20

ByteArray

Le codage binaire

Il faut revenir à l'origine de l'informatique afin de comprendre les raisons de l'existence de la notation binaire.

La plus petite unité de mesure en informatique est représentée par les chiffres 0 et 1 définissant un état au sein d'un circuit électrique :

- 0 : le circuit est fermé.
- 1 : le circuit est ouvert.

Les processeurs équipant les ordinateurs ne comprennent donc que la notation binaire. Rassurez-vous, bien que tout cela puisse paraître compliqué dans un premier temps, il s'agit simplement d'une notation différente pour exprimer des données.

Avant de s'intéresser au fonctionnement de la classe `ByteArray`, nous devons tout d'abord être à l'aise avec le concept de *base arithmétique*.

De par l'histoire, l'homme compte en base 10 car ce dernier possède 10 doigts, c'est ce que nous appelons la *base décimale*. Pour exprimer la notion de temps, nous utilisons une *base sexagésimale* composée de 60 symboles. Ainsi, lorsque 60 secondes se sont écoulées, nous ne passons pas à 61 secondes, nous ajoutons une nouvelle unité, c'est-à-dire une minute et revenons à 0 en nombre de secondes.

Nous utilisons dans la vie de tous les jours les 10 symboles suivant pour représenter les nombres :

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Au delà de 9 nous devons donc combiner les symboles précédents.

Pour cela, nous ajoutons une nouvelle unité, puis nous repartons de 0. A chaque déplacement sur la gauche, nous ajoutons une puissance de 10.

La figure 20-1 illustre comment décomposer un nombre en différents groupes de puissances de 10 :

Figure 20-1. Groupes de puissance de 10.

Notre système décimal fonctionne par puissance de 10, nous pouvons donc exprimer le nombre 750 de la manière suivante :

```
7 * 100 + 5 * 10 = 7 * 10^2 + 5 * 10^1
```

Contrairement à la notation décimale, la notation binaire autorise l'utilisation des symboles 0 et 1 seulement.

Le nombre 150 s'exprime en binaire de la manière suivante :

```
10010110
```

Nous allons apprendre à convertir la notation binaire en notation décimale. Pour cela, nous appliquons le même concept que pour la base décimale en travaillant cette fois par puissance de 2.

La figure 20-2 illustre comment décomposer un nombre exprimé sous forme binaire en différents groupes :

Figure 20-2. Groupes de puissance de 2.

Très jeune, nous avons appris à compter jusqu'à 10 en base décimale. Si la norme avait été l'utilisation de la base binaire, nous aurions mémorisé la colonne de droite du tableau suivant :

Base décimale	**Base binaire**
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Tableau 1. Notation binaire.

Afin de bien comprendre la notation binaire, il convient de s'attarder sur le concept de poids du bit. Une fois cette notion intégrée, nous introduirons la notion d'octet.

A retenir

- La notation binaire permet à l'origine de représenter un état au sein d'un circuit.
- Un processeur ne comprend que la notation binaire.
- Parmi les bases les plus courantes, nous comptons les bases décimales (10) et sexagésimales (60).
- La base 2 est appelée base binaire.

Position et poids du bit

Prenons le cas du nombre 150, que nous pouvons représenter de la même manière sous forme binaire :

```
10010110
```

Les symboles utilisés en notation binaire sont appelés *bit*, le terme provenant de l'Anglais *binary digit*.

A l'inverse de la base décimale, où chaque déplacement sur la gauche incrémente d'une puissance de 10. En notation binaire, chaque déplacement du bit sur la gauche, augmente d'une puissance de 2.

Nous exprimons la position de chaque bit composant l'expression sous forme de poids. Le bit de poids le plus fort (*most significant bit ou msb*) est toujours positionné l'extrême gauche, à l'inverse le bit le plus faible (*less significant bit ou lsb*) est positionné à l'extrême droite.

La figure 20-3 illustre l'emplacement du bit le plus fort :

Figure 20-3. Bit le plus fort.

La figure 20-4 illustre l'emplacement du bit le plus faible :

Figure 20-4. Bit le plus faible.

Plus le poids d'un bit est fort, plus sa modification provoque un changement important du nombre. En continuant avec le nombre 150 nous voyons que si nous passons le bit le plus fort à 0 nous soustrayons 2^7 (128) au nombre 150 :

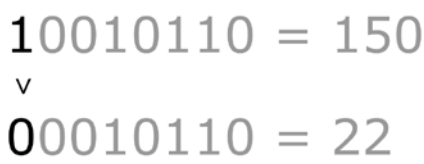


Figure 20-5. Modification du bit le plus fort.

A l'inverse, si nous modifions un bit de poids plus faible, la répercussion est moindre sur le nombre :

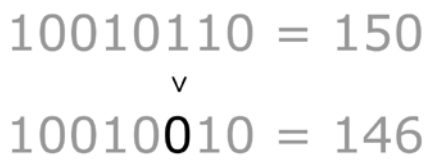


Figure 20-6. Modification d'un bit de poids plus faible.

Afin de convertir la notation binaire en notation décimale, nous multiplions chaque bit par son poids et additionnons le résultat :

$1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$

Soit, sous une forme plus compacte :

$2^7 + 2^4 + 2^2 + 2^1 = 150$

Nous retrouvons la notion de bits au sein des couleurs. Nous avons vu lors du chapitre 12 intitulé *Programmation Bitmap* que les couleurs étaient généralement codées en 8 bit, 16 bit ou 32 bit.

Une blague d'informaticiens consiste à dire qu'il existe 10 types de personnes dans le monde. Ceux qui comprennent le binaire et ceux qui ne le comprennent pas.

Nous savons que 10 en binaire correspond à 2^1 soit la valeur 2.

Vous comprendrez donc sans problème la blague inscrite sur ce fameux t-shirt illustré par la figure 20-7 :

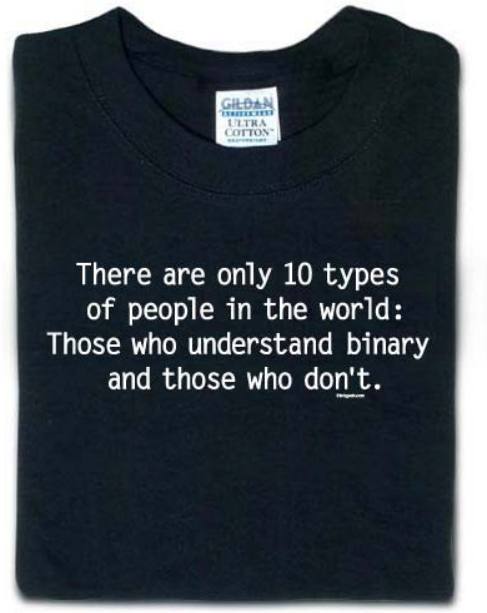


Figure 20-7. Humour et notation binaire.

Comme pour la base 2, d'autres bases existent comme la base 16 appelée plus couramment *base hexadécimale*. Celle-ci est généralement utilisée pour représenter les couleurs. Nous avons déjà abordée cette notation au cours du chapitre 12 intitulé *Programmation bitmap*. Contrairement à la notation binaire composée de 2 symboles, la base 16 est composée de 16 symboles.

Dans le code suivant nous évaluons la valeur 120 en base 16 :

```
var entier:int = 120;

// affiche : 78
trace ( entier.toString( 16 ) );
```

A l'inverse de la base 2, ou de la base 10, la base 16 utilise 16 symboles allant de 0 à F afin d'exprimer un nombre. Nous travaillons donc non plus en puissance de 2 ou 10 mais 16.

La figure 20-8 illustre l'idée :

Figure 20-8. Groupes de puissance de 16.

Le tableau suivant regroupe les symboles utilisés en base 16 :

Base hexadécimale	Base décimale
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Tableau 2. Notation hexadécimale.

La valeur 1A peut donc être exprimée de la manière suivante :

$1*16^1 + 10*16^0 = 26$

Lorsque nous travaillons avec un flux binaire, il peut être nécessaire d'exprimer un nombre sous différentes notations. Au lieu d'effectuer la conversion manuellement, nous pouvons utiliser la méthode

`toString` de la classe `Number`. Celle-ci accepte en paramètre la base dans laquelle convertir le nombre.

Dans le code suivant nous exprimons le nombre décimal 5 en notation binaire :

```
var entier:int = 5;

// affiche : 101
trace ( entier.toString( 2 ) );
```

En comptant de 1 à 10 en binaire nous retrouvons les valeurs du tableau 3 précédent :

```
var zero:int = 0;
var un:int = 1;
var deux:int = 2;
var trois:int = 3;
var quatre:int = 4;
var cinq:int = 5;
var six:int = 6;
var sept:int = 7;
var huit:int = 8;
var neuf:int = 9;
var dix:int = 10;

// affiche : 0
trace( zero.toString( 2 ) );

// affiche : 1
trace( un.toString( 2 ) );

// affiche : 10
trace( deux.toString( 2 ) );

// affiche : 11
trace( trois.toString( 2 ) );

// affiche : 100
trace( quatre.toString( 2 ) );

// affiche : 101
trace( cinq.toString( 2 ) );

// affiche : 110
trace( six.toString( 2 ) );

// affiche : 111
trace( sept.toString( 2 ) );

// affiche : 1000
trace( huit.toString( 2 ) );

// affiche : 1001
trace( neuf.toString( 2 ) );

// affiche : 1010
trace( dix.toString( 2 ) );
```

De la même manière, nous pouvons convertir un nombre en base décimale en notation hexadécimale :

```
var zero:int = 0;
var un:int = 1;
var deux:int = 2;
var trois:int = 3;
var quatre:int = 4;
var cinq:int = 5;
var six:int = 6;
var sept:int = 7;
var huit:int = 8;
var neuf:int = 9;
var dix:int = 10;
var onze:int = 11;
var douze:int = 12;
var treize:int = 13;
var quatorze:int = 14;
var quinze:int = 15;

// affiche : 0
trace( zero.toString( 16 ) );

// affiche : 1
trace( un.toString( 16 ) );

// affiche : 2
trace( deux.toString( 16 ) );

// affiche : 3
trace( trois.toString( 16 ) );

// affiche : 4
trace( quatre.toString( 16 ) );

// affiche : 5
trace( cinq.toString( 16 ) );

// affiche : 6
trace( six.toString( 16 ) );

// affiche : 7
trace( sept.toString( 16 ) );

// affiche : 8
trace( huit.toString( 16 ) );

// affiche : 9
trace( neuf.toString( 16 ) );

// affiche : a
trace( dix.toString( 16 ) );

// affiche : b
trace( onze.toString( 16 ) );

// affiche : c
trace( douze.toString( 16 ) );

// affiche : d
trace( treize.toString( 16 ) );
```

```
// affiche : e
trace( quatorze.toString( 16 ) );

// affiche : f
trace( quinze.toString( 16 ) );
```

Pour des raisons pratiques, les ingénieurs décidèrent alors de grouper les bits par paquets, c'est ainsi que naquit la notion d'octet.

A retenir

- On parle de poids du bit pour exprimer sa puissance.
- Le bit de poids le plus fort est toujours positionné à l'extrême gauche.
- Le bit de poids le plus faible est toujours positionné à l'extrême droite.
- La méthode `toString` de la classe `Number` permet d'exprimer un nombre dans une base différente.

Octet

L'octet permet d'exprimer une quantité de données. Nous l'utilisons tous les jours dans le monde de l'informatique pour indiquer par exemple le poids d'un fichier.

Bien entendu, nous ne dirons pas qu'un fichier MP3 pèse 3145728 octets mais plutôt 3 méga-octets. Nous ajoutons donc généralement un préfixe comme *kilo* ou *méga* permettant d'exprimer un volume d'octets :

- 1 kilooctet (ko) = 10^3 octets (1 000 octets)
- 1 mégaoctet (Mo) = 10^6 octets = 1 000 ko (1 000 000 octets)

Attention à ne pas confondre le terme de *bit* et *byte* :

1 octet = 8 bit

Un octet est donc composé de 8 bit :

```
11111111
```

Ce dernier peut contenir un entier naturel compris entre 0 et 255 lorsque celui-ci est dit *non-signé* (non négatif). Afin de convertir cette notation binaire sous forme décimale, nous utilisons la technique abordée précédemment.

Nous multiplions chaque bit par son poids et additionnons le résultat :

$$1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$$

Ce qui nous donne le résultat suivant :

```
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

Nous verrons qu'il est aussi possible d'exprimer au sein d'un octet une valeur oscillant entre -128 et 127 à l'aide d'un octet dit *signé* (négatif).

> Il est important de noter que dans un contexte de manipulation de données binaire la base 16 est très souvent utilisée au sein de logiciels tels les éditeurs hexadécimaux afin d'optimiser la représentation d'un octet.

Voici une liste non exhaustive de quelques éditeurs hexadécimaux gratuits ou payants :

- Free Hex Editor Neo (gratuit)
  http://www.hhdsoftware.com/Family/hex-editor.html
- HxD (gratuit)
  http://mh-nexus.de/hxd/
- Hex Workshop (payant)
  http://www.hexworkshop.com
- Hexprobe (payant)
  http://www.hexprobe.com/hexprobe

Il est plus facile de lire la valeur d'un octet sous la notation hexadécimale suivante :

```
4E
```

Que sous une notation binaire :

```
1001110
```

Un octet non signé d'une valeur de 255 peut donc être exprimé par deux symboles seulement en notation hexadécimale :

```
FF
```

En additionnant le poids de chaque symbole et en additionnant le résultat nous obtenons la valeur 255 :

$$15*16^1 + 15*16^0 = 240 + 15 = 255$$

Afin de mettre cette notion en pratique, nous avons ouvert une image au format PNG au sein d'un éditeur hexadécimal.

La figure 20-9 illustre une partie des données :

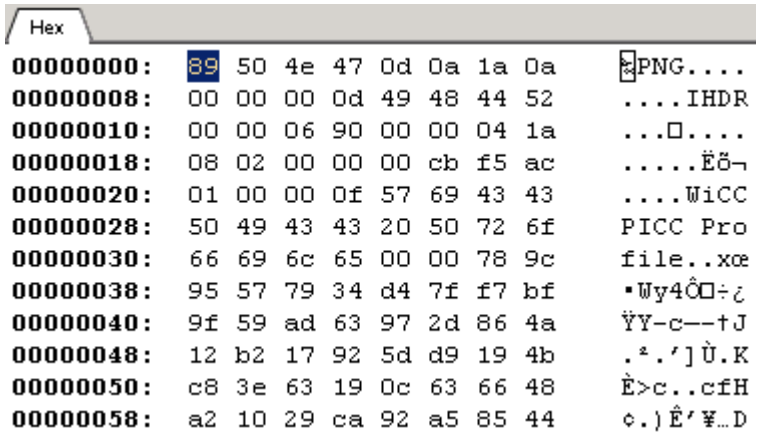


Figure 20-9. Editeur hexadécimal.

Nous pouvons remarquer que chaque colonne est composée de deux symboles représentant un octet. Comme son nom l'indique, l'éditeur hexadécimal représente chaque octet en base 16 à l'aide de deux symboles.

Pourquoi un tel choix ?

Pour des raisons pratiques, car le couple de symboles FF permet de représenter la valeur maximale d'un octet, ce qui est optimisé en termes d'espaces et moins pénible à lire et mémoriser.

Comme nous le verrons plus tard, chaque fichier peut être identifié par son entête. En lisant la spécification du format PNG disponible à l'adresse suivante : http://www.w3.org/TR/PNG/

Nous voyons que tout fichier PNG doit obligatoirement commencer par une signature composée de cette série de valeurs décimales :

```
137 80 78 71 13 10 26 10
```

En convertissant en notation hexadécimale chacune de ces groupes, nous retrouvons les mêmes valeurs dans notre fichier PNG :

```
var premierOctet:int = 137;
var deuxiemeOctet:int = 80;
var troisiemeOctet:int = 78;
var quatriemeOctet:int = 71;
var cinquiemeOctet:int = 13;
var sixiemeOctet:int = 10;
var septiemeOctet:int = 26;
var huitiemeOctet:int = 10;

// affiche : 89
trace( premierOctet.toString ( 16 ) );

// affiche : 50
trace( deuxiemeOctet.toString ( 16 ) );

// affiche : 4e
```

```
trace( troisiemeOctet.toString ( 16 ) );

// affiche : 47
trace( quatriemeOctet.toString ( 16 ) );

// affiche : d
trace( cinquiemeOctet.toString ( 16 ) );

// affiche : a
trace( sixiemeOctet.toString ( 16 ) );

// affiche : 1a
trace( septiemeOctet.toString ( 16 ) );

// affiche : a
trace( huitiemeOctet.toString ( 16 ) );
```

La figure 20-10 illustre la correspondance entre chaque octet :

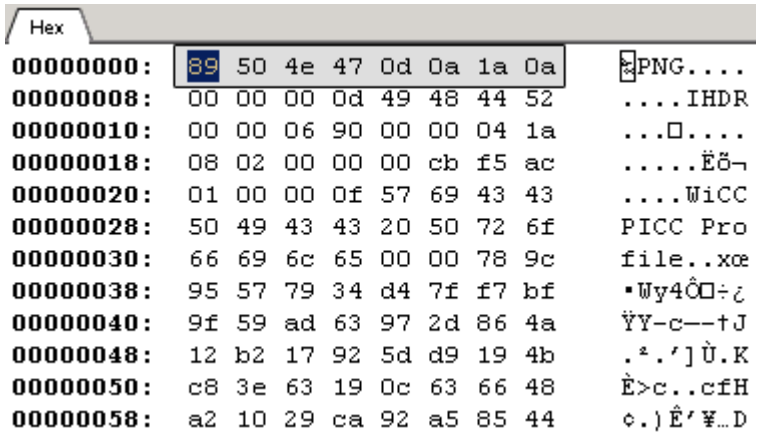


Figure 20-10 Signature d'un fichier PNG.

Nous allons nous intéresser à présent à l'ordre des octets.

A retenir

- Attention à ne pas confondre 1 octet (*byte*) et 1 bit.
- Un octet représente 8 bits.
- Un octet peut contenir une valeur maximale de 255 soit $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$.
- La notation hexadécimale est couramment utilisée pour représenter les octets.

Ordre des octets

Lorsque nous devons utiliser plusieurs octets afin d'exprimer un nombre, il est important de prendre en considération l'ordre de stockage. Cette notion est appelée en Anglais *byte ordering* ou *endian nature*.

Souvenez-vous, dans la partie intitulée *Position et poids du bit* nous avons vu que le bit le plus à gauche était considéré comme celui de poids le plus fort, et inversement le bit le plus à droite était celui de poids le plus faible.

Le même concept s'applique au sein d'un groupe d'octets. Nous parlons alors d'octet le plus fort (*most significant byte* ou *MSB*) et d'octet le plus faible (*less significant byte* ou *LSB*).

> Attention, notez que nous utilisons l'acronyme *MSB* et *LSB* en majuscule pour exprimer l'ordre des octets.
>
> A l'inverse nous utilisons des minuscules dans les acronymes *msb* et *lsb* pour exprimer le poids d'un bit.

Imaginons que nous devions stocker le nombre entier suivant :

```
550000000
```

Ce nombre est un entier non négatif 32 bits et nécessite 4 octets afin d'être stocké, sa valeur en hexadécimale est la suivante :

```
20 C8 55 80
```

Les processeurs tels les Motorola 68000 ou les processeurs *SPARC* équipant les plateformes Sun Microsystems utilise cet ordre de stockage et sont considérés comme *gros-boutiste* ou *big-endian* en Anglais.

Nous utilisons ce terme car l'octet de poids le plus fort (le plus gros) est à gauche de l'expression :

Gros-boutiste (big-endian)			
0	1	2	3
20	C8	55	80

Tableau 3. Stockage des octets gros-boutiste.

D'autres processeurs comme le fameux 5602 de *Motorola* ou x86 d'*Intel* sont *petit-boutiste* ou *little-endian* et stockent les octets dans le sens inverse :

Petit-boutiste (little-endian)			
0	1	2	3

80	55	C8	20

Tableau 4. Stockage des octets petit-boutiste.

Lors du développement d'applications ActionScript 3 utilisant la classe `ByteArray` nous devrons prendre en considération cet ordre, surtout dans un contexte de communication externe.

La notion d'ordre des octets n'a pas de réelle conséquence dans le cas de lecture d'un seul octet à la fois. A l'inverse, lorsque plusieurs octets sont interprétés, nous devons absolument prendre l'ordre des octets en considération.

A retenir

- Certaines architectures utilisent l'ordre petit-boutiste, d'autres l'ordre gros-boutiste.
- L'ordre des octets est important dans un contexte de communication entre plusieurs machines.
- Nous verrons que les classes `ByteArray`, `Socket` et `URLStream` possèdent une propriété `endian` permettant de spécifier l'ordre dans lequel stocker les octets.

La classe ByteArray

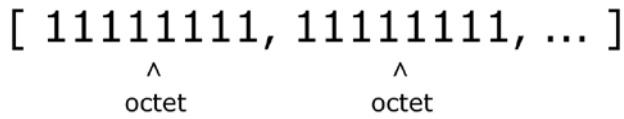
Il est temps de mettre en application toutes les notions abordées précédemment grâce à la classe `flash.utils.ByteArray`.

Même si la classe `ByteArray` figure parmi l'une des classes les plus puissantes du lecteur Flash, une des questions les plus courantes conçerne l'intérêt de pouvoir écrire un flux binaire.

L'intérêt de la classe `ByteArray` réside dans l'accès bas niveau au niveau des données. En d'autres termes, nous allons pouvoir manipuler des types de données non existants et travailler sur des octets. Cela peut nous permettre par exemple d'interpréter ou de générer n'importe quel type de fichiers en ActionScript 3.

Comme son nom l'indique, la classe `ByteArray` représente un tableau d'octets. Pour exprimer 1 kilo-octet nous utiliserons donc 1000 index du tableau.

La figure 20-11 illustre le fonctionnement d'un tableau d'octets :



Thibault Imbert
pratiqueactionscript3.bytearray.org

Figure 20-11. Tableau d'octets.

Comme son nom l'indique, la classe `ByteArray` possède quelques similitudes avec la classe `Array`. Les deux objets demeurent des tableaux et possèdent donc une longueur, mais chaque index composant un tableau d'octet ne peut être codé que sur 8 bits.

Voici une liste non exhaustive de quelques projets utilisant la classe `ByteArray` :

- FC64 : Emulateur Commodore 64.
  http://codeazur.com.br/stuff/fc64_final/
- AlivePDF : Librairie de génération de PDF.
  http://www.alivepdf.org
- FZip : Librairie de compression et décompression d'archives ZIP.
  http://codeazur.com.br/lab/fzip/
- GIF Player : Librairie de lecture de GIF animés.
  http://www.bytearray.org/?p=95
- GIF Player : Librairie d'encodage de GIF animés.
  http://www.bytearray.org/?p=93
- WiiFlash : Librairie de gestion de la manette Nintendo Wiimote.
  www.wiiflash.org
- Encodeurs d'images : Deux classes issues de la librairie corelib fournie par Adobe permettent l'encodage PNG et JPEG.
  http://code.google.com/p/as3corelib/
- Popforge Audio : Librairie de génération de sons.
  http://www.popforge.de/

Attention, les mêmes capacités que la classe `ByteArray` sont aussi présentes dans la classe `flash.net.Socket`. Grâce à celle-ci nous pouvons dialoguer avec l'extérieur au format brut binaire.

Méthodes de lecture et d'écriture

Afin de créer un flux binaire nous utilisons la classe `flash.utils.ByteArray`. Bien que nous puissions créer un tableau traditionnel à l'aide de l'écriture littérale suivante :

```
var monTableau:Array = [ ];
```

Seul le mot clé `new` permet la création d'un tableau d'octets :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();
```

Pour récupérer la longueur du tableau d'octets nous utilisons sa propriété `length` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();
```

```
// affiche : 0
trace ( fluxBinaire.length );
```

Lorsqu'un tableau d'octets est crée, celui-ci est vide, de la même manière qu'un tableau traditionnel.

> Notons qu'en ActionScript 3, la taille d'un tableau d'octets est dynamique et ne peut être fixe comme c'est le cas dans certains langages tels Java, C# ou autres.

Comme nous l'avons abordé précédemment, l'ordre des octets peut varier selon chaque plateforme. La propriété `endian` définie par la classe `ByteArray` permet de spécifier l'ordre des octets.

Par défaut, le tableau d'octets est *gros-boutiste* :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

// affiche : bigEndian
trace ( fluxBinaire.endian );
```

Afin de modifier l'ordre d'écriture des octets nous utiliser les constantes de la classe `flash.utils.Endian` :

- `Endian.BIG_ENDIAN` : octet le plus fort en première position.
- `Endian.LITTLE_ENDIAN` : octet le plus faible en première position.

Dans le code suivant, nous testons si l'ordre des octets est *gros-boutiste* :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

// affiche : true
trace( fluxBinaire.endian == Endian.BIG_ENDIAN );
```

Nous pouvons modifier l'ordre :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

// écriture dans l'ordre petit-boutiste
fluxBinaire.endian == Endian.LITTLE_ENDIAN;
```

Pour stocker l'entier non signé 5 au sein d'une instance de la classe `Array` nous pouvons écrire le code suivant :

```
var donnees:Array = new Array();

var nombre:uint = 5;

donnees[0] = nombre;

// affiche : 1
trace( donnees.length );
```

```
// affiche : 5
trace( donnees[0] );
```

Pour écrire le même nombre au sein d'un tableau d'octets, le nombre doit être séparé en groupe d'octets.

Ainsi, afin d'écrire un nombre entier non signé 32 bits comme c'est le cas pour le type `uint` nous devons séparer l'entier en 4 groupes de 8 bits :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture manuelle en représentation gros-boutiste
fluxBinaire[0] = (nombre & 0xFF000000) >> 24;
fluxBinaire[1] = (nombre & 0x00FF0000) >> 16;
fluxBinaire[2] = (nombre & 0x0000FF00) >> 8;
fluxBinaire[3] = nombre & 0xFF;
```

Notez que nous venons de stocker l'entier en représentation *gros-boutiste*. Une fois l'entier séparé, nous pouvons le reconstituer à l'aide des opérateurs de manipulation de bits :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture manuelle en représentation gros-boutiste
fluxBinaire[0] = (nombre & 0xFF000000) >> 24;
fluxBinaire[1] = (nombre & 0x00FF0000) >> 16;
fluxBinaire[2] = (nombre & 0x0000FF00) >> 8;
fluxBinaire[3] = nombre & 0xFF;

var nombreStocke:uint = fluxBinaire[0] << 24 |  fluxBinaire[1] << 16 |
fluxBinaire[2]  << 8 |  fluxBinaire[3];

// affiche : 5
trace( nombreStocke );
```

Si nous devions stocker les données au format *petit-boutiste* nous devrions inverser l'ordre d'écriture :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture manuelle en représentation petit-boutiste
fluxBinaire[3] = (nombre & 0xFF000000) >> 24;
fluxBinaire[2] = (nombre & 0x00FF0000) >> 16;
fluxBinaire[1] = (nombre & 0x0000FF00) >> 8;
fluxBinaire[0] = nombre & 0xFF;

var nombreStocke:uint = fluxBinaire[3] << 24 |  fluxBinaire[2] << 16 |
fluxBinaire[1]  << 8 |  fluxBinaire[0];
```

```
// affiche : 5
trace( nombreStocke );
```

Il s'avère extrêmement fastidieux d'écrire des données au sein d'un tableau d'octets à l'aide du code précédent.

Rassurez-vous, afin de nous faciliter la tâche le lecteur Flash va automatiquement gérer le découpage des octets ainsi que la reconstitution à l'aide de méthodes de lecture et d'écriture.

Ainsi, pour écrire un entier non signé au sein du tableau, nous utilisons la méthode `writeUnsignedInt` dont voici la signature :

```
public function writeUnsignedInt(value:int):void
```

Voici le détail du paramètre attendu :

- `value` : un entier non signé de 32 bits.

Le code fastidieux précédent peut donc être réécrit de la manière suivante :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);
```

Un entier non signé nécessite 32 bits, en testant la longueur du tableau binaire, nous voyons que 4 octets sont inscrits dans le tableau :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

// affiche : 4
trace( fluxBinaire.length );
```

Afin de lire l'entier non signé écrit, nous utilisons simplement la méthode `readUnsignedInt` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

// lève une erreur à l'exécution
var nombreStocke:uint = fluxBinaire.readUnsignedInt();
```

En testant le code précédent, l'erreur à l'exécution suivante est levée :

```
Error: Error #2030: Fin de fichier détectée.
```

Lors de l'écriture du flux de données, un pointeur interne défini par la propriété `position` de l'objet `ByteArray` se déplace automatiquement auprès de l'octet suivant :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// affiche : 0
trace( fluxBinaire.position );

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

// affiche : 4
trace( fluxBinaire.position );
```

La propriété `position` nous indique que le pointeur est à l'index 4 du tableau d'octets. Ainsi, la prochaine méthode d'écriture commencera à écrire les données à partir de cet index.

La figure 20-12 illustre le principe :

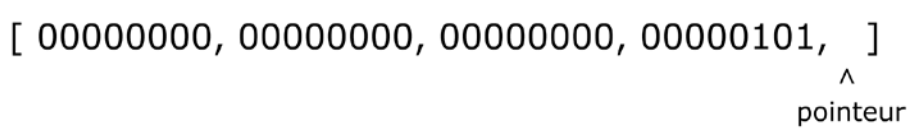


Figure 20-12. Positionnement du pointeur interne.

Afin de gérer les éventuelles erreurs de lecture du flux, nous pouvons placer les appels aux différentes méthodes de lecture au sein d'un bloc `try catch` :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

try

{

    // lève une erreur à l'exécution
    trace( fluxBinaire.readUnsignedInt() );

} catch ( pErreur:Error )
{

    trace ("erreur de lecture");

}
```

Pour pouvoir lire le flux, nous devons obligatoirement remettre à zéro le pointeur interne puis appeler la méthode de lecture :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStocke:uint = fluxBinaire.readUnsignedInt();

// affiche : 5
trace( nombreStocke );
```

Nous allons nous attarder sur un détail important.

Dans le code précédent, nous avons à nouveau utilisé le type `uint` afin de stocker la variable `nombre` ainsi que le résultat de la méthode `readUnsignedInt`. Tout au long de l'ouvrage, nous avons préféré l'utilisation du type `int` qui s'avère dans la plupart des cas plus rapide que le type `uint`.

Dans un contexte de manipulation de flux binaire, il convient de toujours utiliser le type lié à la méthode de lecture ou d'écriture. Nous utilisons donc le type `uint` lors de l'utilisation des méthodes `writeUnsignedInt` et `readUnsignedInt`.

Si nous ne respectons pas cette précaution, nous risquons d'obtenir des valeurs erronées. Afin de confirmer cela, nous pouvons prendre l'exemple suivant.

Dans le code suivant, nous inscrivons au sein du tableau d'octets un entier non signé d'une valeur de 3 000 000 000. En stockant le retour de la méthode `readUnsignedInt` au sein d'une variable de type `int`, nous obtenons un résultat erroné :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 3000000000;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStocke:int = fluxBinaire.readUnsignedInt();

// la machine virtuelle conserve le type à l'exécution
```

```
// affiche : -1294967296
trace( nombreStocke );
```

L'entier retourné par la méthode `readUnsignedInt` n'a pu être stocké correctement car le type `int` ne peut accueillir un entier supérieur à 2 147 483 647.

En utilisant le type approprié, nous récupérons correctement l'entier stocké :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 3000000000;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStocke:uint = fluxBinaire.readUnsignedInt();

// la machine virtuelle conserve le type à l'exécution
// affiche : 3000000000
trace( nombreStocke );
```

En analysant la figure 20-13 ci dessous, nous voyons que le nombre entier non signé d'une valeur de 3 000 000 000 occupe les 4 octets (32 bit) alloués par la machine virtuelle pour le type `uint` :

[0xB2, 0xD0, 0x5E, 0x00]

Figure 20-13. Entier non signé au sein du tableau d'octets.

Avez-vous remarqué que la figure précédente utilisait la notation hexadécimale afin de simplifier la représentation d'un tel nombre ?

> Nous utiliserons désormais la notation hexadécimale afin de simplifier la représentation des données.

Nous pourrions donc exprimer une telle valeur à l'aide de la notation hexadécimale :

```
var nombre:uint = 0xB2D05E00;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);
```

A l'inverse, si nous stockons un entier non signé d'une valeur de 5 à l'aide de la méthode `writeUnsignedInt` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();
```

```
var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStocke:uint = fluxBinaire.readUnsignedInt();

// affiche : 5
trace( nombreStocke );
```

Comme l'illustre la figure 20-14, seuls les 8 bits de poids le plus faible suffisent :

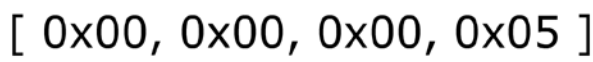


Figure 20-14. Le nombre 5 au sein du tableau d'octets.

Nous pouvons donc stocker l'entier non signé au sein d'un seul octet à l'aide de la méthode `writeByte` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un octet
fluxBinaire.writeByte(nombre);
```

L'octet est inscrit à l'index 0 du tableau, la figure 20-15 illustre l'octet sous notation hexadécimale:

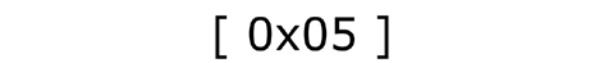


Figure 20-15. Un octet stocké à l'index 0.

Une fois l'octet écrit, nous pouvons le lire au sein du tableau sous la forme d'un entier non signé à l'aide la méthode `readUnsignedByte` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un octet
fluxBinaire.writeByte(nombre);

fluxBinaire.position = 0;

// lecture de l'octet non signé
var nombreStocke:uint = fluxBinaire.readUnsignedByte();

// affiche : 5
trace( nombreStocke );
```

Si nous tentons d'écrire une valeur dépassant un octet en stockage, seuls les bits inférieurs sont écrits au sein du flux.

Dans le code suivant nous évaluons la valeur 260 en binaire :

```
var nombre:uint = 260;

// affiche : 100000100
trace ( nombre.toString( 2 ) );
```

Le nombre entier non signé 260 occupe 9 bits, dont voici la représentation :

```
                1    0 0 0 0 0 1 0 0
|_ _ _ _ _ _ _ _|  |_ _ _ _ _ _ _ _|
 1 2 3 4 5 6 7 8    1 2 3 4 5 6 7 8
```

Comme nous l'avons vu précédemment, chaque index d'un tableau d'octets ne peut contenir qu'un seul octet, soit 8 bits. Ainsi, lorsque nous tentons d'écrire une valeur nécessitant plus de 8 bits, seuls les bits *de poids le plus faible* sont inscrits.

> Rappelez-vous, les bits dits *de poids le plus faible* sont à droite. Les bits dits *de poids le plus fort* sont ceux situés à gauche.

Un octet ne pouvant contenir que 8 bit, le 9ème bit débordant est ignoré :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 260;

// écriture d'un entier non signé
fluxBinaire.writeByte(nombre);

fluxBinaire.position = 0;

// lecture de l'entier non-signé
// affiche : 4
trace ( fluxBinaire.readUnsignedByte() );
```

En convertissant l'entier non signé 4 en notation binaire :

```
// lecture de l'entier non-signé au format binaire
// affiche : 100
trace ( fluxBinaire.readUnsignedByte().toString(2) );
```

Nous retrouvons les 3 bits 6, 7 et 8 de l'octet écrit :

```
                1    0 0 0 0 0 1 0 0
|_ _ _ _ _ _ _ _|  |_ _ _ _ _ _ _ _|
 1 2 3 4 5 6 7 8    1 2 3 4 5 6 7 8
```

Comme nous l'avons vu précédemment, un octet peut contenir une valeur oscillant 0 et 255 lorsque celui-ci est dit non signé. A l'inverse un octet signé, peut contenir un entier relatif allant de -128 à 127.

Ainsi si nous inscrivons un entier signé d'une valeur de -80, nous devons lire l'octet avec la méthode `readByte` :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:int = -80;

// écriture d'un entier signé
fluxBinaire.writeByte(nombre);

fluxBinaire.position = 0;

// lecture de l'octet signé
// affiche : -80
trace( fluxBinaire.readByte() );
```

Nous pouvons alors nous poser la question du type à utiliser lorsque nous utilisons les méthodes `writeByte` et `readByte` et `readUnsignedByte`.

Nous avons vu précédemment qu'il était recommandé d'utiliser le type `uint` lors de l'utilisation des méthodes `writeUnsignedInt` et `readUnsignedInt`. Il n'existe pas, à l'heure d'aujourd'hui de type `byte` ou `ubyte` en ActionScript 3.

Nous utilisons donc le type `int` en remplacement qui permet de stocker toutes les valeurs possibles d'un octet signé ou non signé.

Afin de stocker une valeur supérieure à 255 nous devons travailler sur deux octets. Prenons le cas du nombre entier 1200 sous sa forme binaire :

```
          1 0 0    1 0 1 1 0 0 0 0
|_ _ _ _ _ _ _ _|  |_ _ _ _ _ _ _ _|
 1 2 3 4 5 6 7 8    1 2 3 4 5 6 7 8
```

Deux octets sont nécessaires à l'écriture de ce nombre, pour cela nous utilisons la méthode `writeShort` dont voici la signature :

```
public function writeShort(value:int):void
```

Voici le détail du paramètre attendu :

- `value` : Un entier de 32 bits. Seuls les 16 bits inférieurs sont écrits dans le flux d'octets.

Un couple de deux octets est généralement appelé *mot* ou *word* en Anglais, celui-ci pouvant contenir une valeur allant de -32768 à 32767 ou 0 à 65 535 si celui est dit non signé.

L'intérêt de la méthode `writeShort` réside dans l'écriture d'un entier de 16 bit au lieu de 32 prévus par les types `int` et `uint`. Comme pour le type `byte` ou `ubyte`, il n'existe pas à l'heure d'aujourd'hui de type `short` ou `ushort` en ActionScript 3. Nous utilisons donc à nouveau le type `int` en remplacement.

Dans le code suivant nous inscrivons un entier de 16 bits non signé :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:int = 1200;

// écriture d'un entier de 16 bit non signé
fluxBinaire.writeShort(nombre);

// nombre d'octets stockés
// affiche : 2
trace( fluxBinaire.length );
```

En testant le code précédent nous remarquons que deux octets sont écrits dans le flux.

Afin de lire la valeur stockée nous utilisons la méthode `readShort` :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:int = 1200;

// écriture d'un entier de 16 bit non signé
fluxBinaire.writeShort(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 16 bit signé
// affiche : 1200
trace( fluxBinaire.readShort() );
```

Si nous analysons la représentation binaire de l'entier 1200 nous obtenons la représentation suivante :

```
          1 0 0    1 0 1 1 0 0 0 0
|_ _ _ _ _ _ _ _|  |_ _ _ _ _ _ _ _|
 1 2 3 4 5 6 7 8    1 2 3 4 5 6 7 8
```

L'entier 1200 est donc codé sur 16 bits, si nous appelons successivement la méthode `readUnsignedByte` nous évaluons l'entier octet par octet :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:int = 1200;

// écriture d'un entier de 16 bit non signé
fluxBinaire.writeShort(1200);
```

```
fluxBinaire.position = 0;

// lecture du premier octet
var premierOctet:int = fluxBinaire.readUnsignedByte();

// lecture du deuxième octet
var secondOctet:int = fluxBinaire.readUnsignedByte();

// affiche : 4
trace( premierOctet );

// affiche : 176
trace( secondOctet );
```

Bien entendu, nous pouvons stocker un nombre à virgule flottante au sein du tableau. ActionScript 3 utilise la norme IEEE 754 afin de traiter les nombres à virgule flottante.

Pour cela nous utilisons la méthode `writeFloat` :

```
public function writeFloat(value:Number):void
```

Voici le détail du paramètre attendu :

- `value` : un nombre à virgule flottante de 32 bits.

Dans le code suivant, nous inscrivons un flottant de 32 bits :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:Number = 12.5;

// écriture d'un flottant de 32 bits
fluxBinaire.writeFloat(nombre);

// réinitialisation du pointeur
fluxBinaire.position = 0;

// lecture du flottant
var flottant:Number = fluxBinaire.readFloat();

// affiche : 12.5
trace( flottant );

// affiche : 1100
trace( flottant.toString(2) );
```

La méthode `writeFloat` écrit un nombre au format 32 bits, 4 octets sont donc nécessaire au stockage d'un nombre flottant :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:Number = 12.5;

// écriture d'un flottant de 32 bits
fluxBinaire.writeFloat(nombre);

// affiche : 4
trace( fluxBinaire.length );
```

Il n'existe pas en ActionScript 3 de type `float`, nous utilisons donc le type `Number` en remplacement.

Si nous souhaitons stocker un nombre à virgule flottante plus grand, équivalent au type `Number` codé sur 64 bits nous pouvons utiliser la méthode `writeDouble` :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

// écriture d'un nombre à virgule flottante
fluxBinaire.writeDouble(12.5);

// affiche : 8
trace( fluxBinaire.length );
```

Notons qu'un *double* est l'équivalent du type `Number`.

ECMAScript 4 définit un type *double*, nous pourrions donc voir ce type apparaître un jour en ActionScript 3.

A retenir

- La classe `ByteArray` définit un ensemble de méthodes permettant d'écrire différents types de données.
- Certains types de données écrits au sein du tableau d'octets n'ont pas d'équivalent en ActionScript 3. C'est le cas des types `float`, `byte`, et `short`.
- Ces méthodes découpent automatiquement les valeurs passées en paramètres en groupes d'octets.

Copier des objets

La classe `ByteArray` intègre un sérialiseur et désérialiseur AMF permettant l'écriture de types au format AMF.

Nous pouvons utiliser la méthode `writeObject` de manière détournée en passant un objet, celui-ci est alors inscrit au sein du flux :

```
var fluxBinaire:ByteArray = new ByteArray();

// définition d'un objet
var parametres:Object = { age : 25, nom : "Bob" };

// écriture de l'objet au sein du flux
fluxBinaire.writeObject( parametres );
```

Afin de créer une copie de ce dernier nous appelons la méthode `readObject` :

```
var fluxBinaire:ByteArray = new ByteArray();

// définition d'un objet
var parametres:Object = { age : 25, nom : "Bob" };
```

```
// écriture de l'objet au sein du flux
fluxBinaire.writeObject( parametres );

fluxBinaire.position = 0;

// copie de l'objet parametres
var copieParametres:Object = fluxBinaire.readObject();
```

Nous pouvons ainsi créer une fonction personnalisée réalisant en interne tout ce processus :

```
function copieObjet ( pObjet:* ):*

{

    var fluxBinaire:ByteArray = new ByteArray();

    fluxBinaire.writeObject( pObjet );

    fluxBinaire.position = 0;

    return fluxBinaire.readObject();

}
```

Afin de copier un objet, nous devons simplement appeler la fonction `copieObjet` :

```
// définition d'un objet
var parametres:Object = { age : 25, nom : "Bob" };

var copieParametres:Object = copieObjet ( parametres );

/* affiche :
nom  :  Bob
age  :  25
*/
for ( var p:String in copieParametres ) trace( p, " : ", copieParametres[p] );
```

En modifiant les données de l'objet d'origine, nous voyons que l'objet `copieParametres` est bien dupliqué :

```
var copieParametres:Object = copieObjet ( parametres );

// modification du nom dans l'objet d'origine
parametres.nom = "Stevie";

/* affiche :
nom  :  Bob
age  :  25
*/
for ( var p:String in copieParametres ) trace( p, " : ", copieParametres[p] );
```

Notez que cette technique ne fonctionne que pour les objets littéraux et non les instances de classes.

A retenir

- La méthode `writeObject` permet d'écrire un objet composite au sein du tableau d'octets.
- Grace à la méthode `readObject` nous pouvons créer une copie de l'objet écrit.

Ecrire des données au format texte

Afin de stocker des données au format texte nous pouvons utiliser la méthode `writeUTFBytes` dont voici la signature :

```
public function writeUTFBytes(value:String):void
```

Chaque caractère composant la chaîne est encodé sur une suite d'un à quatre octets. Dans le code suivant, nous écrivons un texte simple :

```
var fluxBinaire:ByteArray = new ByteArray();

var chaine:String = "Bonjour Bob";

// affiche : 11
trace( chaine.length );

fluxBinaire.writeUTFBytes(chaine);

// affiche : 11
trace( fluxBinaire.length );
```

Si nous utilisons les 127 premiers caractères de l'alphabet, nous voyons que chaque caractère est codé sur un octet. A l'aide de la méthode `readByte`, nous pouvons lire récupérer le caractère de chaque octet :

```
var fluxBinaire:ByteArray = new ByteArray();

var chaine:String = "Bonjour Bob";

// affiche : 19
trace( chaine.length );

fluxBinaire.writeUTFBytes(chaine);

// affiche : 20
trace( fluxBinaire.length );

fluxBinaire.position = 0;

// affiche : 66
trace( fluxBinaire.readByte() );

// affiche : 111
trace( fluxBinaire.readByte() );
```

A l'inverse, si nous utilisons des caractères spéciaux, ils seront alors codés sur plusieurs octets :

```
var fluxBinaire:ByteArray = new ByteArray();

var chaine:String = "Bonjour Bob ça va ?";
```

```
// affiche : 19
trace( chaine.length );

fluxBinaire.writeUTFBytes(chaine);

// affiche : 20
trace( fluxBinaire.length );
```

Dans le code précédent, le caractère ç est codé sur deux octets.

Lire des données au format texte

Comme nous l'avons vu précédemment, il est primordial de ne pas tenter sortir du flux. Souvenez-vous, précédemment nous avions tenté de lire un octet non disponible au sein du flux, l'erreur suivante fut levée à l'exécution :

```
Error: Error #2030: Fin de fichier détectée.
```

Afin de garantir de ne jamais sortir du flux, nous utilisons la propriété `bytesAvailable`. Celle-ci renvoie le nombre d'octets disponible, autrement dit la soustraction de la longueur totale du flux et de la position du pointeur interne.

Dans le code suivant nous inscrivons des données texte au sein d'un tableau d'octets et calculons manuellement le nombre d'octets disponibles à la lecture :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// calcul manuel du nombre d'octets disponibles à la lecture
// affiche : 10
trace( fluxBinaire.length - fluxBinaire.position );
```

En réinitialisant le pointeur à l'aide de la propriété `position`, nous obtenons 10 octets disponibles.

Dans l'exemple suivant, nous utilisons la propriété `bytesAvailable`, qui permet de renvoyer automatiquement le nombre d'octets disponibles à la lecture :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// affiche : 10
```

```
trace( fluxBinaire.bytesAvailable );
```

Nous utilisons très souvent cette propriété afin d'être sur de ne pas aller trop loin dans la lecture des octets.

Afin d'extraire le texte stocké au sein du flux, nous utilisons la méthode `readUTFBytes` ayant la signature suivante :

```
public function readUTFBytes(length:uint):String
```

En passant le nombre d'octets à lire, celle-ci décode automatiquement chaque octet en caractère UTF-8. Ainsi dans le code suivant, nous lisons uniquement les 3 premiers caractères du flux :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// extrait les 3 premiers caractères du flux
// affiche : Bob
trace( fluxBinaire.readUTFBytes( 3 ) );
```

Nous pouvons donc utiliser la propriété `bytesAvailable` afin d'être sur de lire la totalité du texte stockée dans le flux :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// extrait la totalité de la chaîne de caractères
// affiche : Bob Groove
trace( fluxBinaire.readUTFBytes( fluxBinaire.bytesAvailable ) );
```

Nous utilisons généralement la propriété `bytesAvailable` avec une boucle `while` afin d'extraire chaque caractère :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

while ( fluxBinaire.bytesAvailable > 0 )

{

    /* affiche :
    B
    o
    b
```

```
        G
        r
        o
        o
        v
        e
        */
        trace( String.fromCharCode( fluxBinaire.readByte()) );

    }
```

Nous lisons chaque octet, tant qu'il en reste à lire. Afin de trouver le caractère correspondant au nombre, nous utilisons la méthode `fromCharCode` de la classe `String`.

Si nous insérons des caractères spéciaux, l'appel de la méthode `readUTFBytes` décode correctement les caractères encodés sur plusieurs octets :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte avec des caractères spéciaux
fluxBinaire.writeUTFBytes("Bob ça Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// extrait la totalité de la chaîne de caractères
// affiche : Bob ça Groove
trace( fluxBinaire.readUTFBytes( fluxBinaire.bytesAvailable ) );
```

A l'inverse, si nous utilisons la méthode `readByte` en évaluant chaque caractère, nous ne pourrons interpréter correctement le caractère `ç` encodé sur deux octets :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte avec des caractères spéciaux
fluxBinaire.writeUTFBytes("Bob ça Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

while ( fluxBinaire.bytesAvailable > 0 )

{

    /* affiche :
    B
    o
    b
    H
    ⊏
    a

    G
    r
    o
```

```
    o
    v
    e
    */
    trace( String.fromCharCode( fluxBinaire.readByte()) );

}
```

La méthode `readByte` n'est pas adaptée au décodage de chaînes encodées.

A retenir

- Afin de décoder sous forme de chaîne de caractères UTF-8 un ensemble d'octets, nous utilisons la méthode `readUTFBytes`.
- Celle-ci accepte en paramètre, le nombre d'octets à lire au sein du flux.

Compresser des données

La classe `ByteArray` dispose d'une méthode de compression de données utilisant l'algorithme zlib dont la spécification est disponible à l'adresse suivante :

http://www.ietf.org/rfc/rfc1950.txt

Il peut être intéressant d'utiliser cette fonctionnalité afin de compresser des données devant être sauvegardées.

Prenons le cas d'application suivant :

Nous devons sauver sur le poste de l'utilisateur un volume de données important. Afin de faciliter la représentation de ces données, un objet XML est utilisé. Pour stocker des données sur le poste client, nous utilisons la classe `flash.net.SharedObject` permettant de créer des cookies permanents.

Dans le code suivant, nous chargeons un fichier XML d'une taille de 1,5Mo au format binaire afin d'obtenir directement un objet `ByteArray` contenant le flux XML :

```
var chargeur:URLLoader = new URLLoader();

chargeur.dataFormat = URLLoaderDataFormat.BINARY;

chargeur.load( new URLRequest ("donnees.xml") );

chargeur.addEventListener( Event.COMPLETE, chargementTermine );

function chargementTermine ( pEvt:Event ):void

{
```

```
        // accès au flux binaire XML
        var fluxXML:ByteArray = pEvt.target.data;

        // affiche : 1547.358
        trace( fluxXML.length / 1024 );

    }
```

Puis nous sauvegardons le flux XML au sein d'un cookie permanent :

```
    var chargeur:URLLoader = new URLLoader();

    chargeur.dataFormat = URLLoaderDataFormat.BINARY;

    chargeur.load( new URLRequest ("donnees.xml") );

    chargeur.addEventListener( Event.COMPLETE, chargementTermine );

    // crée un cookie permanent du nom de "cookie"
    var monCookie:SharedObject = SharedObject.getLocal("cookie");

    function chargementTermine ( pEvt:Event ):void

    {

        // accès au flux binaire XML
        var fluxXML:ByteArray = pEvt.target.data;

        // affiche : 1547.358
        trace( fluxXML.length / 1000 );

        // écriture du flux XML dans le cookie
        monCookie.data.donneesXML = fluxXML;

        // sauvegarde du cookie sur le disque dur
        monCookie.flush();

    }
```

A l'appel de la méthode `flush`, le lecteur Flash tente de sauvegarder les données sur le disque. Le flux XML de large poids nécessite plus de 1 Mo d'espace disque et provoque l'ouverture du panneau *Enregistrement local* afin que l'utilisateur accepte la sauvegarde.

La figure 20-16 illustre le panneau :

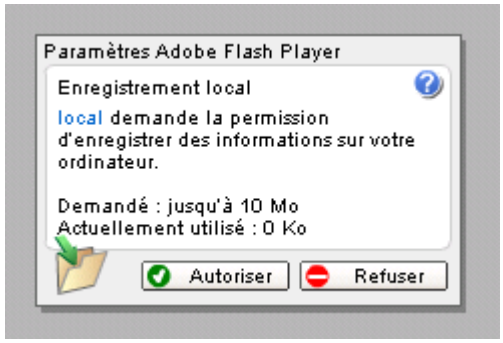


Figure 20-16. Panneau Enregistrement local.

Nous remarquons que le lecteur Flash nécessite alors une autorisation de 10 mo afin de sauver le cookie. 1548 Ko sont nécessaire à la sauvegarde du cookie :

La figure 20-17 illustre l'espace actuellement utilisé par le cookie :

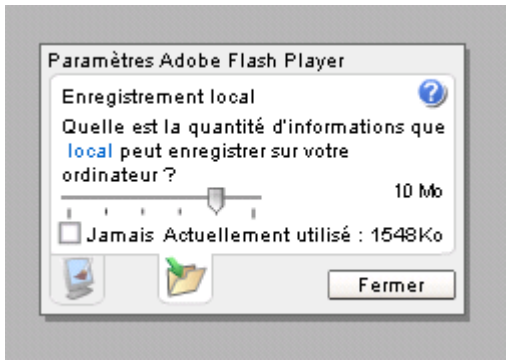


Figure 20-17. Espace utilisé par le cookie permanent.

En compressant simplement le flux XML à l'aide de la méthode `compress`, nous réduisons la taille du flux de près de 700% :

```
function chargementTermine ( pEvt:Event ):void

{

    // accès au flux binaire XML
    var fluxXML:ByteArray = pEvt.target.data;

    // affiche : 1547.358
    trace( fluxXML.length / 1024 );

    // compression du flux XML
    fluxXML.compress();

    // affiche : 212.24
    trace( fluxXML.length / 1024 );

    // écriture du flux XML dans le cookie
    monCookie.data.donneesXML = fluxXML;

    // sauvegarde du cookie sur le disque dur
    monCookie.flush();

}
```

En testant le code précédent, si nous affichons le panneau *Enregistrement local*, nous voyons que le cookie ne nécessite que 213 Ko d'espace disque :

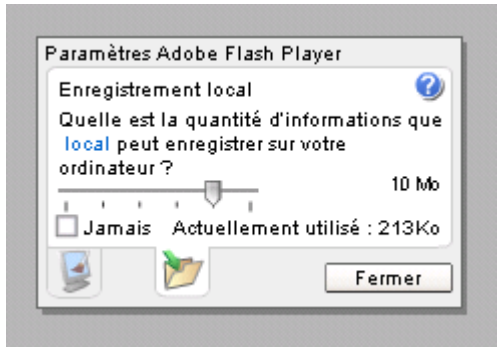


Figure 20-18. Un octet stocké à l'index 0.

Afin de lire le flux sauvegardé, nous devons appeler la méthode `uncompress`, puis extraire la chaîne compressée à l'aide de la méthode `readUTFBytes` :

```
function chargementTermine ( pEvt:Event ):void

{

    // accès au flux binaire XML
    var fluxXML:ByteArray = pEvt.target.data;

    // affiche : 1547.358
    trace( fluxXML.length / 1024 );

    fluxXML.compress();

    // affiche : 212.24
    trace( fluxXML.length / 1024 );

    // ecriture du flux XML dans le cookie
    monCookie.data.donneesXML = fluxXML;

    // sauvegarde du cookie sur le disque dur
    monCookie.flush();

    var fluxBinaireXML:ByteArray = monCookie.data.donneesXML;

    // décompression du flux XML binaire
    fluxXML.uncompress();

    // lecture de la chaîne XML
    var chaineXML:String = fluxBinaireXML.readUTFBytes (
fluxBinaireXML.bytesAvailable );

    // reconstitution d'un objet XML
    var donneesXML:XML = new XML ( chaineXML );

}
```

Grâce à la chaîne extraite du tableau d'octets, nous récréons un objet XML utilisable. La méthode `compress` nous a permis de réduire le poids du fichier XML de près de 1335 Ko.

A retenir

- La méthode `compress` de la classe `ByteArray` permet la compression des données binaires à l'aide de l'algorithme zlib.
- La méthode `uncompress` permet de décompresser le flux.

Sauvegarder un flux binaire

Le lecteur Flash 9 n'a pas la capacité d'exporter un flux généré en ActionScript 3 par la classe `FileReference`. Il serait intéressant de pouvoir passer à la méthode `download` de l'objet `FileReference` un tableau d'octets afin que le lecteur nous propose de sauver le flux, mais une telle fonctionnalité n'est pas présente dans le lecteur Flash 9.

Souvenez-vous, au cours du précédent chapitre nous avons exporté une image du même type grâce à AMFPHP. Dans l'exemple suivant nous allons exporter le tableau d'octets sans utiliser AMFPHP. La première étape consiste à utiliser une classe générant un fichier spécifique telle une image, une archive zip, ou n'importe quel type de fichier.

Nous allons réutiliser la classe d'encodage `EncodeurPNG` que nous avons utilisé au cours du précédent chapitre. Rappelez-vous, la classe `EncodeurPNG` nécessite en paramètre, une image de type `BitmapData` afin d'encoder les pixels dans une enveloppe PNG.

Nous créons dans le code suivant, une image d'une dimension de 320 par 240 pixels :

```
// import de la classe EncodeurPNG
import org.bytearray.encodage.images.EncodeurPNG ;

// création d'une image
var donneesBitmap:BitmapData = new BitmapData ( 320, 240, false, 0x990000 );

// encodage au format PNG
var fluxBinairePNG:ByteArray = EncodeurPNG.encode ( donneesBitmap );

// affiche : 690
trace( fluxBinairePNG.length );
```

Nous obtenons une longueur de 690 octets. Souvenez, vous en début de chapitre, nous avions abordé la structure d'un fichier PNG.

Pour exporter le flux nous devons transmettre par connexion HTTP le flux binaire, puis prévoir un script serveur afin de rendre le flux disponible en téléchargement par une fenêtre appropriée.

Pour permettre cela, nous créons un fichier `export.php` contenant le code PHP suivant :

```
<?php
```

```
if ( isset ( $GLOBALS["HTTP_RAW_POST_DATA"] )) {

    $flux = $GLOBALS["HTTP_RAW_POST_DATA"];

    header('Content-Type: image/png');
    header("Content-Disposition: attachment; filename=".$_GET['nom']);
    echo $flux;

}  else echo 'An error occured.';

?>
```

Nous sauvons le fichier `export.php` sur notre serveur local, puis nous transmettons le flux binaire au script distant en passant le tableau d'octets à la propriété `data` de l'objet `URLRequest` :

```
// import de la classe EncodeurPNG
import org.bytearray.encodage.images.EncodeurPNG ;

// création d'une image
var donneesBitmap:BitmapData = new BitmapData ( 320, 240, false, 0x990000 );

// encodage au format PNG
var fluxBinairePNG:ByteArray = EncodeurPNG.encode ( donneesBitmap );

// entête HTTP au format brut
var enteteHTTP:URLRequestHeader = new URLRequestHeader ("Content-type",
"application/octet-stream");

var requete:URLRequest = new
URLRequest("http://localhost/export_image/export.php?nom=sketch.jpg");

requete.requestHeaders.push(enteteHTTP);

requete.method = URLRequestMethod.POST;

requete.data = fluxBinairePNG;

navigateToURL(requete, "_blank");
```

Notons que grâce à la classe à la classe `URLRequestHeader`, nous indiquons au lecteur Flash de ne pas traiter les données à transmettre en HTTP en tant que chaîne mais en tant que flux binaire.

> Souvenez-vous, lors du chapitre 14 intitulé *Charger et envoyer des données*, nous avions découvert qu'il était impossible depuis l'environnement de développement de Flash, d'utiliser la méthode POST en utilisant la fonction `navigateToURL`.
>
> Il est donc obligatoire de tester le code précédent au sein d'une page navigateur. Dans le cas contraire, le flux binaire sera transmis par la méthode GET et sera donc traité telle une chaîne de caractères.

Si nous avions voulu sauver l'image sur le serveur, nous aurions utilisé le code PHP suivant au sein du fichier `export.php` :

```
<?php

if ( isset ( $GLOBALS["HTTP_RAW_POST_DATA"] )) {

    $flux =  $GLOBALS["HTTP_RAW_POST_DATA"];

    $fp = fopen($_GET['nom'], 'wb');
    fwrite($fp, $im);
    fclose($fp);

}

?>
```

Bien entendu, le code précédent peut être modifié afin de sauver le flux binaire dans un répertoire spécifique.

A retenir

- Le lecteur Flash est incapable d'exporter un flux binaire sans un script serveur.
- Lors de l'envoi de données binaire depuis le lecteur Flash, nous devons obligatoirement utiliser la classe `URLRequestHeader` afin de préciser que les données transmises sont de type binaire.
- Les données binaires arrivent en PHP au sein du la propriété `HTTP_RAW_POST_DATA` du tableau `$GLOBALS` : `$GLOBALS["HTTP_RAW_POST_DATA"]`.

Générer un PDF

Afin de démontrer l'intérêt de la classe `ByteArray` dans un projet réel, nous allons générer un fichier PDF dynamiquement en ActionScript 3. Pour cela, nous allons utiliser la librairie `AlivePDF` qui s'appuie sur la classe `ByteArray` pour générer le fichier PDF.

En réalité il est aussi possible de générer un fichier PDF en ActionScript 1 et 2 sans avoir recours à la classe `ByteArray`, car le format PDF est basé sur une simple chaîne de caractères. En revanche l'intégration d'images ou autres fichiers nécessite une manipulation des données binaire ce qui est réservé à ActionScript 3.

Voici le contenu d'un fichier PDF ouvert avec le bloc notes :

```
%PDF-1.4
1 0 obj
<< /Type /Catalog
/Outlines 2 0 R
/Pages 3 0 R
>>
```

```
endobj
2 0 obj
<< /Type Outlines
/Count 0
>>
endobj
3 0 obj
<< /Type /Pages
/Kids [4 0 R]
/Count 1
>>
endobj
4 0 obj
<< /Type /Page
/Parent 3 0 R
/MediaBox [0 0 612 792]
/Contents 5 0 R
/Resources << /ProcSet 6 0 R >>
>>
endobj
5 0 obj
<< /Length 35 >>
stream
…Page-marking operators…
endstream
endobj
6 0 obj
[/PDF]
endobj
xref
0 7
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000179 00000 n
0000000300 00000 n
0000000384 00000 n
trailer
<< /Size 7
/Root 1 0 R
>>
startxref
408
%%EOF
```

Afin de générer un PDF en ActionScript nous utilisons la librairie `AlivePDF` disponible à l'adresse suivante :

http://code.google.com/p/alivepdf/downloads/list

Nous allons utiliser la version 0.1.4 afin de créer un PDF de deux pages contenant du texte sur la première page, puis une image sur la deuxième page.

Une fois les sources de la librairie téléchargées. Nous plaçons le répertoire `org` contenant les classes à côté d'un nouveau document FLA, puis nous importons la classe PDF :

```
import org.alivepdf.pdf.PDF;
import org.alivepdf.layout.Orientation;
import org.alivepdf.layout.Unit;
import org.alivepdf.layout.Size;
import org.alivepdf.layout.Layout;
import org.alivepdf.display.Display;
import org.alivepdf.saving.Download;
import org.alivepdf.saving.Method;

var myPDF:PDF = new PDF ( Orientation.PORTRAIT, Unit.MM, Size.A4 );
```

Nous définissons le mode d'affichage du PDF :

```
myPDF.setDisplayMode( Display.FULL_PAGE, Layout.SINGLE_PAGE );
```

Puis nous ajoutons une nouvelle page à l'aide de la méthode `addPage` :

```
myPDF.addPage();
```

Enfin nous sauvons le PDF :

```
myPDF.savePDF ( Method.REMOTE, 'http://localhost/pdf/create.php',
Download.ATTACHMENT, 'monPDF.pdf' );
```

Nous devons passer en deuxième paramètre à la méthode `savePDF`, l'adresse du script `create.php` présent dans les sources du projet `AlivePDF`.

A l'exécution, le SWF ouvre la fenêtre illustrée en figure 20-19 :

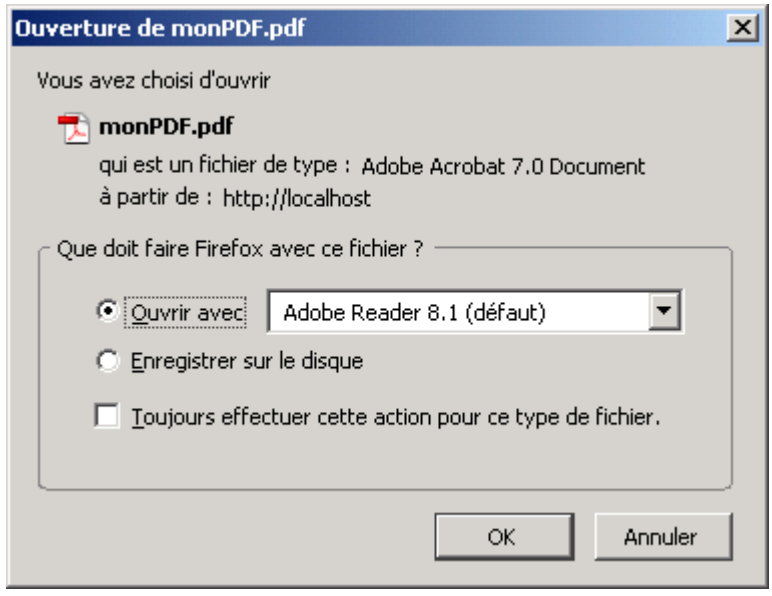


Figure 20-19. Sauvegarde du document PDF.

A l'ouverture le PDF contient une seule page blanche. Afin d'enrichir notre document PDF, nous allons ajouter du texte, en important deux nouvelles classes :

```
import org.alivepdf.fonts.Style;
import org.alivepdf.fonts.FontFamily;
import org.alivepdf.colors.RGBColor;
```

Puis nous utilisons les différentes méthodes d'écriture de texte :

```
myPDF.textStyle ( new RGBColor ( 255, 100, 0 ) );
myPDF.setFont( FontFamily.HELVETICA, Style.BOLD );
myPDF.setFontSize ( 20 );
myPDF.addText ('Voilà du texte !', 70, 12);
myPDF.addLink ( 70, 4, 52, 16, "http://alivepdf.bytearray.org");
```

En testant le code précédent, un document PDF est généré, à l'ouverture, le texte est affiché et redirige sur le site du projet `AlivePDF` lors du clic.

La figure 20-20 illustre le résultat :

Figure 20-20. Texte intégré au PDF.

Afin d'intégrer une image, nous importons une image dans la bibliothèque, puis nous l'associons à une classe nommée Logo.

Grâce à la méthode `addImage`, nous intégrons l'image bitmap en deuxième page du PDF :

```
myPDF.addPage();

var donneesBitmap:Logo = new Logo(0,0);

var imageLogo:Bitmap = new Bitmap ( donneesBitmap );

myPDF.addImage ( imageLogo );
```

La figure 20-21 illustre le résultat :

Figure 20-21. Image intégrée au PDF.

Ainsi se termine notre aventure binaire. Au cours du prochain chapitre, nous mettrons en application la plupart des concepts abordés durant l'ensemble de l'ouvrage.