilisé par les Shadocks qui comptent en base 4 : `https://www.youtube.com/watch?v=lP9PaDs2xgQ`

Pour éviter une confusion, on met des parenthèses autour des nombres qui ne sont pas en base 10. Cela permet de ne pas confondre 22 et $(22)_3$ qui représente 8 (c'est bien le 8^e nombre en base 3 : 1,2,10,11,12,20,21,22). Lorsque c'est évident qu'on travaille en base 2, on a l'habitude d'omettre cette règle. On écrira alors directement 10010101 par exemple plutôt que $(10010101)_2$

Pour passer de la base b à la base 10

Que représente par exemple $(345)_7$?
Le 5 de $(345)_7$ représente 5 unités, c'est facile.
Le 4 de $(345)_7$ représente 4 paquets de 7, donc 28 en base 10.
Le 3 de $(345)_7$ représente 3 paquets de 7 paquets de 7, donc $3\times7\times7$
et donc : $(345)_7 = 5\times7^0 + 4\times7^1 + 3\times7^2 = 5 + 28 + 147 = 180$
De manière générale :

$$(a_p\ a_{p-1}\ ...\ a_1\ a_0)_b = \sum_{i=0}^{i=p} a_i b^i$$

Pour passer de la base 10 à la base b

On décompose le nombre en paquets de b. Par exemple, écrivons 180 en base 7 :

180 : 7 = 25 et il reste 5. On sait donc que le chiffre de droite est un 5. Il faut maintenant écrire 25 en base 7. On recommence.
25 : 7 = 3 et il reste 4. On sait donc que le chiffre de droite est un 4. Il faut maintenant écrire 3 en base 7. Comme il est plus petit que 7, il s'écrit 3.
Résultat : $180 = (345)_7$
De manière générale, le nombre en base b d'un nombre en base 10 est la suite inversée des restes des divisions successives par b.

La base 2 est très utilisée en informatique, comme on l'a vu précédemment puisque l'information la plus petite est une information binaire. La base 2 permet aussi facilement de faire des additions. Il y a des circuits spéciaux qui permettent cela dans un ordinateur.

```
  00101011  (43)
+ 00001001  (9)
= 00110100  (52)
```

N'oubliez pas que, en base 2, 1+1=10 ! On écrit le zéro et on a une retenue.

hexadécimal

La base 16, appelée aussi hexadécimale, est aussi utilisée en informatique, parce qu'elle donne des nombres plus concis et parce qu'elle permet de passer très facilement à la base 2 puisque 16 est une puissance de 2. Il nous faut 16 symboles : aux 10 symboles de la base 10 on ajoute A, B, C, D, E, F. Comme précédemment, on compte 1, 2, 3... 9, A, B, C, D, E, F, 10 (un paquet de 16 et 0 unités), 11, 12... 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, 21.....FD, FE, FF, 100, 101... La base 16 est par exemple utilisée pour coder les couleurs dans certains logiciels[2].

Pour passer de la base 16 à la base 2

Puisque 16 est une puissance de 2, on peut passer très facilement de la base 16 à la base 2, en décomposant chaque symbole directement dans les quatre symboles de la base 2 correspondants. Par exemple : A84 s'écrit : 1010 1000 0100. De la même manière, 10 0110 0111 1100 0001 0110 1111 s'écrit aussi 267C16F.

 Attention à regrouper à partir du chiffre des unités ! Si vous regroupez en partant de la gauche, vous aurez un résultat erroné.

2 Voir par exemple : https://www.w3schools.com/colors/colors_rgb.asp

Représentation des nombres décimaux

Les nombres décimaux sont représentés en codant séparément la partie entière, puis la partie décimale. En base 10, le nombre 63,426 représente 6 dizaines d'unités, 3 unités, 4 dixièmes d'unités, 2 dixièmes de dixièmes d'unités, etc. Donc :

$63{,}426 = 6\times10^{1} + 3\times10^{0} + 4\times10^{-1} + 2\times10^{-2} + 6\times10^{-3}$.

C'est équivalent en base 2 : le premier chiffre de la partie décimale représente 2^{-1}, qui vaut 1/2. Le second chiffre représente 2^{-2}, qui vaut 1/4, etc. Donc :

13,5 s'écrit **1101,1** puisque le dernier 1 représente $1\times2^{-1} = 0{,}5$.

De même,

6,25 s'écrit 110,01 ($1\times2^{2} + 1\times2^{1} + 0\times2^{0} + 0\times2^{-1} + 1\times2^{-2}$).

Pour convertir la partie décimale, on cherche combien de fois il y a 1/2 dans cette partie décimale, combien de fois il y a 1/2 de 1/2 dans ce qui reste, etc. On va donc diviser la partie décimale par 1/2 mais cela revient à... la multiplier par 2 (diviser par 1/2 équivaut à multiplier par 2) !

0,25 × 2 = **0**,5. on marque **0** et il reste 0,5
0,5 × 2 = **1**,0. on marque **1** et il reste 0 !
Donc 0,25 = $(0,\mathbf{01})_2$

Autre exemple : comment s'écrit 11,421 en base 2 ?
0,421 × 2 = **0**,842. on marque **0** et il reste 0,842
0,842 × 2 = **1**,684. on marque **1** et il reste 0,684
0,684 × 2 = **1**,368. on marque **1** et il reste **0,368**
0,368 × 2 = **0**,736. on marque **0** et il reste 0,736
0,736 × 2 = **1**,472. on marque **1** et il reste 0,472
0,472 × 2 = **0**,944. on marque **0** et il reste **0,944**
etc.
donc **11,421** = $(\mathbf{1011,011010}...)_2$

C'est la même chose dans une autre base. Comment s'écrit 11,421 en base 8 ?
0,421 × 8 = 3,368. on marque **3** et il reste **0,368**
0,368 × 8 = 2,944. on marque **2** et il reste **0,944**
etc.
donc **11,421** = $(\mathbf{13,32}...)_8$

On remarque qu'on écrit trois fois plus de lignes en base 2 et que l'on retrouve les mêmes restes une fois sur 3. C'est normal puisque 2^3=8 !

 Certains nombres peuvent être infinis dans une base et finis dans une autre. Par exemple, 1/3 s'écrit 0,333333 en base 10 mais 0,1 en base 3 !

Représentation des entiers négatifs

La difficulté est de représenter des négatifs avec uniquement des chiffres puisqu'on n'a pas le symbole '-' : on n'a que des 0 et de 1 ! On utilise donc un codage : certains nombres vont servir à coder des négatifs et leur valeur habituelle ne pourra donc pas être représentée.

On aurait pu décider de représenter un bit pour le signe (par exemple, 1 pour négatif et 0 pour positif), mais cela aurait conduit à perdre systématiquement un chiffre binaire. De plus, on aurait eu deux représentations pour le zéro. On aurait aussi des difficultés pour faire des opérations. Par exemple si on représente -6 par **1**0000110 et 5 par **0**0000101, on aura une erreur si on fait une simple addition puisque le résultat sera 100001011, ce qui fait -11 !!!.

La méthode du complément vrai.

complément vrai

Il faut d'abord penser que, dans un ordinateur, on représente toujours les nombres dans un espace limité (8 bits, 16 bits, 32 bits, etc.). Quand on veut représenter un négatif, on verra qu'il faut toujours se poser la question du nombre de chiffres dont on dispose. Coder -7 sur 8 bits ne donne pas le même résultat que sur 16 bits.

Présentation intuitive en base 10. Pour bien comprendre la technique, on va supposer que nos nombres ont 4 chiffres et sont exprimés en base 10, et par la suite, vous pourrez oublier cette explication et on passera en base 2. Avec 4 chiffres en base 10, on peut représenter 10^4=10 000 nombres. On va représenter à peu près autant de positifs que de négatifs, donc on représentera les 10 000 nombres qui se trouvent entre -5000 et 4999. Mais on n'a pas de signe « - » , uniquement des chiffres ! L'astuce est de considérer que

$$-A = -A + 9999 + 1 - 10\,000 = (9999 - A + 1) - 10\,000$$

$9999 - A + 1$ est appelé le *complément vrai* de A et va donc être le nombre qui va coder $-A$. On pourra objecter qu'il reste à soustraire 10 000, mais cette dernière opération ne change absolument rien à nos 4 chiffres et on peut donc l'omettre.

Prenons un exemple, toujours en base 10 pour mieux comprendre. On veut représenter -2817 avec 4 chiffres. On calcule donc son complément vrai qui vaut $9999 - 2817 + 1 = 7183$. Le nombre -2817 est donc codé par 7183. De toutes façons, on

ne pouvait pas représenter 7183 puisque on ne représente que des nombres entre −5000 et 4999.

On peut vérifier que l'opération d'addition fonctionne toujours. Supposons que l'on veuille additionner 4324 et −2817, ce qui devrait nous donner 1507. Le nombre 4324 étant un positif, il est représenté par ses chiffres habituels : 4324. À l'inverse, −2817 est représenté par 7183 comme vu plus haut. Additionnons 4324 et 7183, ce qui nous donne 11 507. On néglige le 5e chiffre puisqu'on ne travaille que sur 4 chiffres et on trouve le bon résultat.

La méthode du complément vrai pour représenter les négatifs a donc trois avantages :

- elle préserve l'addition ;
- le complément vrai du complément vrai redonne le nombre initial (1234 ➡ 8766 ➡ 1234) ; le calcul du complément vrai permet donc de coder et de décoder comme on le verra dans un exemple plus loin ;
- elle n'empiète pas sur les positifs. En effet, on cherche à représenter autant de négatifs que de positifs, sans que les uns empiètent sur les autres ! Par exemple, avec 4 chiffres décimaux, on peut représenter 10 000 nombres (de 0 à 9999). On pourra donc représenter 5000 positifs (codés de 0 à 4999) et 5000 négatifs (codés de 5000 à 9999). 5001 représente −4999, 5002 représente −4998, 5003 représente −4997, ... 9998 représente −2 et 9999 représente −1.

 Attention, seuls les négatifs sont représentés en complément vrai !

Ce détour par la base 10 était nécessaire pour bien comprendre, mais cette technique n'est utilisée que pour la base 2. Dans cette base, on calcule d'abord le complément logique de la valeur absolue du nombre `xxxxxxx` à représenter (l'équivalent de 9999 − `xxxx`). Par exemple, si on travaille sur 8 bits, on va calculer `11111111 - xxxxxxxx`. C'est très facile, les 0 deviennent des 1 (1-0=1) et les 1 des 0 (1-1=0). Le complément logique de `0010100` est donc `1101011`. Il faut ensuite ajouter 1 pour le complément vrai. Le complément vrai de `0010100` est donc `1101011 + 1 = 1101100`.

 Il existe cependant une autre méthode pour calculer très rapidement le complément vrai en base 2. Il suffit de recopier en partant de la droite les zéros (s'ils existent) ainsi que le 1 qui suit et d'inverser tout le reste. Pour reprendre l'exemple précédent, le complément vrai de `0010100` s'écrit donc, en partant de la droite : les deux zéros, le 1 et tout le reste inversé : `1101100`.

Sur n bits, on peut représenter 2^n nombres. Pour pouvoir représenter à peu près autant de négatifs que de positifs, l'intervalle utilisé est $[-2^{n-1},2^{n-1}-1]$ **(intervalle à connaître par cœur)**. Par exemple, si l'on veut représenter des entiers sur 8 bits, on

peut représenter 256 valeurs (2^8) ; avec la représentation des négatifs en complément vrai, on peut donc représenter des nombres de -2^7 à 2^7-1, c'est-à-dire de -128 à 127.

Par exemple, pour trouver comment représenter -12 sur un octet, on écrit 12 en binaire sur un octet ce qui donne `00001100`. On prend le complément vrai ce qui donne `11110100` (on recopie les 2 zéros de droite, le premier 1 et on inverse le reste). Le nombre -12 s'écrit donc `11110100` sur 8 bits. On peut chercher quel nombre code -12 en convertissant en base 10 ce résultat : $(\texttt{11110100})_2 = 128+64+32+16+4 = 244$. C'est donc le nombre 244 qui code le nombre -12. Notez qu'il n'y a pas d'ambiguïté puisque de toutes façons on ne peut pas représenter 244 sur un octet lorsque l'on veut représenter des négatifs. On a dit plus haut qu'on ne peut représenter que des nombres de -128 à 127.

On peut remarquer que les négatifs ont le bit le plus à gauche à 1 (puisque dans la forme positive du nombre, il y avait un 0 que le complément vrai a transformé en 1).

Pour retrouver le nombre négatif de départ à partir de son codage, il suffit de reprendre le complément vrai ce qui nous donnera la valeur absolue du nombre initial. En effet, le complément vrai du complément vrai ramène au nombre de départ. Repartons de l'exemple précédent dont le code obtenu était `11110100`. Son complément vrai est `00001100` (on recopie les 2 zéros de droite, le premier 1 et on inverse le reste). Ce nombre correspond au nombre décimal : $2^2+2^3= 4 + 8 = 12$. Le nombre initial était donc -12.

Additions

Lorsque les négatifs sont représentés en complément vrai, on peut effectuer des additions directement. Supposons que le nombres soient représentés sur n bits. Le n^e bit (le plus à gauche) renseigne sur un *débordement* éventuel. En effet, si on ajoute deux positifs (dont les n^e bit sont des 0), on doit trouver un positif ; si ce n'est pas le cas parce que le résultat a son n^e bit à 1, on a un débordement. Même chose pour l'addition de deux négatifs (dont les n^e bit sont des 1) qui doit donner un négatif avec un n^e bit à 1. Un débordement signifie que le résultat ne peut pas être représenté avec le nombre de bits disponibles. C'est le cas par exemple si on ajoute 80 et 70, chacun sur 8 bits. Le résultat (150) est en dehors de l'intervalle de représentation sur 8 bits qui est [−128,127]. L'addition d'un positif et d'un négatif ne provoque jamais de débordement. Si le $n+1^e$ bit est quand même à 1 dans ce cas, c'est un *report*, qui sera oublié.

débordement

Exemple : 36+(−48) sur 10 bits.

report

```
36    = 0000100100
- 48  = 1111010000
       1111110100
```
(= −12 puisque son complément vrai = 0000001100 = 12).

Il n'y avait pas de débordement possible puisqu'on additionnait un négatif et un positif...

Exemple : -64+(-80) sur 8 bits

```
-64    = 11000000
-80    = 10110000
       1 01110000
```

Il y a un débordement : le 8e bit à zéro indique qu'on a un positif. Or, on a ajouté deux négatifs ! Donc on ne peut pas représenter -64-80 sur un octet. En effet, ce résultat (-144) n'est pas dans l'intervalle [-128,127] correspondant aux octets.

Pour résumer :

- s'il y a un n+1^{e} bit supplémentaire à gauche lors de l'addition, c'est un report, on l'ignore simplement ;
- si les deux bits les plus à gauche des nombres à additionner sont différents, on a donc un négatif et un positif et il n'y a pas de débordement possible.
- si les deux bits les plus à gauche des nombres à additionner sont identiques, le n^{e} bit du résultat doit être identique aussi. Sinon, il y a un débordement.
- Attention à ne pas confondre le n^{e} bit qui permet de repérer un débordement et le n+1^{e} bit qui est juste un report.

Représentation des réels

On change maintenant de type de nombres puisqu'on va s'intéresser aux réels, que l'on appelle communément des flottants. Les techniques de représentation sont très différentes. Par exemple, l'entier 3 et le réel 3 vont avoir des représentations complètement différentes. C'est la raison pour laquelle en Java il faut indiquer au préalable si la variable que l'on définit est un entier ou un flottant.

Le format est celui défini par la norme IEEE 754 qui est la plus utilisée pour la représentation des nombres réels. L'objectif est toujours le même : représenter, uniquement avec des chiffres, des nombres qui utilisent généralement d'autres symboles. En effet, pour représenter des nombres très grands ou très petits, on utilise généralement les puissances de 10. Par exemple : 3,2341 × 10^{54} ou −7,77661 × 10^{-23}. On doit donc indiquer deux nombres : la *mantisse* (par exemple −7,77661) et

mantisse

exposant

l'exposant (par exemple, −23). Dans l'ordinateur, c'est la même chose, excepté le fait que la représentation est binaire. Le nombre à représenter est réécrit sous la forme

$$1{,}\ldots \times 2^{\cdots}$$

ce qui nous donne une mantisse et un exposant (on verra comment plus loin). On a donc un ensemble de bits pour la mantisse et un ensemble de bits pour l'exposant. On réserve aussi un bit pour le signe : 1 pour les négatifs et 0 pour les positifs (contrairement aux entiers, on code le signe par un bit particulier pour les réels).

Par exemple, si pour représenter de tels nombres on décide d'utiliser 4 octets, c'est-à-dire 32 bits, on peut décider de réserver 1 bit pour le signe, 8 pour l'exposant et 23 bits pour la mantisse (dans cet ordre). Mais on pourrait aussi avoir 1 bit de signe, 11 bits d'exposant et 20 bits pour la mantisse. Les informaticiens qui inventent les langages décident de la meilleure représentation. En java, il y a deux types de réels :

- *float* : 4 octets : 1 bit de signe, 8 bits d'exposant, 23 bits de mantisse
- *double* : 8 octets : 1 bit de signe, 11 bits d'exposant, 52 bits de mantisse

Représentation de l'exposant. L'exposant est, lui, un entier. Il peut être positif ou négatif. Ce n'est pas la méthode du complément vrai qui a été retenue pour représenter cet entier, mais une autre méthode, celle de l'excédent. Dans une représentation à n bits, l'idée est de représenter tous les nombres en leur ajoutant la valeur 2^{n-1}-1, qu'ils soient positifs ou négatifs. En ajoutant cet excédent, le résultat est forcément positif et on le représente comme d'habitude. Par exemple, sur 8 bits, on ajoute 2^7-1 =127 $(\texttt{01111111})_2$ à tous les nombres qu'on veut représenter. Le nombre −50 s'écrit donc $-50 + 127 = 77 = (\texttt{01001101})_2$. Le nombre 10 s'écrit $10 + 127 = 137 = (\texttt{10001001})_2$. On remarque que, cette fois-ci, les négatifs ont un 0 sur le bit le plus à gauche et les positifs un 1. Il y a juste deux cas particuliers :

excédent

- l'exposant 2^n-1 (`111...111` en binaire) est utilisé pour coder, soit l'infini lorsque la mantisse est nulle, soit une erreur comme le résultat d'une division par 0, appelée NaN pour *Not A Number,* lorsque la mantisse est non nulle ;
- l'exposant 0 (`000...000` en binaire) est utilisé pour coder le 0 (qui ne peut pas s'écrire sous la forme $1{,}\ldots \times 2^{\cdots}$).

Ainsi, avec un *float* qui dispose de 8 bits pour l'exposant, on pourra donc représenter des exposants de -2^7-2 à 2^7-1, c'est-à-dire de −126 à +127. Si vous voulez représenter des nombres plus petits ou plus grands, il faudra choisir un *double* et non un *float.*

Représentation de la mantisse. La mantisse est composée des premiers chiffres de la représentation binaire pour laquelle un seul chiffre non nul se trouve à gauche de la virgule (c'est comme en base 10 : on n'écrit pas 61,4.1 x 10^{18}, mais 6,141 × 10^{19}). Or,

en binaire, ce chiffre non nul est forcément un 1 ! On n'a donc pas besoin de le représenter et on écrit donc uniquement les premiers chiffres de la partie décimale.

Prenons un exemple et essayons de représenter le nombre 23,75. Celui-ci s'écrit :

10111,11

On l'écrit sous la forme 1,.... $\times$ 2$^{\cdots}$, avec mantisse et exposant, en décalant la virgule vers la gauche de façon à n'avoir qu'un seul 1 à gauche de la virgule :

$1{,}011111 \times 2^4$

Il faut maintenant écrire le 4 en excédent (voir la partie précédente) sur les 8 bits dont on dispose pour l'exposant, ce qui donne 10000011. La représentation sur 32 bits du nombre 23,75 est donc 01000001101111100000000000000000 (1 bit de signe, 8 bits d'exposant et 23 bits de mantisse)

Si le nombre à représenter est plus petit que 1, il faut décaler la virgule vers la droite et on obtient un exposant négatif. Comme précédemment, il faut garder un seul 1 à gauche de la virgule. Par exemple, $0{,}001101 = 1{,}101 \times 2^{-3}$

La limite entre les bits réservés à la mantisse et ceux réservés à l'exposant détermine la nature de ce que l'on peut représenter. Si on décide d'allouer plus de bits pour l'exposant en réduisant ceux de la mantisse (par exemple 16 bits d'exposant et 15 bits pour la mantisse), on pourra représenter des nombres bien plus grands, mais au détriment de la précision, c'est-à-dire du nombre de chiffres après la virgule. A l'inverse, si on veut représenter les nombres avec une grande précision, on sera obligé de limiter la valeur maximale de ce que l'on peut représenter.

Représentation des textes

ASCII

Pour représenter des textes, c'est la même chose, l'ordinateur les transforme sous la forme de nombres. Depuis très longtemps, il existe un codage qui permet d'associer à chaque caractère un nombre. C'est le code ASCII (*American Standard Code for Information Interchange*). Au début, sept bits suffisait puisque cela permettait de coder toutes les lettres, les chiffres, les caractères de ponctuation, etc. Combien de caractères peut-on coder au fait avec 7 bits ?[3]

Voici ci-contre quelques exemples de codes : le A est codé 65, c'est-à-dire 1000001.

Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	_
48	0	64	@	80	P	96	`	112	p		

[3] Cent-vingt-huit = 2 puissance 7

Par la suite, on a étendu ce code à 8 bits parce qu'on avait d'autres caractères à coder (comme é, ≤ ou ù). On pouvait donc représenter 256 caractères.

On utilise maintenant un codage international qui permet de coder des caractères de toutes les langues, en n'utilisant pas toujours une taille fixe. Ce standard s'appelle Unicode.