SOBRE UNICODE

Seguramente ya habrás estudiado que es Unicode en otros módulos, aquí simplemente incluimos unas pinceladas:

- Hay que distinguir entre **Unicode** y **codificaciones unicode**. Unicode es más abstracto que las codificaciones Unicode ya que simplemente asocia un carácter a un entero (no a bits).
- Este entero no se representa normalmente en decimal si no con 4 dígitos hexadecimales de la forma u+0000 a u+FFFF.
- La especificación original de unicode es de 16 bits pero luego se hizo una ampliación a 32 bits (veremos por ahí caracteres con hasta 8 dígitos hexadecimales). La extensión de 32 bits es necesaria para idiomas no occidentales.
- Como se hace corresponder el entero unicode a bits depende del esquema de codificación que interese utilizar, por curiosidad hay los siguientes esquemas de codificación: UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE
- cada esquema de codificación se aplica en un contexto por razones de eficiencia y otras razones, por ejemplo utf-8 normalmente para ficheros de disco, utf-16 para ram …
- cada esquema tiene sus características de almacenamiento, por ejemplo: utf-8 es de longitud variable y para los primeros 128 caracteres ascii utiliza sólo un byte pero para los siguientes 128 en lo que está la ñ, los acentos, ...utiliza dos bytes y para otras idiomas con caracteres raros no occidentales utiliza 3 y 4 bytes . Por tanto, aunque inicialmente el "8" se refería a 8 bits(1 byte) de almacenamiento, el código evolucionó y un carácter se puede almacenar con 1,2,3 o 4 bytes.

Para ver un buen ejemplo de relación entre Unicode y sus codificaciones consulta

https://www.w3.org/International/articles/definitions-characters/index.es#charsets

FLUJOS DE I/O

Stream vs Stream I/O

Al aparecer en el JDK el API stream de programación funcional tenemos el término stream duplicado en dos API:

- En el api de programación funcional
- En el api de E/S de toda la vida

Aunque se puede establecer algún paralelismo conceptual entre ambos ya que en ambos casos se refieren a una "secuencia que fluye", son conceptos diferentes para contextos diferentes.

Por tanto, AHORA, en este boletín, FLUJO(STREAM) SE REFIERE A LOS DATOS DE E/S. Es decir se refiere a una serie de mecanismos para que la cpu se comunique con el exterior. Ejemplo típico: para leer/escribir en un fichero en disco se usan flujos. Por tanto, para evitar

confusiones con "stream funcional", no olvides añadir el término I/O si te quieres referir a un flujo de entrada salida de forma inequívoca-

Los flujos I/O en Java

Cada lenguaje de programación busca la forma de ocultar detalles hardware para hacer operaciones de E/S. Como podríamos esperar, en Java se accede a la E/S a través de objetos. Estos objetos hacen de intermediarios entre el hardware y nuestros programas. Los objetos son como siempre instancias de clases. Hay un montón de clases para la E/S y se organizan, como no, en jerarquías. Es común a todas las clases de E/S hacernos ver los datos de E/S como un flujo("stream"), es decir, como una tira de bits. Aunque todas las clases comparten este concepto, luego cada clase ofrece sus métodos y es apropiada para un contexto específico. Por ejemplo, las clases que trabajan con flujos de una conexión de red y las que trabajan con ficheros de disco, tendrán cosas en común pero también peculiaridades específicas. La E/S es un tema complejo y sólo se aclara a base de practicar y utilizar las clases.

Aquí la palabra *flujo* la usamos para referirnos:

- Al objeto que hace la E/S
- A la clase o molde del objeto que hace la E/S
- A la fila de bits que manipulan los objetos de E/S

Tienes que estar atento al contexto para interpretar correctamente a que se refiere la palabra "flujo"

FLUJOS DE BYTES Y FLUJOS DE CARACTERES

Los flujos se pueden clasificar por distintos tipos de vista, por ejemplo, por su origen/destino tendremos:

- sockets para referirnos a los flujos de conexiones de red.
- flujos de ficheros para referirnos a los flujos que se originan con los ficheros del disco
- flujos de E/S Standard para referirnos a los flujos con los dispositivos de E/S Standard, normalmente teclado y pantalla respectivamente.
- Etc.

Por la dirección del flujo:

- Flujo de entrada
- Flujo de salida

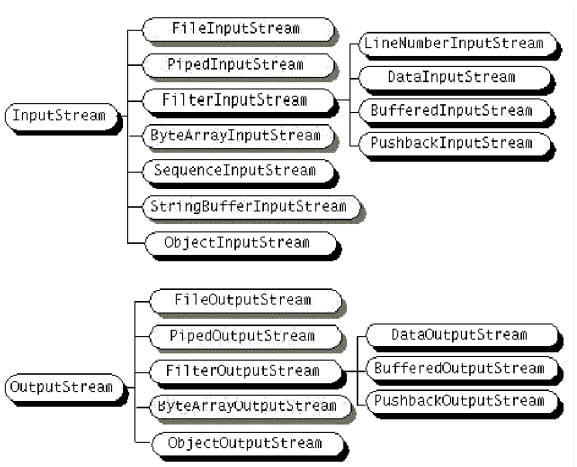
Pero a raíz de cómo se clasifican las clases de E/S en java la clasificación más esencial es la que divide a los flujos en dos tipos:

- flujos de bytes =>visualizamos el flujo organizado como una secuencia de bytes
- flujos de caracteres. =>visualizamos el flujo organizado como una secuencia de caracteres

Recuerda que un byte no es un carácter. En codificaciones como UTF-8 un carácter puede representarse con varios bytes.

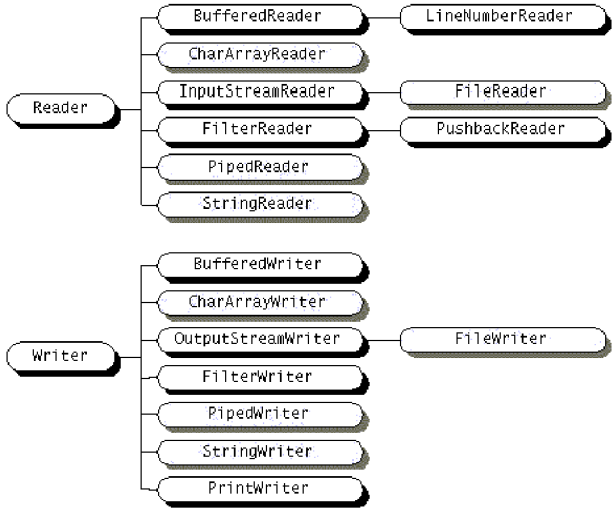
La biblioteca standard nos permite ver los datos de la E/S con la visión en bytes o en caracteres y usaremos las que nos convenga. Para usar una u otra visión usaremos bien clases de flujos orientados a bytes o bien clases de flujos orientados a caracteres.

La jerarquía de clases para trabajar con flujos de bytes es la siguiente



Observa que hay dos jerarquías de flujos de bytes. Una para flujos de entrada (con InputStream como superclase) y otra para flujos de salida entrada (con OutputStream como superclase) .

La jerarquía para trabajar con flujos de caracteres es la siguiente:



Un ejemplo en el que se prefieren los flujos de caracteres.

Los flujos de caracteres se utilizan para trabajar con texto. También es posible trabajar con texto con flujos de bytes pero para trabajar con texto se prefieren los de caracteres por dos ventajas:

- Ya sabes que un carácter se puede almacenar con 1, 2, 3 o 4 bytes, dependiendo de qué tipo de codificación se use (ANSI, utf-8, utf-16, …) y de qué carácter se trate. Si trabajamos con flujos de bytes nuestro código tiene que tener en cuenta esto para obtener un carácter a partir de 1 o varios bytes. Tendremos un código a base de *ifs* y *switchs* complicados y tenemos que conocer muy bien las normas de codificación. Este código sólo valdrá para una codificación concreta. En cambio, si usamos flujos de

caracteres, sólo tendremos que indicar al flujo que codificación debe usar para leer y escribir y él sólo se enfrenta a como recuperar un 1 carácter a partir de un conjunto de bytes. Esto también tiene que ver con la internalización de código. Es decir, para crear programas que se pueden adaptar fácilmente a diversas lenguas. Este es un tema complejo que se sale de nuestro alcance.

- Las clases que trabajan con flujos de carácter tienen métodos "cómodos" para trabajar con texto como readLine()(método que lee una línea completa) que no existen en las clases que trabajan con flujos de bytes.

Supongamos que tenemos un fichero de texto "texto.txt"que almacena la palabra "niño" en UTF-8. Si consultamos una tabla UTF-8 observamos que todos los caracteres se almacenan en 1 byte excepto la ñ que ocupa dos

0x6E	0x69	0xC3 0xB1	0x6F
n	i	ñ	o

Ahora supongamos que quiero leer esa información e imprimir la palabra por pantalla:

- Con flujos de caracteres(FileReader en ejemplo), usaré métodos que proporcionan el siguiente carácter. Sólo tengo que concatenar los caracteres e imprimirlos.

```
import java.io.FileReader;
import java.io.IOException;

public class App {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("texto.txt");

        // Leer el primer caracter
        int caracter1 = fr.read();
        System.out.print((char) caracter1);

        // Leer el segundo caracter
        int caracter2 = fr.read();
        System.out.print((char) caracter2);

        // Leer el tercer caracter
        int caracter3 = fr.read();
        System.out.print((char) caracter3);

        // Leer el cuarto caracter
        int caracter4 = fr.read();
        System.out.print((char) caracter4);

        fr.close();
    }
}
```

- Con flujos de bytes, usaré métodos que me devuelven un byte.

```
import java.io.FileInputStream;
import java.io.IOException;

public class App {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("texto.txt");

        // Leer el primer byte
        int byte1 = fis.read();
        System.out.println("Byte 1: " + byte1);

        // Leer el segundo byte
        int byte2 = fis.read();
        System.out.println("Byte 2: " + byte2);

        // Leer el tercer byte
        int byte3 = fis.read();
        System.out.println("Byte 3: " + byte3);

        // Leer el cuarto byte
        int byte4 = fis.read();
        System.out.println("Byte 4: " + byte4);

        // Leer el quinto byte
        int byte5 = fis.read();
        System.out.println("Byte 5: " + byte5);

        fis.close();
    }
}
```

Pero lo que quería era imprimir *niño* así que le hago un (char) a cada byte

```
import java.io.FileInputStream;
import java.io.IOException;

public class App {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("texto.txt");

        // Leer el primer byte
        int byte1 = fis.read();
        System.out.println("Byte 1: " + (char) byte1);

        // Leer el segundo byte
        int byte2 = fis.read();
```

```
        System.out.println("Byte 2: " + (char) byte2);

        // Leer el tercer byte
        int byte3 = fis.read();
        System.out.println("Byte 3: " + (char) byte3);

        // Leer el cuarto byte
        int byte4 = fis.read();
        System.out.println("Byte 4: " + (char) byte4);

        // Leer el quinto byte
        int byte5 = fis.read();
        System.out.println("Byte 5: " + (char) byte5);

        fis.close();
    }
}
```

La impresión falla. El problema es que interpreto mal la ñ ya que realmente imprimo el 3 y 4 byte por separado sin interpretar ambos como una ñ

Para imprimir la ñ tengo que interpretar el byte 3 y 4 como un único caracter pero no llega con ensamblarlos uno a continuación, hay que meter ifs que contemplen las normas de codificación UTF-8. ¡vaya rollo!. Efectivamente compensa para este casos usar las clases de flujos de caracteres.

Un ejemplo en el que se prefiere el flujo de bytes

Cuando fichero no es de texto le llamamos binario, aunque realmente los de texto también son binarios pero como tienen una biblioteca "extra" que trabaja por encima de los bytes se hace está clasificación entre binario(no de texto) y de texto.

Cuando un fichero no es de texto, no tiene sentido usar flujos de caracteres. Por ejemplo un fichero bitmap que como no es de texto le llamamos genéricamente "binario" tiene la siguiente estructura
https://es.wikipedia.org/wiki/Windows_bitmap

La estructura anterior puedo verla como una secuencia de bytes pero no como una secuencia de caracteres. Leer de byte en byte es un engorro de bajo nivel pero si no tengo una biblioteca que me proporcione métodos específicos siempre puedo tratar el fichero correctamente leyendo de byte en byte.

Si quiero saber el alto y ancho en píxeles del bmp no tengo más remedio que desplazarme secuencialmente por los bytes del fichero hasta llegar a los que me indica el alto y ancho. Nada se puede hacer aquí con los flujos de caracteres que solo entienden códigos Unicode, Ansi, etc. pero desconocen absolutamente la estructura de un bmp.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
```

```java
class App {
    public static void main(String[] args) throws IOException {
        File archivo = new File("unaimagen.bmp");
        FileInputStream fis = new FileInputStream(archivo);

        // Saltar los primeros 18 bytes, ya que no contienen información sobre la anchura y altura de la imagen
        for (int i = 0; i < 18; i++) {
            fis.read();
        }

        // Leer los siguientes 8 bytes que contienen la anchura y altura de la imagen
        byte[] bytesAnchuraYAltura = new byte[8];
        for (int i = 0; i < 8; i++) {
            bytesAnchuraYAltura[i] = (byte) fis.read();
        }

        int anchura = byteArrayToInt(bytesAnchuraYAltura, 0);
        int altura = byteArrayToInt(bytesAnchuraYAltura, 4);

        System.out.println("La anchura de la imagen BMP es: " + anchura + " píxeles");
        System.out.println("La altura de la imagen BMP es: " + altura + " píxeles");

        fis.close();
    }

    public static int byteArrayToInt(byte[] bytes, int offset) {
        // Convertir un 4 bytes de un array de bytes a un valor entero
        return (bytes[offset + 3] & 0xFF) << 24 | (bytes[offset + 2] & 0xFF) << 16 | (bytes[offset + 1] & 0xFF) <<
8 | (bytes[offset] & 0xFF);
    }
}
```

La sentencia try-with-resources

Finally para cerrar flujos

Es una buena práctica, que se puede considerar una obligación del programador, cerrar todos los flujos que se abren.

Abrir un flujo => crear un nuevo flujo por ejemplo hacer new FileInputStream(archivo)

Cerrar un flujo =>ejecutar sobre el objeto flujo abierto el método close().

Para asegurarse que el close() ocurre, la estructura típica de un programa que trabaja con ficheros pasa por incluir en un finally el close().

Por ejemplo:

El siguiente código copia un fichero xanadu.txt en otro outagain.txt. Observa que aunque los ficheros son de texto usamos flujo de bytes la copia es OK porque lo único que hace este programa es copiar de byte en byte, no tiene que ensamblarlos para obtener caracteres para por ejemplo imprimir caracteres por pantalla como en otros ejemplos.

El código es tremendo. Los métodos de las clases de E/S suelen lanzar muchas excepciones y está todo lleno de llaves. Vaya lío. Pero es como se hacía antes del JDK 7. Especialmente pesado es el método close() que también lanza IOEXCEPTION, y decimos que es especialmente pesado porque siempre se debe cerrar un flujo que se abre.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class App{
   public static void main(String[] args){

      FileInputStream in = null;
      FileOutputStream out = null;
      try {
         in = new FileInputStream("xanadu.txt");
         out = new FileOutputStream("outagain.txt");
         int c;

         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } catch (FileNotFoundException ex) {
         System.out.println("o bien xanadu.txt no existe o bien outagain.txt es un directorio");
      } catch (IOException ex) {
          System.out.println("error de E/S al leer o grabar");
      } finally {
         if (in != null) {
            try {
               in.close();
            } catch (IOException ex) {
               System.out.println("error al cerrar xanadu.txt");
            }
```

```
        }
        if (out != null) {
            try {
                out.close();
            } catch (IOException ex) {
                System.out.println("error al cerrar outagain.txt");
            }
        }
    }
  }
}
```

Ejemplo anterior utilizando *try-with-resources.* *Chao a finally y close*

Fíjate cómo se complica el Finally. El problema es que close() puede lanzar una IOException y tengo que tratarla. Siempre que haga un close() va a ocurrir esto. Es un trabajo mecánico y por eso es evitable con try-with-resources(try con recursos).

¿Qué es un *resource* (recurso)?

Un recurso es un flujo, una conexión de BD, o cualquier cosa que se deba cerrar (close()) para liberar recursos del sistema. En el siguiente código después del try hay unos paréntesis en los que se declaran los recursos que quiero cerrar automáticamente al abandonar el try.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;


public class App{
   public static void main(String[] args){
      try( FileInputStream in = new FileInputStream("xanadu.txt");
           FileOutputStream out = new FileOutputStream("outagain.txt");) {

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } catch (FileNotFoundException ex) {
         System.out.println("o bien xanadu.txt no existe o bien outagain.txt es un directorio");
      } catch (IOException ex) {
         System.out.println("error de E/S al leer o grabar");
      }
   }
}
```

ADVERTENCIA IMPORTANTE PARA EL RESTO DE LOS BOLETINES

Con flujos de E/S el cierre de flujos y el tratamiento de excepciones es un tema muy importante para una App profesional pero genera mucho código repetitivo. Nosotros por

razones didácticas a menudo para trabajar con ejemplos de código más limpio y más conciso:

1. "Nos libramos del tratamiento de excepciones " haciendo por ejemplo un throws Exception en el main lo que insistimos no es una buena práctica para una App real
2. Los close() normalmente los haremos en un try-with-resources pero incluso a veces no lo usamos para ganar concisión