1. The first bits

Flash Player 9 opened new areas of discovery for Flash developers, by allowing them to manipulate data at a low-level. In this chapter, we are going to discover the basics of binary data and see how to use that knowledge in the Flash Player through the ByteArray class. Each concept will be illustrated with different use cases. Once this chapter finished, you will have everything you need to understand for instance any file specification and create poweful things like a file parser or generate any kind of files.

Binary coding

We have to come back to the roots of computing in order to understand the reasons behind the existence of binary data. The lowest level of input a computer can understand is represented is represented by the numbers 0 and 1, which define the state of an electrical circuit :

- 0 : the circuit is opened.
- 1 : the circuit is closed.

Processors only understand binary. Though all this may seem complicated at first sight, it is simply a different notation to express data. Before focusing on the `ByteArray` class, we must first be comfortable with the concept of *base arithmetic*. The man is counting with a base 10 and this is what we call the *decimal base*. To illustrate the notion of time, we use a *sexagesimal base* consisting of 60 symbols. Thus, when 60 seconds have elapsed, we do not write 61 seconds, we add a new unit, that is to say, a minute and return to 0 in the number of seconds.

In everyday's life, we use all the following symbols to represent numbers :

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

Beyond 9, we must combine the previous symbols. To this we add a new unit, then we start from 0 again. For every movement on the left, we add a power of 10. The Figure 23.1 illustrates the splitting of a number into different groups of power of 10 :

Figure 23.1

Groups of power of 10.

Our decimal system works by power of 10, we can express the number 750 as follows :

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

Unlike the decimal, binary notation allows the use of symbols 0 and 1 only. The number 150 is expressed in binary as follows :

```
10010110
```

We will learn how to convert binary notation into decimal notation. This time, we apply the same concept as with decimals, but now with powers of 2. Figure 23.2 illustrates first how to convert a number from its binary form to different groups of power of 2 :

Figure 23.2

Groups of power of 2.

At school, we learned to count to 10 in decimal. If the standard had been using the binary notation, we would read the right column of Table 23.1 very naturally and the left column would seem confusing :

Tableau 23.1 illustrates the difference between decimal and binary notation :

Decimal notation	**Binary notation**
0	0
1	1
2	10
3	11
4	100
5	101

6	110
7	111
8	1000
9	1001
10	1010

To understand the binary notation, we must focus and understand what the weight of a bit is all about. Once this will be understood, we will introduce the concept of byte.

Note

- A CPU understands only binary.
- Among the most well-known notations, the most common ones are decimal (base 10) and sexagesimal (base 60).
- Our decimal system (base 10) works by power of 10, the binary notation (base 2) works by power of two.

Let's discover right away the concept of weight when talking about bits.

Position and weight of the bit

Let's get back to the number 150 which is expressed this way in binary :

```
10010110
```

The symbols used in binary are called bit, the word comes from *binary digit*. Unlike the decimal notation which works with powers of 10, the binary notation works with power of 2. Each bit movement modifies the original value related to the weight of the bit and the base being used. Bearing in mind that the position of the bit defines its weight, the strongest bit (*most significant bit* or *msb*) is always positioned at the extreme left, unlike the weakest bit (*less significant bit* or *lsb*) which is positioned at the extreme right.

The following figure illustrates the idea :

Figure 23.3

Less significant bit.

The Figure 23.4 illustrates the location of the most significant bit (msb) :

Figure 23.4

Most significant bit.

As a result, toggling one of the bits will cause a modification on the number depending on the weight of each bit. Continuing with the number 150, we see that if we toggle the strongest bit to 0, we subtract 128 (2^7) to the number 150 (see Figure 23.5) :

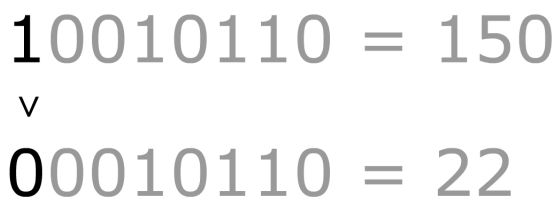

Figure 23.5

Toggling the most significant bit to 0.

Conversely, if we toggle a less significant bit, the impact on the final value is less important, in the following case, we only substract 4 (2^2) to the value (see Figure 23.6) :

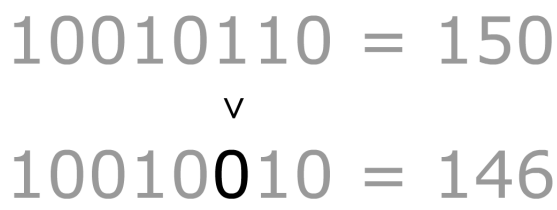

Figure 23.6

Toggling a less significant bit to 0.

As a result, in order to convert a binary notation to decimal, it is just a simple as that. Taking our number 150 :

```
10010110
```

We need to multiply each bit by its weight and sum the results :

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

This gives us in a more compact form :

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

Now, a quick exercise, convert the following binary notation to decimal :

```
1000
```

This results to 8 in decimal, got it ?

Now, another one even simpler :

```
101
```

This gives us 5, maybe you already noticed that a binary value wih its less significant bit set to 1 cannot be even. We will come back to this later and discover how to detect even or odd values with bitwise operators.

We often find the concept of bits in colors. We will see later that the colors are generally stored into 8, 16 or 32-bits. A nerdy joke says that there are 10 types of people in the world. Those who understand binary and those who don't. We know that 10 in binary corresponds to $1*2^1$, which gives us 2 :

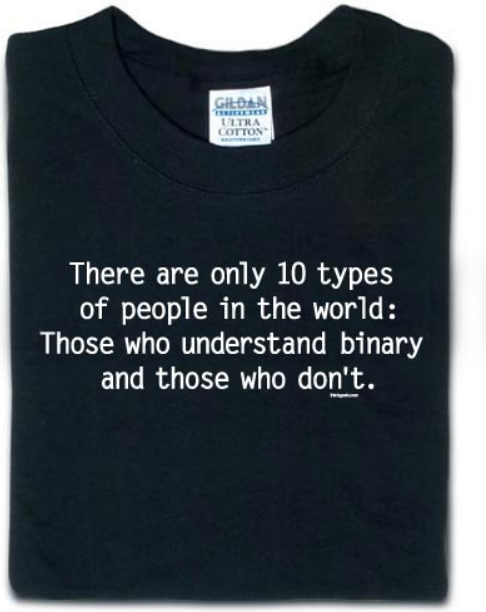

Figure 23.7. Nerdy joke.

As for the base 2, other databases exist, such as the bases 8 or 16, more commonly called hexadecimal notation which is generally used to represent colors and its components or more generally data inside hexdecimal editors. Unlike the binary notation made of only 2 symbols, the base 16 is using 16 symbols.

In the following code we convert the decimal value 120 to base 16 :

```
var integer:int = 120;

// outputs : 78
trace ( integer.toString( 16 ) );
```

Unlike the base 2, or base 10, base 16 uses 16 symbols ranging from 0 to F in order to express a number. We are therefore working with powers of 16 (see Figure 23.7) :

1 A

16^1 16^0

Figure 23.7

Groups of power of 16.

Table 23.2 lists all the symbols used by the base 16 :

Hexadecimal notation	Decimal notation
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

The hexadecimal value 1A can be converted to decimal as follows :

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

Remember that by default, the Flash Player outputs values using the decimal notation (base 10). However, when working with a binary stream, it may be necessary to express numbers in different bases. Instead of making the conversion manually, we can rely on the `toString` method from the `Number` class. Here is its signature :

```
toString(radix:*=10):String
```

The `radix` parameter defines the base to convert the value into. In the following code we convert the decimal number 5 to its binary notation, we use 2 as parameter as for base 2 :

```
var value:int = 5;

// outputs : 101
trace ( value.toString( 2 ) );
```

If we start counting from 1 to 10 in binary, we find the values from Table 23.2 :

```
var zero:int = 0;
var one:int = 1;
var two:int = 2;
var three:int = 3;
var four:int = 4;
var five:int = 5;
var six:int = 6;
var seven:int = 7;
var eight:int = 8;
var nine:int = 9;
var ten:int = 10;

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

// outputs : 1
trace( one.toString( 2 ) );

// outputs : 10
trace( two.toString( 2 ) );

// outputs : 11
trace( three.toString( 2 ) );

// outputs : 100
trace( four.toString( 2 ) );

// outputs : 101
trace( five.toString( 2 ) );

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

// outputs : 111
trace( seven.toString( 2 ) );

// outputs : 1000
trace( eight.toString( 2 ) );

// outputs : 1001
trace( nine.toString( 2 ) );

// outputs : 1010
trace( ten.toString( 2 ) );
```

Similarly, we can convert a decimal number into its hexadecimal notation :

```
var zero:int = 0;
var one:int = 1;
var two:int = 2;
var three:int = 3;
var four:int = 4;
var five:int = 5;
var six:int = 6;
var seven:int = 7;
var eight:int = 8;
var nine:int = 9;
```

```
var ten:int = 10;
var eleven:int = 11;
var twelve:int = 12;
var thirteen:int = 13;
var fourteen:int = 14;
var fifteen:int = 15;

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

// outputs : 1
trace( one.toString( 16 ) );

// outputs : 2
trace( two.toString( 16 ) );

// outputs : 3
trace( three.toString( 16 ) );

// outputs : 4
trace( four.toString( 16 ) );

// outputs : 5
trace( five.toString( 16 ) );

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

// outputs : 7
trace( seven.toString( 16 ) );

// outputs : 8
trace( eight.toString( 16 ) );

// outputs : 9
trace( nine.toString( 16 ) );

// outputs : a
trace( ten.toString( 16 ) );

// outputs : b
trace( eleven.toString( 16 ) );

// outputs : c
trace( twelve.toString( 16 ) );

// outputs : d
trace( thirteen.toString( 16 ) );

// outputs : e
trace( fourteen.toString( 16 ) );

// outputs : f
trace( fifteen.toString( 16 ) );
```

We just talked about all those bits, but how do we store them to express something ? To handle and store a set of bits, engineers decided to group them in packets, thus was born the concept of *byte* that we will discuss now.

Note

- We talk about the weight of a bit to express its strength.
- The most significant bit (*msb*) is always positionned on the extreme left.
- The less significant bit (*lsb*) is always positionned on the extreme right.
- The `toString` method from the `Number` class allows us to express a numerical value in different bases.

What is a byte ?

The *byte* enables us to express a quantity of data. We use it every day in the computer world to show by example the weight of a file. Of course, we will not say that an MP3 file weighs 3 145 728 bytes, but rather 3MB.

We generally add a prefix as *kilo* or *mega* to express a volume of bytes :

- 1 kilo-byte (kb) = 10^3 bytes (1 000 bytes) ;
- 1 mega-byte (mb) = 10^6 bytes = 1 000 kb (1 000 000 bytes).

Be careful not to confuse the notion of *bit* and *byte*. A byte is made of 8 bits :

```
 1 0 1 1 0 0 0 0  byte
|_ _ _ _ _ _ _ _|
 7 6 5 4 3 2 1 0  bits
```

As you may have already noticed, the bit indexing is zero based and as a byte only contains 8 bits, it cannot contain a value above 255 (*unsigned* byte). To convert a full byte into decimal, we use the technique discussed before.

We multiply each bit by its weight and sum the results :

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

Which gives us :

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

We will see that it is also possible to express in a byte a value ranging between -128 and 127 by using what we call a *signed* byte (negative). Now, you may wonder why a maximum value of 127 ?

In case of negative values, the msb is used to specifiy the sign, as a result, only 7 bits are therefore available to store the value :

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

Which gives us :

$$64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$$

It is important to note that when working with binary data, the base 16 is often used in software such as hexadecimal editors to simplify the representation of each byte. Choose your favorite editor, and make it your best friend when working with binary.

Here is a list of some of the best hexadecimal editors :

- Free Hex Editor Neo (free) : http://www.hhdsoftware.com/Family/hex-editor.html ;
- HxD (free) : http://mh-nexus.de/hxd ;

- Hex Workshop : http://www.hexworkshop.com ;
- Hexprobe : http://www.hexprobe.com/hexprobe.

In practice, it is easier to read a byte in hexadecimal as follows :

```
4E
```

Rather than in the following binary notation :

```
1001110
```

The value 255 stored inside an unsigned byte, can be expressed by two symbols thanks to the following hexadecimal notation :

```
FF
```

Remember, by combining the weight of each symbol and summing the result we get the original value :

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

To put all that into practice, we will open a PNG image into our favorite hexadecimal editor.

The Figure 23.8 illustrates the beginning of the binary stream :

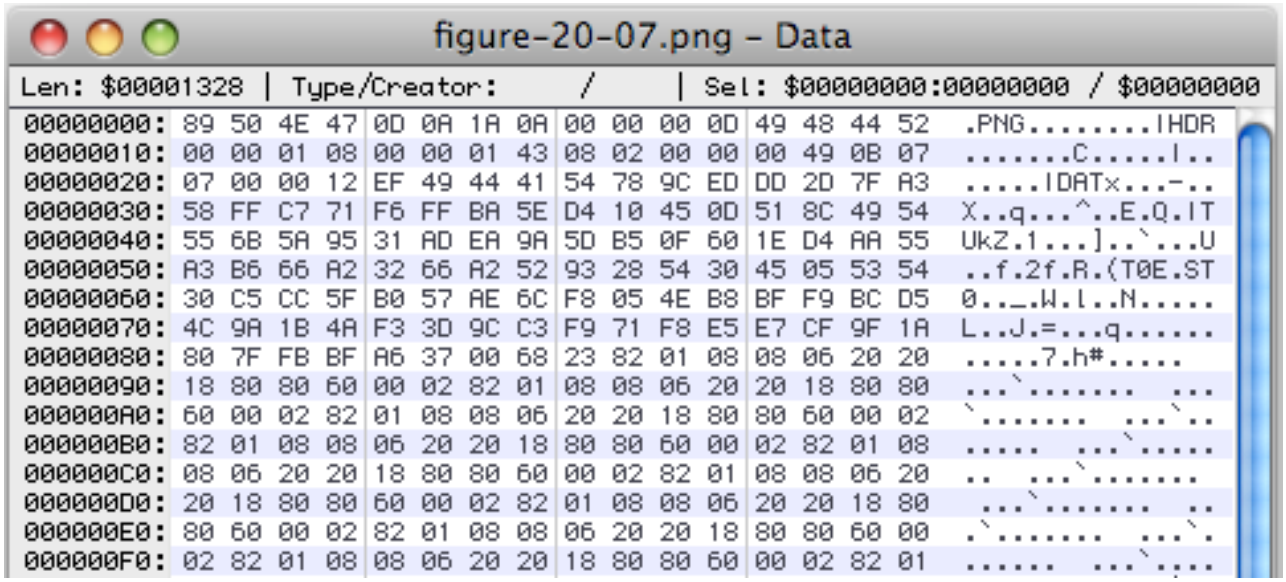

Figure 23.8

Hexadecimal editor.

We can see that each column is made of two symbols representing one byte. As its name suggests, an hexadecimal editor shows each byte using the hexadecimal notation. Why such a decision ? This is explained for practical reasons : the FF pair of symbols used to represent the maximum value of a byte is optimized in terms of space and less painful to read and memorize.

As we shall see later, each file can be identified by its header. Hence, by reading the PNG specification (http://www.w3.org/TR/PNG), we see that any valid PNG file **must** start with a signature composed of the following decimal values :

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

By converting to hexadecimal notation each of these decimal values, we end up with the same values as in our PNG specification :

```
var firstByte:int = 137;
var secondByte:int = 80;
var thirdByte:int = 78;
var fourthByte:int = 71;
var fifthByte:int = 13;
var sixthByte:int = 10;
var seventhByte:int = 26;
var eighthByte:int = 10;

// outputs : 89
trace( firstByte.toString ( 16 ) );

// outputs : 50
trace( secondByte.toString ( 16 ) );

// outputs : 4e
trace( thirdByte.toString ( 16 ) );

// outputs : 47
trace( fourthByte.toString ( 16 ) );

// outputs : d
trace( fifthByte.toString ( 16 ) );

// outputs : a
trace( sixthByte.toString ( 16 ) );

// outputs : 1a
trace( seventhByte.toString ( 16 ) );

// outputs : a
trace( eighthByte.toString ( 16 ) );
```

Figure 23.9 highlights the PNG file header :

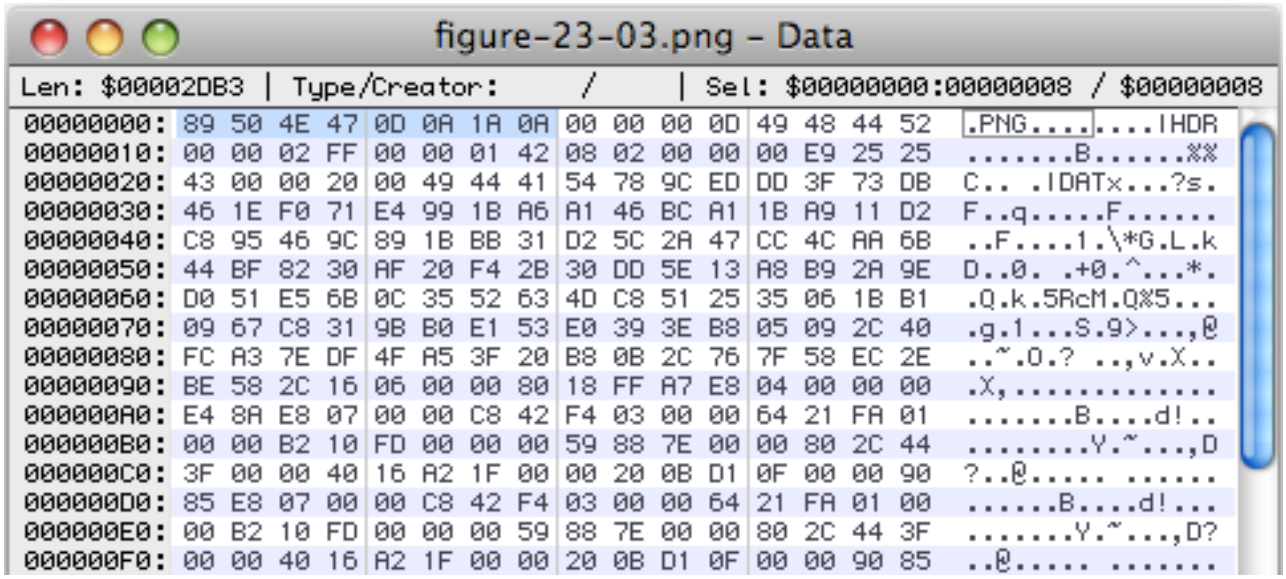

Figure 23.9

PNG file signature.

Let's spend some time now on the byte ordering mechanism, with the concept of *endianness*.

Note

- Do not confuse 1 *byte* and 1 *bit*.
- A *byte* is made of 8 bits.
- An unsigned byte can contain the maximum value 255 ($2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$).
- An signed byte can contain the maximum value 127 ($2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$).
- The hexadecimal notation is often used to represent bytes.

Endianness

When using multiple bytes to store a value, it is important to take into account the storage order. This concept is called *endianness* or *byte ordering*. Remember, in the section entitled *« Position and weight of the bit »*, we saw the differences of weights between bits though the notion of *msb* (most significant bit) and *lsb* (less significant bit). The same concept applies to a group of bytes. We then speak about the *most significant byte* or *MSB* and *less significant byte* or *LSB*.

> Note that we use the acronym *MSB* and *LSB* capitalized to express byte order. Conversely we use lowercase in the *msb* and *lsb* acronyms to describe the weight of a bit.

Suppose we were to store the following value :

```
550000000
```

This number is a 32-bit unsigned integer and requires 4 bytes to be stored :

```
00100000110010000101010110000000
```

We can group those into four bytes :

```
00100000 11001000 01010101 10000000
```

Either way represented into as hexadecimal :

```
20 C8 55 80
```

Processors, such as the Motorola 68000 and SPARC platforms used by Sun Microsystems, use this kind of storage and are considered *big-endian*. We use this term because the groups of bytes « ends » on the left with the most significant byte (*MSB*). Other processors, like the famous 5602 Motorola or Intel x86 are *little-endian* and store the bytes in the opposite order (see Table 23.3).

Tableau 23.3 : Difference between big-endian and little endian.

Big-endian			
0	1	2	3
20 (MSB)	C8	55	80
Little-endian			

0	1	2	3
80	55	C8	20 (MSB)

Hence, big-endian CPU's will end up with the value 550000000 when reading the following bytes :

```
20 C8 55 80
```

Little-endians will interpret the exact same result when reading the bytes in the opposite order :

```
80 55 C8 20
```

During the development of ActionScript 3 applications using the `ByteArray` class, we must remember to always consider endianness, especially in a context of external communication. The concept of byte ordering has no real consequence when reading one byte at a time. Conversely, when multiple bytes must be interpreted to express one single value, we must **absolutely** take into account *endianness*.

Note

- Endianness only applies to bytes, never on bits.
- Some CPU architectures use little-endian, others use big-endian.
- Byte ordering or endianness is an important concept to consider when sending data across different computers which may have a different CPU architecture.
- Usually sockets API on operating systems handle this automatically through simple API's.

Now let's work with bitwise operators to really dig into the the power of binary data manipulation with ActionScript 3.

Bitwise operators

In order to accurately manipulate each bit across bytes, we will use throughout the book operators called bitwise operators. These are listed below :

- & (Bitwise AND) : Converts expression1 and expression2 to 32-bit unsigned integers, and performs a Boolean AND operation on each bit of the integer parameters.
- << (Bitwise left shift) : Converts expression1 and shiftCount to 32-bit integers, and shifts all the bits in expression1 to the left by the number of places specified by the integer resulting from the conversion of shiftCount.
- ~ (Bitwise NOT) : Converts expression to a 32-bit signed integer, and then applies a bitwise one's complement.
- | (Bitwise OR) : Converts expression1 and expression2 to 32-bit unsigned integers, and places a 1 in each bit position where the corresponding bits of either expression1 or expression2 are 1.
- >> (Bitwise right shift) : Converts expression and shiftCount to 32-bit integers, and shifts all the bits in expression to the right by the number of places specified by the integer that results from the conversion of shiftCount.
- >>> (Bitwise unsigned right shift) : The same as the bitwise right shift (>>) operator except that it does not preserve the sign of the original expression because the bits on the left are always filled with 0.

- ^ (Bitwise XOR) : Converts expression1 and expression2 to 32-bit unsigned integers, and places a 1 in each bit position where the corresponding bits in expression1 or expression2, but not both, are 1.

Don't worry, we will not use all of these operators, only some of them remain truly useful in everyday's life as a developer, but it is important to understand all of them. The most important operators to understand are the bitwise shifting operators expressed by the left (<<) or right (>> and >>>) arrows.

Keep in mind that using any of those operators will always convert the values you are working with, to signed or unsigned integers. Any floating value will be converted to integers whatever the bitwise operator used :

```
// a decimal value
var value:Number = 67.4;

// outputs : 1
trace( value & 1 );

//outputs : 134
trace( value << 1 );

// outputs : -68
trace( ~value );

// outputs : 67
trace( value | 1 );

// outputs : 33
trace( value >> 1 );

// outputs : 33
trace( value >>> 1 );

// outputs : 66
trace( value ^ 1 );
```

We are going to cover now the bitwise shifting operators which are really simple to understand. By moving the bits to the right and to the left, we will produce a modification on the value being shifted. Each bit movement equals to a division or a multiplication of a power of two related to the weight of the bit.

To understand the following mechanism, always think in binary and forget the usual decimal notation. So take the number 8 in binary notation :

```
1000
```

If we start moving the bits to the right, no free space is available. As a result, shifting all the bits one bit to the right gives us the value 4 (100 in binary) :

```
1000
0100 >> 1
---------
 100
```

Thus, this approach is extremely useful for achieving faster performance division or multiplication by a power of 2, each n shift will multiply or divide by 2^n :

```
expression >> shiftCount
```

To translate that in ActionScript, let's take the following code :

```
var size:int = 8;
```

```
// division by 2^1
var result:int = size >> 1;

// outputs : 4
trace( resultat );
```

In the previous case, we shifted all the bits one bit to the right, so we just divided by 2^1. By moving 2 bits to the right we divide by 2^2:

```
var size:int = 8;

// division by 2^2
var result:int = size >> 2;

// outputs : 2
trace( result );
```

You would imagine, moving the bits to the opposite direction will just produce the opposite result, and you would be just right :

```
var size:int = 8;

// multiply by 2^2
var result:int = size << 2;

// outputs : 32
trace( result );
```

By moving the bits to the left we produce a multiplication :

```
  1000
100000 << 2
-----------
100000
```

In terms of performance, it may be advantageous to use this technique to divide or multiply a value. In the following code we use the traditional division operator :

```
var started:Number = getTimer();

var value:int = 8;
var result:int;

for ( var i:int = 0; i< 5000000; i++ )
    result = value / 4;

// outputs : 2
trace( result );

//outputs : 617
trace( getTimer() - started );
```

Now let's use use the bitwise operator, as we need to divide, we will use the bitwise right shift operator :

```
var started:Number = getTimer();

var value:int = 8;
var result:int;

for ( var i:int = 0; i< 5000000; i++ )
    result = value >> 2;

// outputs : 2
trace( result );
```

```
//outputs : 586
trace( getTimer() - started );
```

We see a very slight difference in terms of performance. Of course, the bitwise shifting operators are not intented to be used only for that. Now you may wonder what the bitwise unsigned right shift operator is for, we will come back this one later.

Likewise, to toggle a bit, we use the | (OR) operator. This operator, just like an OR statement for a condition simply performs an OR operation on the bits of two patterns of the same length :

```
expression1 | expression2
```

The result is 1 if the first bit is 1 OR the second bit is 1, OR both bits are 1. The following figure illustrates the behavior :

```
   0110
OR 0011
   ----
   0111
```

To toggle the bits of a byte easily, we can use the OR operator. In the following code, we toggle the 7th bit (msb) to 1 :

```
// byte : 127
var byte:int = 0x7F;

// set the 7th bit on the left (msb)
// outputs : 255
trace( byte | 1<<7 );
```

Note that we just used the bitwise left shifting operator to produce a mask to process the OR operation on the bits. To understand what is happening internally, here is the binary operation :

```
  01111111
| 10000000  1 << 7
  --------
  11111111
```

Likewise, we can use the ^ (XOR) operator, which simply performs an XOR operation on the bits of two patterns of the same length :

```
expression1 ^ expression2
```

The result in each position is 1 if the two bits are different, and 0 if they are the same. The following figure illustrates the behavior :

```
    1101
XOR 1011
    ----
    0110
```

Then comes the & (AND) operator. Just like an AND statement for a condition, the & operator performs an AND operation on the bits of two patterns of the same length :

```
expression1 & expression2
```

The result is 1 if the first bit is 1 AND the second bit is 1, otherwise, the result is 0 :

```
    1101
AND 1011
```

```
    ----
    1001
```

It is commonly used in bit programming to test if a bit is set through the use of a mask (bit masking). In the following code, we check whether each bit contained in a byte is set :

```
// byte : 255
var byte:int = 0xFF;

// testing of each bit from the byte
// outputs : true
trace( (byte & (1<<7)) != 0 );
// outputs : true
trace( (byte & (1<<6)) != 0 );
// outputs : true
trace( (byte & (1<<5)) != 0 );
// outputs : true
trace( (byte & (1<<4)) != 0 );
// outputs : true
trace( (byte & (1<<3)) != 0 );
// outputs : true
trace( (byte & (1<<2)) != 0 );
// outputs : true
trace( (byte & (1<<1)) != 0 );
// outputs : true
trace( (byte & 1) != 0 );
```

In the same way, by taking the original value :

```
1
```

And shifting the bits to the left, we get the following binary outcome :

```
      1
1000000 1 << 7
-------
1000000
```

Note how we used the bitwise left shift operator (<<) to shift the bits to the left to produce once again the appropriate mask for detecting each bit state. The purpose of the mask here is to create an appropriate pattern which will allow us to preserve only the bits we want to test.

We now have our mask to test the value of the most significant bit (msb) :

```
  11111111
& 10000000
  --------
  10000000
```

In the following binary representation, we see that the mask is compared again through an AND operation to the other bits, only the corresponding bits are preserved :

```
  01010100
& 01000000
  --------
  01000000
```

In the following situation, the corresponding bit does not match, therefore nothing is preserved, all the bits are reset :

```
  00100000
& 01000000
  --------
  00000000
```

In order to store easily some bits, we can create easily a `writeBits` and `readBits` function :

```
var buffer:int = 0;
var inc:int = 0;
var incRead:int = 0;

function writeBit (pBit:uint):uint
{
    return buffer |= (pBit << inc++);
}

function readBit ():uint
{
    return uint( (buffer & ( 1 << incRead++ )) != 0 );
}

// outputs : 1
trace( writeBit ( 1 ).toString ( 2 ) );

// outputs : 11
trace( writeBit ( 1 ).toString ( 2 ) );

// outputs : 111
trace( writeBit ( 1 ).toString ( 2 ) );

// outputs : 111
trace( writeBit ( 0 ).toString ( 2 ) );

// outputs : 10111
trace( writeBit ( 1 ).toString ( 2 ) );
```

We can then move those to a specific `BitWriter` class :

```
package org.bytearray.utils
{
    public class BitWriter
    {
        private var buffer:uint = 0;
        private var length:uint = 0;
        private var pointer:uint = 0;

        public function writeBit(pBit:int):uint
        {
            if(0 > pBit || 1 < pBit) throw new Error("Argument should be 0 or 1");
            return buffer |= (pBit << (length = pointer++));
        }

        public function readBit():uint
        {
            if ( pointer > length ) throw new Error ("End of data encountered.");
            return uint( (buffer & (1 << pointer++)) != 0 );
        }

        public function set position (pPosition:int):void
        {
            pointer = pPosition;
        }

        public function get position ():int
        {
            return pointer;
        }
    }
}
```

Anytime we need to store bits, a few lines of code are required :

```
import org.bytearray.utils.BitWriter;

var writer:BitWriter = new BitWriter();

// outputs : 1
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// outputs : 11
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// outputs : 111
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// outputs : 111
trace( writer.writeBit ( 0 ).toString ( 2 ) );

// outputs : 10111
trace( writer.writeBit ( 1 ).toString ( 2 ) );
```

To read the stored bits, we just need to call the `readBit` method :

```
import org.bytearray.utils.BitWriter;

var writer:BitWriter = new BitWriter();

// outputs : 1
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// outputs : 11
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// outputs : 111
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// outputs : 111
trace( writer.writeBit ( 0 ).toString ( 2 ) );

// outputs : 10111
trace( writer.writeBit ( 1 ).toString ( 2 ) );

// reset position
writer.position = 0;

// outputs : 1
trace( writer.readBit() );

// outputs : 1
trace( writer.readBit() );

// outputs : 1
trace( writer.readBit() );

// outputs : 0
trace( writer.readBit() );

// outputs : 1
trace( writer.readBit() );
```

You may wonder why we would require to store bits this way. In some situations we may need to optimize the space being used as much as possible and pack for instance 8 booleans in a byte, where each bit will be worth 0 for `false` and 1 for `true`. This technique allows us for instance, to store 32 states in a single `uint` (32-bit) value instead of creating an Array of `boolean`. Using such a trick can be really useful for data transfer.

Storing button states in a byte for instance can be really powerful for game controllers for instance. In WiiFlash (wiiflash.bytearray.org), each button state of the Wiimote is stored as a single bit. 0 means the button is not pressed and 1 means it is. Through this technique, we are able to represent all the states of the buttons on a Wiimote through a few bits. Here is a part of the C# source code of the WiiFlash Server.

You may notice that bitwise operators remains the same in C# which is also true for many other languages :

```
// create the ByteArray
MemoryStream ms = new MemoryStream(buffer);

// create a byte variable to store the states
byte buttonStateHI = 0;

// create a byte variable to store the states
byte buttonStateLO = 0;

if (ws.ButtonState.One)
    buttonStateHI |= 1 << 7;

if (ws.ButtonState.Two)
    buttonStateHI |= 1 << 6;

if (ws.ButtonState.A)
    buttonStateHI |= 1 << 5;

if (ws.ButtonState.B)
    buttonStateHI |= 1 << 4;

if (ws.ButtonState.Plus)
    buttonStateHI |= 1 << 3;

if (ws.ButtonState.Minus)
    buttonStateHI |= 1 << 2;

if (ws.ButtonState.Home)
    buttonStateHI |= 1 << 1;

if (ws.ButtonState.Up)
    buttonStateHI |= 1;

if (ws.ButtonState.Down)
    buttonStateLO |= 1 << 7;

if (ws.ButtonState.Right)
    buttonStateLO |= 1 << 6;

if (ws.ButtonState.Left)
    buttonStateLO |= 1 << 5;
```

The AND operation is also commonly used to test if a value is even or odd. As we said earlier, an odd number always have its lsb set to 1. Thus, we can test if the bit is set by a simple bit masking operation :

```
// in binary : 100
var value:int = 4;

// checking of the less significant bit (lsb)
// outputs : 0
trace( value & 1 );

// in binary : 101
value = 5;
```

```
// checking of the less significant bit (lsb)
// outputs : 1
trace( value & 1 );
```

As expected, the `isEven` function returns `true` when the value is odd, and `false` when even :

```
// in binary: 100
var value:int = 4;

// checking of the less significant bit (lsb)
// outputs : true
trace( isEven ( value ) );

// in binary : 101
value = 5;

// checking of the less significant bit (lsb)
// outputs : false
trace( isEven ( value ) );

function isEven ( pValue:Number ):Boolean
{
    return ( (pValue & 1) == 0);
}
```

The & operator is also commonly used to cut a set of bits so that they do not exceed a specific maximum value. In order to emulate a CPU, we need to create registers. Let's say that we need to emulate 8-bits registers, if available we may use the `byte` type for that.

In a byte world, by adding 1 to 0xFF, the value should revert to 0 because the overflow would entail the following result :

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

As the `byte` type cannot hold 9 bits, only the 8 lowest bits (lsb) would be preserved, giving us the final value :

```
00000000
```

Either :

```
0
```

Perfect right ? But as we don't have a `byte` or `ubyte` type in ActionScript 3, we generally use the `int` or `uint` types for registers. Keep in mind those are 32-bit integers. Do you see the problem ?

When using such types, adding 1 to 0xFF would just give 256 which would crash our CPU emulation :

```
100000000
```

Thanks to the AND operator, we can make sure we always preserve the 8 less significant bits (lsb). The following figure illustrates the idea :

```
  100000000 Register
&  11111111 Mask
  ---------
   00000000
```

Which gives us the following ActionScript code :

```
// byte : 256
var byte:int = 0x100;

// masking, only the 8 (lsb) bits are preserved
// outputs : 0
trace( byte & 0xFF );
```

More generally, the & operator is also commonly used to extract the components of a color. Colors are generally stored as 24-bit or 32-bit integers where each channel (red, green, blue) is stored on 8-bits. (Hence the concept of 8-bit per component you may see in various files specs).

In the following code, we take a random 24-bit color and extract each component :

```
// color 24 bits
var color:uint = 0xFF9911;

// extraction of the red component
trace( color & 0xFF0000 );

// extraction of the green component
trace( color & 0x00FF00 );

// extraction of the blue component
trace( color & 0x0000FF );
```

With the code above, we just used the & operator to perform a bit masking operation to extract the bits we want to preserve. To understand this first step, let's take a look at the following figure which shows first our color in binary notation :

```
11111111 10011001 00010001
  RED     GREEEN    BLUE
```

Each byte (8-bit) stores an offset, now we can filter that and retrieve each channel by using a simple & (AND) operation with an appropriate mask for each channel.

In the following figure, we retrieve the red channel :

```
  11111111 10011001 00010001 Color
& 11111111 00000000 00000000 Mask
  -------------------------
  11111111 00000000 00000000
```

We now need to shift the 8 bits (red channel) 16 bits to the right to get our final value :

```
// color 24 bits
var color:uint = 0xFF9911;

// extraction of the red component
// outputs : 255
trace( (color & 0xFF0000) >> 16 );
```

For the green we get the following result :

```
  11111111 10011001 00010001 Color
& 00000000 11111111 00000000 Mask
  -------------------------
  00000000 10011001 00000000
```

In this case, we must shift the green bits 8 bits to the right :

```
// 24 bits color
```

```
var color:uint = 0xFF9911;

// extraction of the red component
// outputs : 255
trace( (color & 0xFF0000) >> 16 );

// extraction of the green component
// outputs : 153
trace( (color & 0x00FF00) >> 8 );
```

For the blue channel, no shifting is required as they are already aligned to the right :

```
  01011001 00111111 10101100 Color
& 00000000 00000000 11111111 Mask
  -------------------------
  00000000 00000000 10101100
```

In this case, a simple AND operation is enough :

```
// 24 bits color
var color:uint = 0xFF9911;

// extraction of the red component
// outputs : 255
trace( (color & 0xFF0000) >> 16 );

// extraction of the green component
// outputs : 153
trace( (color & 0x00FF00) >> 8 );

// extraction of the blue component
// outputs : 17
trace( color & 0x0000FF );
```

Thanks to the `toString` method we convert the base 10 to base 16 :

```
// 24 bits color
var color:uint = 0xFF9911;

// extraction of the red component
// outputs : ff
trace( ((color & 0xFF0000) >> 16).toString(16) );

// extraction of the green component
// outputs : 99
trace( ((color & 0x00FF00) >> 8).toString(16) );

// extraction of the blue component
// outputs : 11
trace( (color & 0x0000FF).toString(16) );
```

Sweet, we just extracted correctly our color channels. Now let's discover the bitwise unsigned right shift operator.

In the following code we try to extract the alpha channel from a 32-bit color :

```
// 32-bit color
var color:uint = 0xCCFF9911;

// extraction of the alpha component
// outputs : -34
trace( ((color & 0xFF000000) >> 24).toString(16) );

// extraction of the red component
// outputs : ff
trace( ((color & 0xFF0000) >> 16).toString(16) );
```

```
// extraction of the green component
// outputs : 99
trace( ((color & 0x00FF00) >> 8).toString(16) );

// extraction of the blue component
// outputs : 11
trace( (color & 0x0000FF).toString(16) );
```

As you can see, the value -34 does not reflect the alpha offset, which should return 204 (0xCC). The problem is that we cannot shift the alpha bits 24 bits to the right because this requires a 32-bit space to work, so we need to work with the unsigned right shift operator (>>>) :

```
// 32 bits color
var color:uint = 0xCCFF9911;

// extraction of the alpha component
// outputs : cc
trace( ((color & 0xFF000000) >>> 24).toString(16) );

// extraction of the red component
// outputs : ff
trace( ((color & 0xFF0000) >> 16).toString(16) );

// extraction of the green component
// outputs : 99
trace( ((color & 0x00FF00) >> 8).toString(16) );

// extraction of the blue component
// outputs : 11
trace( (color & 0x0000FF).toString(16) );
```

Conversely, the | (OR) operator can then be also used to create a color. Remember we used that operator previously to detect if a bit was set. In the following code we will create a 32-bit color from 4 components (alpha, red, green and blue) :

```
var transparency:int = 0xFF;
var red:int = 0x88;
var green:int = 0x7E;
var blue:int = 0x11;

var color:uint = transparency << 24 | red << 16 | green << 8 | blue;

// outputs : FF887E11
trace ( color.toString ( 16 ).toUpperCase() ) ;
```

Just to make sure you understand what is happening internally, we will spend a few seconds on what we just did at the binary level :

```
   11111111 00000000 00000000 00000000 <- Transparency (shift 24 bits on the left)

OR          10001000 00000000 00000000 <- Red (shift 16 bits on the left)

OR                   01111110 00000000 <- Green (shift 8 bits on the left)

OR                            00010001 <- Blue (no shift required)
---------------------------------------------------

   11111111 10001000 01111110 00010001 (base 2)
---------------------------------------------------

     0xFF     0x88     0X7E     0x11    (base 16)
```

If we need to change only one component (byte) from the 32-bit color, we can use the OR operator with a simple mask to only modify the bits we are interested into :

```
// color 32 bits
var color:uint = 0xFFAA9911;

color |= 0x00BB0000;

// outputs : ffbb9911
trace( color.toString ( 16 ) );
```

And here is what is happening at the binary level :

```
  11111111 10101010 10001001 00010001 Color
| 00000000 10110111 00000000 00000000 Blue Mask
  ------------------------- --------
  11111111 10111111 10001001 00010001
```

Similarly, if we want to set a color channel to 0, we cannot do this with a single operator, we will need to use the bitwise NOT (~) operator :

```
~expression
```

The NOT operator performs an invert on all the bits :

```
NOT 1011
    ----
    0100
```

As we cannot shift zeroes, we will shift our bits to the left, then revert the whole expression to get the following mask :

```
00000000 11111111 11111111
```

Thanks to this mask we are then able to use it with the AND operator to reset the red offset :

```
  11111111 10101010 10001001 00010001 Color
& 00000000 11111111 11111111 11111111 Red Mask
  ------------------------- --------
  00000000 10101010 10001001 00010001
```

We can then apply our mask as the following code illustrates :

```
var color:uint = 0xFFAA9911;

color = color & ~(0xFF<<16);

// outputs : ff009911
trace( color.toString ( 16 ) );
```

But we can compact the code by using the &= operator :

```
// color 32 bits
var color:uint = 0xFFAA9911;

color &= ~(0xFF<<16);

// outputs : ff009911
trace( color.toString ( 16 ) );
```

We will come back to the assignement operators we just used very soon. Note that the Flash Player in the trace output, does not show the trailing zeros. For instance when trying the code below, we see that the trailing zeros are hidden :

```
var color:uint = 0xFF<<8;

// outputs : 1111111100000000
trace( color.toString ( 2 ) );
```

When working with masks and binary stuff, you can use a little custom function when lost in this binary world, to see for instance in a 32-bit space what is moved and what is the room available for moving the bits.

In the code below we use a simple `convertToBinary` function which does the job for us :

```
function convertToBinary(numberToConvert:Number):String
{
    var result:String = "";
    var lsb:Number;
    const lng :int = 32;
    for (var i:int = 0; i < lng; i++)
    {
        // Extract least significant bit using bitwise AND.
        lsb = numberToConvert & 1;
        // Add this bit to the result.
        result = (lsb == 1 ? "1" : "0") + result;
        // Shift numberToConvert right by one bit, to see next bit.
        numberToConvert >>= 1;
    }
    return result;
}

var color:uint = (0xFF<<8);

// outputs : 00000000000000001111111100000000
trace( convertToBinary ( color ) );
```

As you can see, the output shows the 16-bit remaining at the beginning of the expression. Anytime this behavior is required just call the `convertToBinary` function :

```
var value:int = 128;

// outputs : 00000000000000000000000010000000
trace(convertToBinary(value));
```

Bitwise operators using the equal sign, allows us just like with other operators to execute the operation and assign the result directly to the variable. Instead of writing the following verbose code :

```
var size:int = 8;

// division by 2exp1
size = size >> 1;

// outputs : 4
trace( size );
```

We will write :

```
var size:int = 8;

// division by 2exp1
size >>= 1;

// outputs : 4
trace( size );
```

Similarly, instead of writing :

```
var size:int = 0x100;

size = size & 0xFF;

// outputs : 0
```

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

We will prefer writing :

```
var size:int = 0x100;

size &= 0xFF;

// outputs : 0
trace( size.toString ( 16 ) );
```

Now that you understand the basics, let's discover the `ByteArray` class Which is going to help us writing and reading bytes.

Note

- Bitwise operators converts expressions to integers (unsigned or signed).
- Bitwise operators can offer nice ActionScript speed improvements.
- Bits can be easily used to store states.

The ByteArray class

It is time to implement all the concepts previously discussed with the `flash.utils.ByteArray` class. Even if this class is among one of the most powerful classes of the Flash Player, the most common question related to it, is :

« What is the real power of such a class, what can I do with it ? »

It is hard to describe in a few lines the power of the `ByteArray` class, but let's start with a list of some projects requiring the `ByteArray` class :

- **FC64**. Commodore 64 emulator, http://codeazur.com.br/stuff/fc64_final/ ;
- **Intel8080** : Intel8080 "Space Invaders" CPU Emulation : http://www.bytearray.org/?p=622 ;
- **AlivePDF**. PDF generation library, http://alivepdf.bytearray.org ;
- **FZip**. ZIP library, http://codeazur.com.br/lab/fzip/ ;
- **GIF Player**. GIF player library, http://www.bytearray.org/?p=95 ;
- **GIF Encoder**. GIF encoding library, http://www.bytearray.org/?p=93 ;
- **WiiFlash**. Wii peripherals library, http://wiiflash.bytearray.org ;
- **Encodeurs d'images**. JPEG and PNG encoding classes, http://code.google.com/p/as3corelib/ ;
- **Popforge Audio**. Runtime sound generation library, http://www.popforge.de/.

Without byte access, those projects could not be developed. The power of the `ByteArray` class is the low level data manipulation it allows. In other words, we will be able to manipulate and handle data at the byte level, something never exposed in previous versions of ActionScript (1 and 2).

This may enable us, for instance, to interpret or generate any type of files in ActionScript 3 as soon as we know what we are doing and what is possible in terms of performance. As its name suggests, the `ByteArray` class represents an array of bytes. To express 1K, we will use 1 000 indices (see Figure 23.10) :

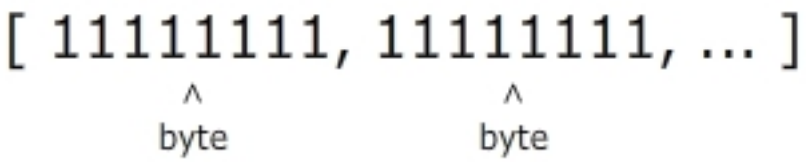

Figure 23.10

Byte Array.

As its name suggests, the `ByteArray` class has some similarities with the `Array` class. Both objects are arrays and thus have a length, but each index in an array of bytes (`ByteArray`) is limited to 8 bits, preventing any number above 255 to be stored in any of the indices.

Note that the same capabilities as the `ByteArray` class are also present in the `flash.net.Socket` and `flash.net.URLSteam` classes.

Writing and reading methods

To create a stream of bytes we use the `ByteArray` class, although we can create an `Array` instance using the following literal notation :

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

A `ByteArray` can only be instanciated through the `new` keyword :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();
```

To retrieve the length of the byte stream, we use its `length` property :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// outputs : 0
trace ( bytes.length );
```

When a byte stream is created, as you would expect, its length returns 0.

> Note that in ActionScript 3, the size of a ByteArray is dynamic and cannot be fixed as in other languages like C# or Java.

As we previously discussed, the byte order may vary according to each platform. The `endian` property defined by the `ByteArray` class allows you to specify the byte ordering.

By default, a `ByteArray` is big-endian in the Flash Player :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// outputs : bigEndian
trace ( bytes.endian );
```

To change the order in writing or reading of bytes we use the `flash.utils.Endian` class constants :

- `Endian.BIG_ENDIAN` : MSB in first position ;
- `Endian.LITTLE_ENDIAN` : LSB in first position ;

In the following code, we check if the byte ordering is big-endian :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// outputs : true
trace( bytes.endian == Endian.BIG_ENDIAN );
```

We can see that the order is `bigEndian` by default, but we can change that easily, by setting the `endian` property :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we will be using little-endian for byte ordering
bytes.endian = Endian.LITTLE_ENDIAN;
```

To store the value 5 as a 32-bit unsigned integer within an instance of the `Array` class, we can write the following code :

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

var value:uint = 5;

buffer[0] = value;

// outputs : 1
trace( buffer.length );

// outputs : 5
trace( buffer[0] );
```

To store the same value with the same datatype in an array of **bytes**, the number should be separated into groups of **bytes**. In this case, for a 32-bit value, we must split the value into 4 bytes :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// manual writing (big-endian)
bytes[0] = (value & 0xFF000000) >> 24;
bytes[1] = (value & 0x00FF0000) >> 16;
bytes[2] = (value & 0x0000FF00) >> 8;
bytes[3] = value & 0xFF;
```

Once stored, we can reconstruct the value by using a few bitwise operators :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// manual writing (big-endian)
bytes[0] = (value & 0xFF000000) >> 24;
bytes[1] = (value & 0x00FF0000) >> 16;
bytes[2] = (value & 0x0000FF00) >> 8;
bytes[3] = value & 0xFF;

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

// outputs : 5
trace( finalValue );
```

To store data in little-endian format, we can simply reverse the order of writing :

```
// we create an array of bytes (ByteArray)
```

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

var value:uint = 5;

// manual writing (little-endian)
bytes[3] = (value & 0xFF000000) >> 24;
bytes[2] = (value & 0x00FF0000) >> 16;
bytes[1] = (value & 0x0000FF00) >> 8;
bytes[0] = value & 0xFF;

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

// outputs : 5
trace( finalValue );
```

Note that we can also achieve the same result as above with the code below :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// manual writing (little-endian)
bytes[3] = (value >> 24) & 0xFF;
bytes[2] = (value >> 16) & 0xFF;
bytes[1] = (value >> 8) & 0xFF;
bytes[0] = value & 0xFF;

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

// outputs : 5
trace( finalValue );
```

As you can see, it is extremely tedious to write data into a `ByteArray` using the previous code. Rest assured, to facilitate our task the Flash Player will automatically manage all that by using the appropriate reading and writing methods. Thus, to write an unsigned integer in a `ByteArray`, we will use the `writeUnsignedInt` method with the folllowing signature :

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

A 32-bit unsigned integer is expected. As a result, the previous tedious code can be rewritten as follows :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);
```

By testing the length of the `ByteArray`, we can see that 4 bytes have been written in the stream of bytes :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

// outputs : 4
trace( bytes.length );
```

To read back the value, we simply use the `readUnsignedInt` method :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

// throws a runtime error
var finalValue:uint = bytes.readUnsignedInt();
```

If we try to run the pevious code, a runtime error is thrown :

```
Error: Error #2030: End of file detected.
```

Keep in mind that when writing data inside a `ByteArray`, an internal pointer, defined by the `position` property, automatically moves to the next position where writing or reading will occur :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// outputs : 0
trace( bytes.position );

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

// outputs : 4
trace( bytes.position );
```

The `position` property is a simple pointer being moved when data is written or read. This is the point at which the next call to a read method starts reading or a write method starts writing.

The following figure (see Figure 23.11) illustrates the idea :

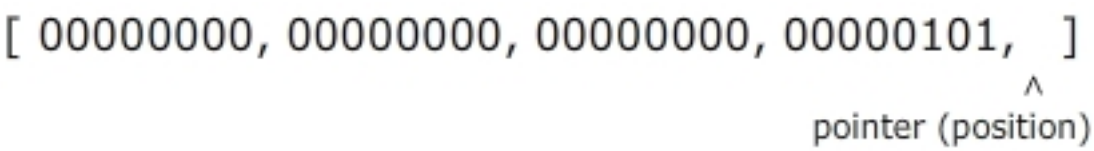

Figure 23.11

Internal pointer (position property).

Always remember that in order to read through the stream, you must first reset the internal pointer to the position where you want to start reading. Once set, you can call the different reading methods :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

// reset the pointer so that we can read the 4 following bytes
```

```
bytes.position = 0;

// reading a 32-bit unsigned integer
var finalValue:uint = bytes.readUnsignedInt();

// outputs : 5
trace( finalValue );
```

Keep in mind that just like with any other API's, to prevent any runtime error to occur, we can place the different reading methods calls within a `try catch` statement :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

try
{
    // throws a runtime error
    trace( bytes.readUnsignedInt() );

} catch ( e:Error )
{
    trace ("Error while reading.");
}
```

Note that in the previous code, we used again the `uint` type to store the result of the `readUnsignedInt` method. You may have heard that the `int` type is preferred sometimes for performance reasons, especially for mathematical operators or more simply inside loops. In the context of binary data manipulation, you should **always** use the most appropriate type related to the reading and writing methods. As a result, we will use the `uint` type when using methods like `writeUnsignedInt` and `readUnsignedInt`.

If we do not respect this, we may get wrong result. The following example illustrates this situation. We store a 32-bit unsigned integer with a value of 3 000 000 000 in a `ByteArray`. By storing the result of the `readUnsignedInt` method within a variable of type `int`, we end up with a wrong result :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 3000000000;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

bytes.position = 0;

// reading a 32-bit unsigned integer
var finalValue:int = bytes.readUnsignedInt();

// AVM runtime conversion
// outputs : -1294967296
trace( finalValue );
```

The unsigned integer returned by the method `readUnsignedInt` cannot been stored properly using the `int` type which cannot handle an integer greater than 2 147 483 647.

Using the `uint` type, our value is properly stored :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 3000000000;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

bytes.position = 0;

// reading a 32-bit unsigned integer
var finalValue:uint = bytes.readUnsignedInt();

// AVM runtime conversion
// outputs : 3000000000
trace( finalValue );
```

By looking at the Figure 23.12, we remember that a 32-bit unsigned integer occupies 4 bytes which are allocated by the VM for the `uint` type :

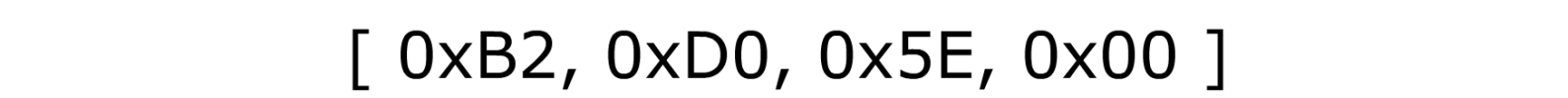

Figure 23.12

32-bit (4-bytes) unsigned integer stored in a ByteArray.

Have you noticed that Figure 23.12 uses the hexadecimal notation to simplify the representation of such a number ?

We will now always use hexadecimal notation to simplify data representation.

Therefore, we can express such a value by using the hexadecimal notation Inside our ActionScript code :

```
var value:uint = 0xB2D05E00;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);
```

Very easily, we can store the value 3000000000 in little-endian :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

bytes.endian = Endian.LITTLE_ENDIAN;

var value:uint = 0xB2D05E00;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);
```

Bytes are written using the little-endian ordering, we can confirm that by reading each byte one by one :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

bytes.endian = Endian.LITTLE_ENDIAN;

var value:uint = 0xB2D05E00;

// we write a 32-bit unsigned integer
```

```
bytes.writeUnsignedInt(value);

bytes.position = 0;

// outputs : 0
trace( bytes.readUnsignedByte().toString( 16 ) );

// outputs : 5E
trace( bytes.readUnsignedByte().toString( 16 ) );

//outputs : D0
trace( bytes.readUnsignedByte().toString( 16 ) );

//outputs : B2
trace( bytes.readUnsignedByte().toString( 16 ) );
```

Of course, by using the readings methods of the `ByteArray` class, the conversion is transparent and done automatically :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

bytes.endian = Endian.LITTLE_ENDIAN;

var value:uint = 0xB2D05E00;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

bytes.position = 0;

// outputs : b2d05e00
trace( bytes.readUnsignedInt().toString ( 16 ) );
```

We will get back to the concept of byte ordering when creating encoders or parsers. Similarly when emulating a processor, respecting the endianness of the CPU is essential.

In some situations, it can be faster doing the conversion manually. In the following code, we read a 16-bit integer from memory by calling the `readShort` method which handles endianness automatically.

This forces us to set the `position` property so that we read the appropriate data from the stream :

```
protected function readShort(address:int):int
{
    // set the pointer to access memory
    memory.position = inAddress;
    // return the short (16-bit integer)
    return memory.readShort();
}
```

We can optimize the previous code, by removing the `position` property call and access the memory index directly and swap the bytes manually :

```
protected function readShort(address:int):int
{
    // return the short (16-bit integer)
    return memory[int(address+1)] << 8 | memory[address];
}
```

This version is compact and will perform faster. We manually shift the LSB to the left, to return a little-endian short.

Now if we store an unsigned integer with a value of 5 using the `writeUnsignedInt` method, remember that only 3 bits are really required :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a 32-bit unsigned integer
bytes.writeUnsignedInt(value);

bytes.position = 0;

// reading a 32-bit unsigned integer
var finalValue:uint = bytes.readUnsignedInt();

// outputs : 5
trace( finalValue );
```

We will be wasting 3 bytes here, as the following figure illustrates :

Figure 23.13

The value 5 stored in a Byte Array in a 32-bit space.

As you can imagine, such a value can fit in a single byte. We will be using the `writeByte` method for that :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a single byte
bytes.writeByte(value);
```

The byte is placed at index 0, Figure 23.14 illustrates the byte in the `ByteArray` in hexadecimal notation :

Figure 23.14

A byte stored at the index 0.

To read the value back, we will be using as you may have guessed, the `readUnsignedByte` method :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 5;

// we write a single byte
bytes.writeByte(value);

bytes.position = 0;

// reading a 32-bit unsigned integer
```

```
var finalValue:uint = bytes.readUnsignedByte();

// outputs : 5
trace( finalValue );
```

As a byte is not be able to hold more than 8 bits. If we try to write a value exceeding one byte in storage, we will end up with a byte overflow. As a result, only the 8 lsb will be written into the stream.

In the following code we convert the decimal value 260 to binary :

```
var value:uint = 260;

// outputs : 100000100
trace ( value.toString( 2 ) );
```

As we can see, we need 9 bits to hold the value 260, which gives us the following representation in binary :

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

As discussed previously, each index of a `ByteArray` cannot hold more than 8 bits. Thus, when we try to write a value requiring more than 8 bits, only the 8 lsb will be written into the index.

> Remember, the less significant bits (lsb) are located on the right. The most significant bits (msb) are located on the left.

The folllowing code illustrates the scenario :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:uint = 260;

// we try to store the integer (260) into a single byte (8-bits)
bytes.writeByte(value);

bytes.position = 0;

// we read the byte
// outputs : 4
trace ( bytes.readUnsignedByte() );
```

By converting the unsigned integer 4 to binary notation :

```
// we read the byte and show its value in binary base
// outputs : 100
trace ( bytes.readUnsignedByte().toString(2) );
```

Which is simply our 3 lsb :

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

As we have seen previously, a byte can hold a value ranging from 0 to 255 when it is said unsigned. An signed byte, on its hand can contain an integer ranging from -128 to 127. So if we want to sore a signed integer with a value of -80, we will use the most appropriate method called `readByte` :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:int = -80;

// we store the integer (-80) into a single byte
bytes.writeByte(value);

bytes.position = 0;

// we read the byte
// outputs : -80
trace( bytes.readByte() );
```

We can then ask ourselves what is the best type to use when we using methods like `writeByte`, `readByte`, or even `readUnsignedByte`. We saw previously that it was recommended to use the `uint` type when using methods and `writeUnsignedInt` `readUnsignedInt`. Unfortunately, there is no `byte` or `ubyte` type in ActionScript 3. As a result, we will use the `int` type instead which can easily store any `byte` or `ubyte` value.

We have been working with 32-bit integers since the beginning of this chapter. If we store a value greater than 255, we must work with two bytes. Consider the 16-bit integer 1200 in its binary form :

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

To store such a value, we will use the `writeShort` method with the folllowing signature :

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

As you can imagine it would make more sense having such a signature :

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

Unfortunately, such a type is not available in ActionScript 3, we will then use the `int` type when dealing with short values inside the `ByteArray`.

The `writeShort` method allows us to write 16-bit integers into the stream of bytes. A pair of two bytes is generally known as a « word » in the binary world, it can contain a value ranging from -32 768 to 32 767 or 0 to 65 535 if it is said unsigned.

> Note that the `ByteArray` class does not expose any API's to handle unsigned short types. As a result, you won't find any `writeUnsignedShort` or `readUnsignedShort` methods.

In the following code, we store a 16-bit integer into the `ByteArray` :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:int = 1200;

// we try to store the integer (1200) into a word (16-bits)
bytes.writeShort(value);

// number of bytes written
// outputs : 2
trace( bytes.length );
```

By testing the code above we see that two bytes are written into the stream. To read back the stored value we simply use the `readShort` method :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:int = 1200;

// we store the unsigned integer (1200) into a word (16-bits)
bytes.writeShort(value);

bytes.position = 0;

// we read a 16-bit signed integer
// outputs : 1200
trace( bytes.readShort() );
```

If we analyze the binary representation of the integer 1 200 we obtain the following representation:

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

Such a value requires a 16-bit space. If we call the method `readUnsignedByte` successively, we can retrieve the whole value byte by byte :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:int = 1200;

// we store the unsigned integer (1200) into a word (16-bits)
bytes.writeShort(1200);

bytes.position = 0;

// we read the first byte
var firstByte:int = bytes.readUnsignedByte();

// we read the second byte
var secondByte:int = bytes.readUnsignedByte();

// outputs : 4
trace( firstByte );

// outputs : 176
trace( secondByte );

// outputs : 1200
trace ( firstByte << 8 | secondByte );
```

The first byte will hold the value 4 :

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

And the second one the value 176 :

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

Now if we want to store a floating point number in a ByteArray, ActionScript 3 uses the standard IEEE 754 to handle floating point numbers through the `writeFloat` method :

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

Here again, you may notice that the `float` type is not used in the signature of the `writeFloat` method, such a signature would have been sweeter :

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

Unfortunately, such a type is not handled by the AVM2 used in ActionScript 3. We will simply use the `Number` type (which is a 64-bit datatype) to store floats.

In the following code, we store a 32-bit float inside the `ByteArray` :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:Number = 12.5;

// we write a 32-bit float
bytes.writeFloat(value);

// set the pointer to 0
bytes.position = 0;

// we read the float
var float:Number = bytes.readFloat();

// outputs : 12.5
trace( float );
```

As a `float` requires 32-bit, 4 bytes are required and stored inside the stream of bytes :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var value:Number = 12.5;

// we write a 32-bit float
bytes.writeFloat(value);

// outputs : 4
trace( bytes.length );
```

If we want to store a floating point number requiring more than 32 bits, we will be using the `writeDouble` method :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write a 64-bit float
bytes.writeDouble(12.5);

// outputs : 8
trace( bytes.length );

bytes.position = 0;

// outputs : 12.5
trace( bytes.readDouble() );
```

Note that a *double* is the equivalent of type `Number` and uses 64-bit. ECMAScript 4 defines a *double*, then we might see such a type in ActionScript 3 one day.

Note

- The ByteArray class defines a set of methods allowing developers to write different types of numeric types.
- Some data types handled by the ByteArray API's does not have any equivalents types in the AVM2. This is true for `float` (writeFloat), `byte` (writeByte), `ubyte` (readUnsignedByte) and `short` (writeShort).
- The writing methods automatically divides values in groups of bytes.
- The reading methods automatically merges groups of bytes as a final value.

Writing string to the byte stream

We will see now how text can be stored inside a stream of bytes. To write text data inside a `ByteArray`, we can use the `writeUTFBytes` method with the following signature :

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

Each character code is simply written into the `ByteArray`, in the following code, we write a simple sentence :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var chars:String = "Bonjour Bob";

// outputs : 11
trace( chars.length );

// we write the string into the stream
bytes.writeUTFBytes(chars);

// outputs : 11
trace( bytes.length );
```

If we use the first 127 characters of the ASCII codes, we see that each character is coded on one byte. Using the method `readByte`, we can retrieve each character code manually :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var chars:String = "Hey there!";

// outputs : 10
trace( chars.length );

// we write the string into the stream
bytes.writeUTFBytes(chars);

// outputs : 10
trace( bytes.length );

bytes.position = 0;

// outputs : 72
trace( bytes.readByte() );

// outputs : 101
trace( bytes.readByte() );
```

By sending those character codes to the `fromCharCode` method of the `String` class, we discover each character :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

var chars:String = "Hey there!";

// outputs : 10
trace( chars.length );

// we write the string into the stream
bytes.writeUTFBytes(chars);

// outputs : 10
trace( bytes.length );

bytes.position = 0;

// outputs : H
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : e
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : y
trace( String.fromCharCode( bytes.readByte() ) );

// outputs :
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : t
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : h
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : e
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : r
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : e
trace( String.fromCharCode( bytes.readByte() ) );

// outputs : !
trace( String.fromCharCode( bytes.readByte() ) );
```

Conversely, if we use special characters, they will be multi-byte. In the following code, we use the « ç » character from the French alphabet :

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

var chars:String = "Bonjour Bob ça va ?";

// outputs : 19
trace( chars.length );

bytes.writeUTFBytes(chars);

// outputs : 20
trace( bytes.length );
```

Such a character will be encoded on two bytes. Another method is available to write strings in a `ByteArray`, this method is called `writeUTF` :

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

This method is almost similar to `writeUTFBytes`, the only difference is that the string will be preceded by a 16-bit integer indicating the length of the string. Let's take a simple example, in the following code we write the single character « b » through the `writeUTF` method:

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write a single character
bytes.writeUTF("b");

// outputs : 3
trace( bytes.length );
```

If we had used the `writeUTFBytes` method, the length would have been 1. By using the `writeUTF` method, we have 2 more bytes (16-bit integer) which indicates the length of the string. We can manually retrieve that value with the `readShort` method :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write a single character
bytes.writeUTF("b");

// outputs : 3
trace( bytes.length );

bytes.position = 0;

// outputs : 1
trace( bytes.readShort() );
```

The length returns 1 which means that the string stored right after the length is stored on a single byte :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write a single character
bytes.writeUTF("b");

// outputs : 3
trace( bytes.length );

bytes.position = 0;

// outputs : 1
trace( bytes.readShort() );

// we retrieve the char code
var characterCode:uint = bytes.readUnsignedByte();

// outputs : b
trace( String.fromCharCode( characterCode ) );
```

As you can imagine, we will not be using this approach to retrieve text stored inside a `ByteArray`. Let's cover now the API's available to translate character codes to readable strings.

Reading string from the byte stream

As we have seen previously, keep in mind setting the `position` pointer accurately, to prevent any end of file runtime error. Remember, what happened previously when we did not handle the position property properly :

```
Error: Error #2030: End of file detected.
```

To track the number of bytes available for read, we will always use the `bytesAvailable` property. Internally, the Flash Player is subtracting the `position` pointer value to the total length of the `ByteArray` to compute the number of bytes available :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write the string into the stream
bytes.writeUTFBytes("Bob Groove");

// we set the pointer to 0
bytes.position = 0;

// compute manually the bytes available
// outputs : 10
trace( bytes.length - bytes.position );

// outputs : 10
trace( bytes.bytesAvailable );
```

To extract the text stored in the stream, we use the `readUTFBytes` method, with the following signature :

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

The method expects the number of bytes to read, each character is decoded internally to construct a final UTF-8 string. Thus in the following code, we read only the first 3 characters of the stream :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write the string into the stream
bytes.writeUTFBytes("Bob Groove");

// we set the pointer to 0
bytes.position = 0;

// we extract the three first characters from the stream
// outputs : Bob
trace( bytes.readUTFBytes( 3 ) );
```

We can therefore use the `bytesAvailable` property to be sure to read the full text stored in the stream :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write the string into the stream
bytes.writeUTFBytes("Bob Groove");

// we set the pointer to 0
bytes.position = 0;

// we extract the entire string from the stream
```

```
// outputs : Bob Groove
trace( bytes.readUTFBytes( bytes.bytesAvailable ) );
```

We usually use the property `bytesAvailable` with a `while` loop to extract each character:

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write the string into the stream
bytes.writeUTFBytes("Bob Groove");

// we set the pointer to 0
bytes.position = 0;

while ( bytes.bytesAvailable > 0 )
{
    /* outputs :
    0 : B
    1 : o
    2 : b
    3 :
    4 : G
    5 : r
    6 : o
    7 : o
    8 : v
    9 : e
    */
    trace( String.fromCharCode( bytes.readByte()) );
}
```

To map the character code to the appropriate character, we use the `fromCharCode` method of the `String` class. If we insert special characters, calling the method `readUTFBytes` characters correctly decode multi-byte :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write the string into the Stream including spécial characters
bytes.writeUTFBytes("Bob ça Groove");

// we set the pointer to 0
bytes.position = 0;

// we extract the entire string from the stream
// outputs : Bob ça Groove
trace( bytes.readUTFBytes( bytes.bytesAvailable ) );
```

Conversely, if we use the `readByte` method to decode each character, we cannot decode multi-byte characters :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write the string into the Stream including spécial characters
bytes.writeUTFBytes("Bob ça Groove");

// we set the pointer to 0
bytes.position = 0;

while ( bytes.bytesAvailable > 0 )
{
    /*
    0 : B
    1 : o
    2 : b
```

```
    3 :
    4 : □
    5 : □
    6 : a
    7 :
    8 : G
    9 : r
    10 : o
    11 : o
    12 : v
    13 : e
    */
    trace ( bytes.position + " : " + String.fromCharCode( bytes.readByte() ) )
}
```

As you can imagine, the `readUTFBytes` comes very handy when reading strings encoded with the `writeUTF` method. Now, let's say that we don't know the length of the string, such a method is intented to be used with `readUTFBytes` which is expecting the length of the string to be read, we will come back to such a method riht after this section :

```
// we create an array of bytes (ByteArray)
var bytes:ByteArray = new ByteArray();

// we write a string
bytes.writeUTF("Hello there!");

// outputs : 14
trace( bytes.length );

bytes.position = 0;

// we retrieve the length of the String following in the stream
var stringLength:uint = bytes.readShort();

// outputs : Hello there!
trace( bytes.readUTFBytes( stringLength ) );
```

This ends our first chapter through the binary world in Flash. In the next chapter we will see how to use the ByteArray class in everyday use as a Flash developer with real world cases, like embedding assets and shared libraries as ByteArray's, XML runtime compression, and much more.

Note

- In order to decode a set of bytes as a UTF-8 string, we use the `readUTFBytes` method.
- To take in account character sets encoding, we can use the `writeMultiByte` method.