



# I/O Files

Capítulo 10  
Modulo 4

# File

Un archivo (file) es un recurso computacional que permite almacenar información en un dispositivo de almacenamiento.

# Ventajas de Manejar Archivos

---

Permite almacenar información permanentemente.

---

Reutilizar la misma información en distintas computadoras o programas.

---

Permite trabajar con grandes cantidades de información convenientemente.

---

Facilita la transferencia de información

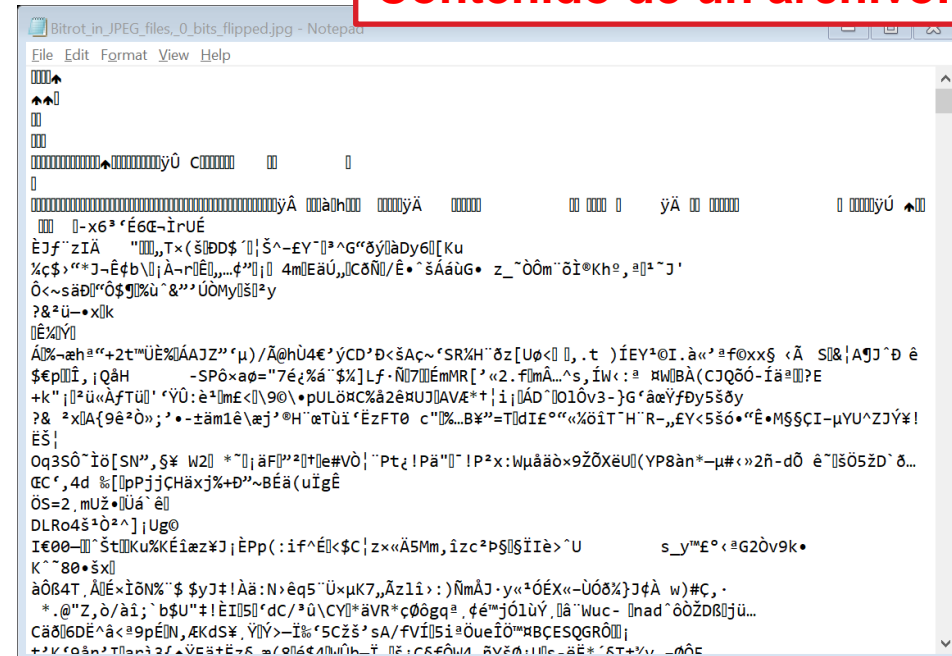
# Recordemos las bases

Recordemos que las computadoras no pueden almacenar “letras”, “números”, “fotos” o “videos”.

Lo único que puede almacenar una computadora son *bits*, es decir, 0's y 1's.



Contenido de un archivo.



# ¿Cómo se interpretan los bits?

Para entender cómo se representa cualquier objeto, necesitamos definir reglas, o un ***encoding***.

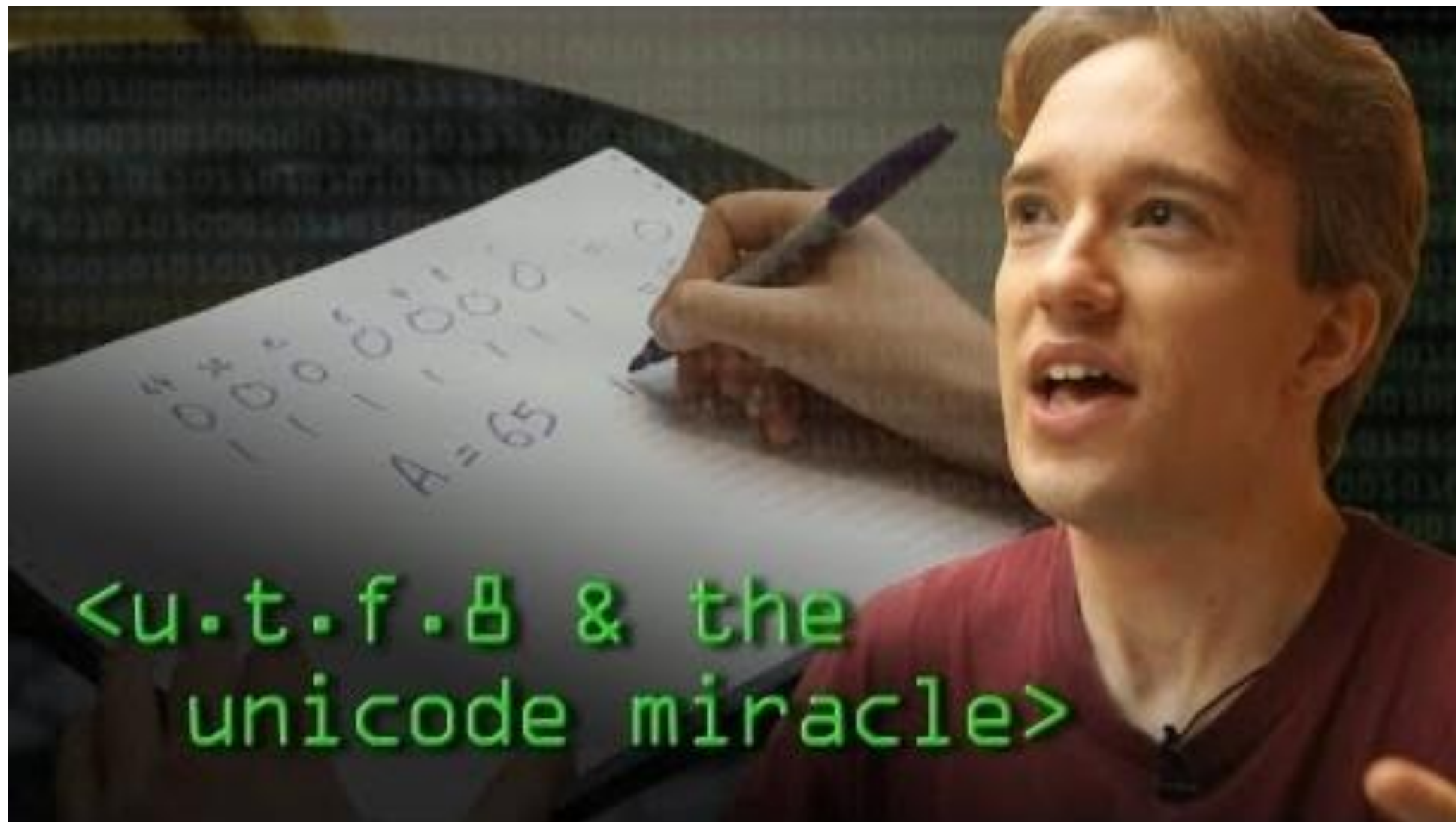
Por ejemplo:

**Texto:** ASCII, UTF-8

**Audio:** MP3, WAV, AAC, FLAC

**Video:** MP4, AVI, MOV

La clave es que cuando utilicemos un archivo, nuestro programa entienda y sepa cómo interpretar la secuencia de bytes contenidas dentro del archivo. **Es decir, que pueda traducirlo.**



[Link](#)

# Emoji... ¿cómo funcionan?

El **Unicode Consortium** (<https://unicode.org/>) es el organismo encargado de definir, mantener y crear los Emojis.

La lista completa de Emojis la podemos encontrar en Emojipedia  
<https://emojipedia.org/>

## Person Surfing

A person on a surfboard, riding on a wave in the ocean. Wears a wetsuit or board shorts in most versions.


*Person Surfing* was approved as part of [Unicode 6.0](#) in 2010 under the name “Surfer” and added to [Emoji 1.0](#) in 2015.

Copy and paste this emoji:







Copy

## Codepoints

 U+1F3C4

Después podemos encontrar su representación hexadecimal en la página:  
<https://www.utf8-chartable.de/unicode-utf8-table.pl>

U+1F3C2		f0 9f 8f 82	SNOWBOARDER
U+1F3C3		f0 9f 8f 83	RUNNER
U+1F3C4		f0 9f 8f 84	SURFER
U+1F3C5		f0 9f 8f 85	SPORTS MEDAL

f0 9f 8f 84



# Archivos de texto y archivos binarios

Veamos la comparación, este archivo de texto y el archivo binario contienen la misma información...

*A text file*

1	2	3	4	5		-	4	0	2	7		8		...
---	---	---	---	---	--	---	---	---	---	---	--	---	--	-----

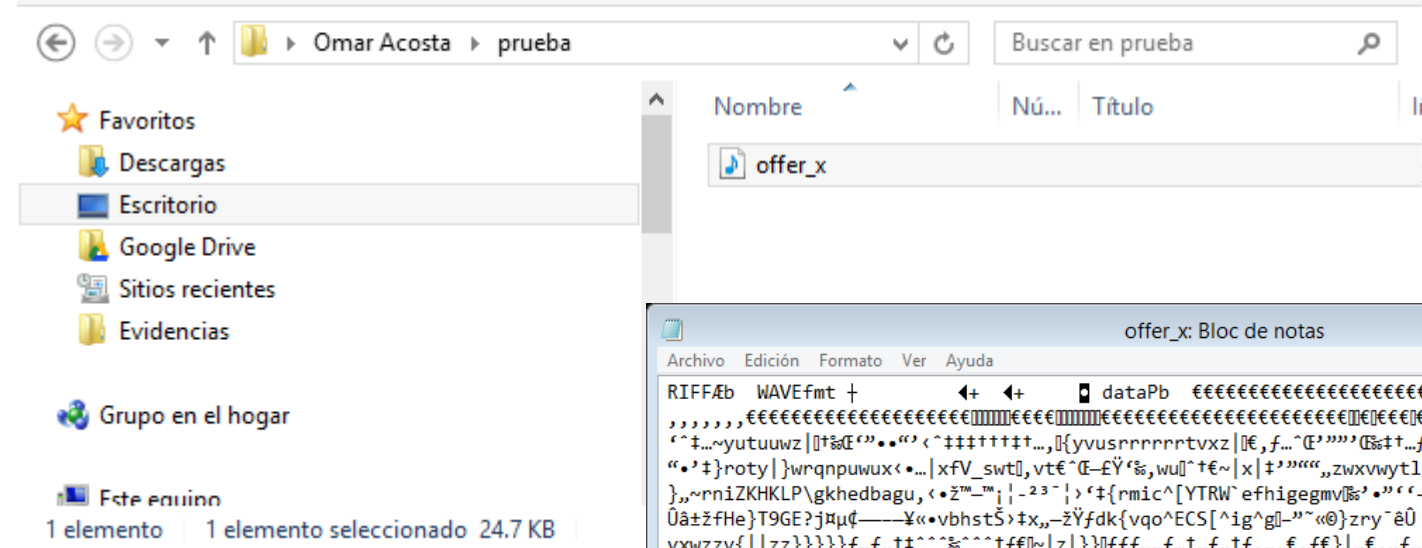
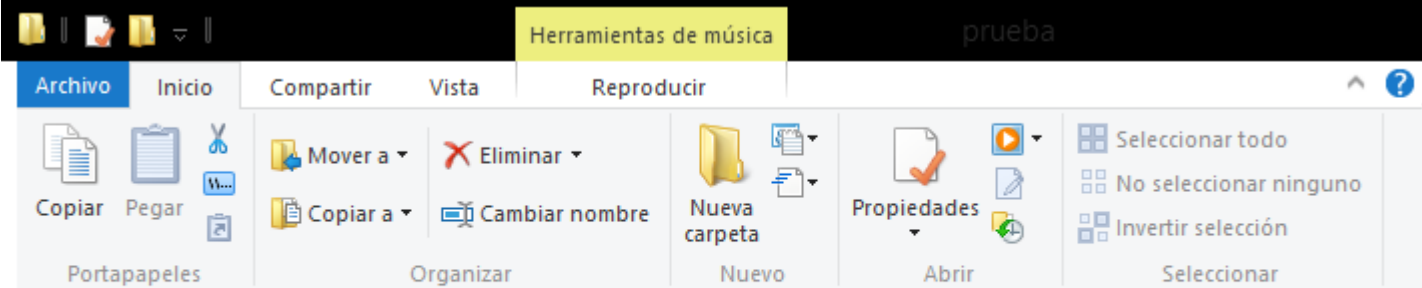
*A binary file*

12345	-4072	8	...
-------	-------	---	-----

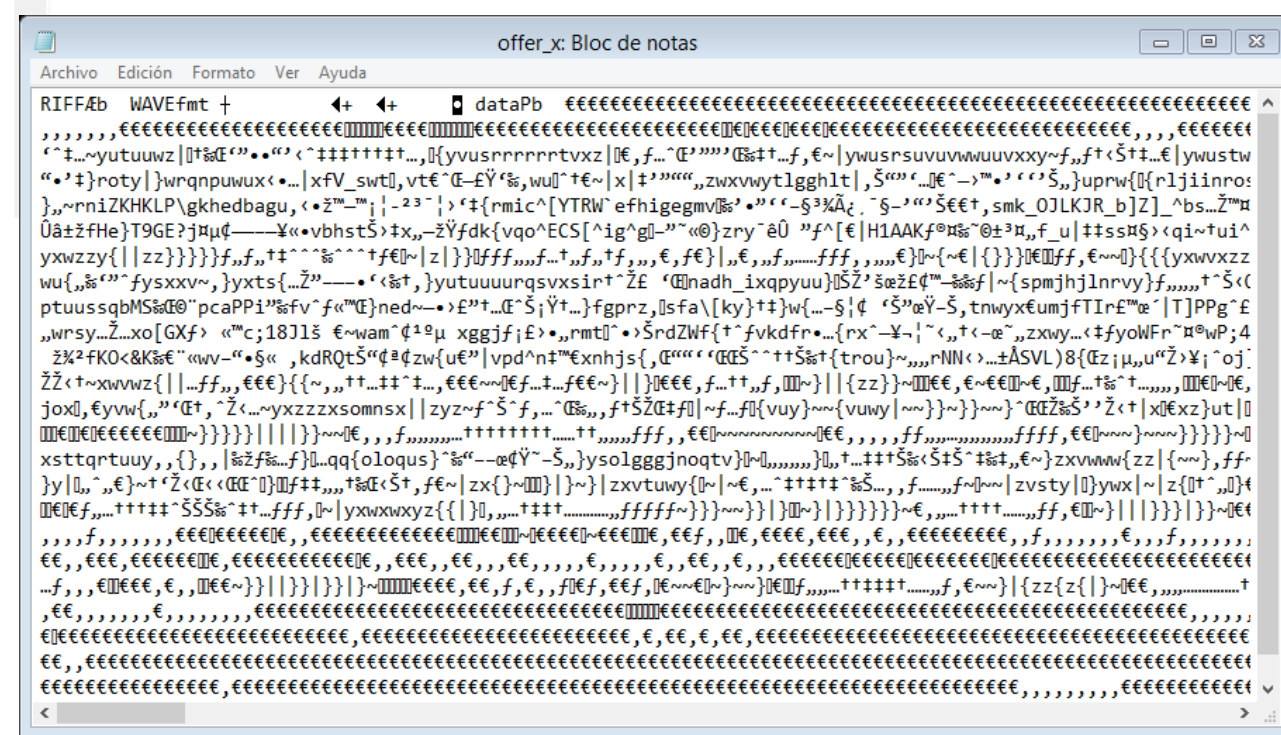
Cada casilla equivalen a 2 bytes

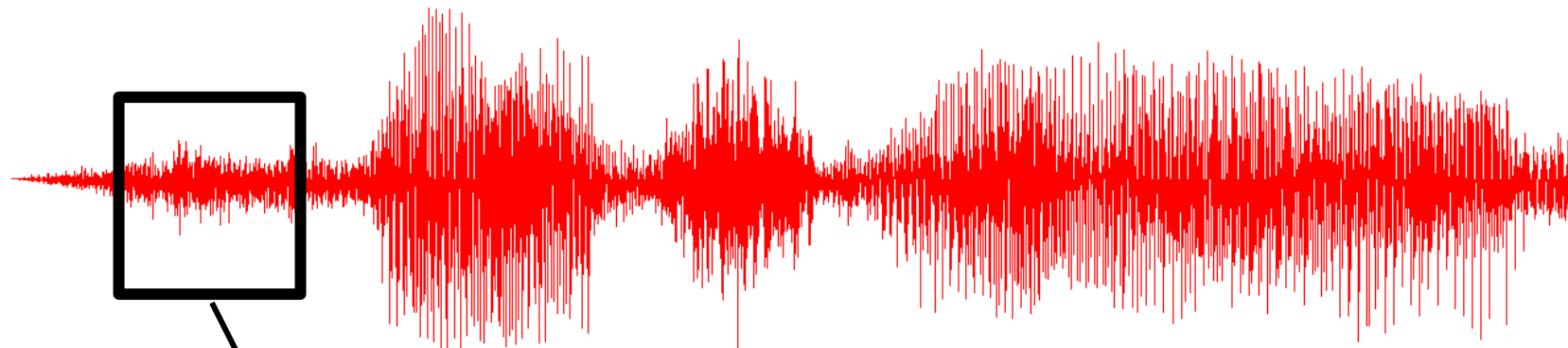
**NOTA:** El archivo binario es de 6 bytes.

El archivo de texto es de 28 bytes.

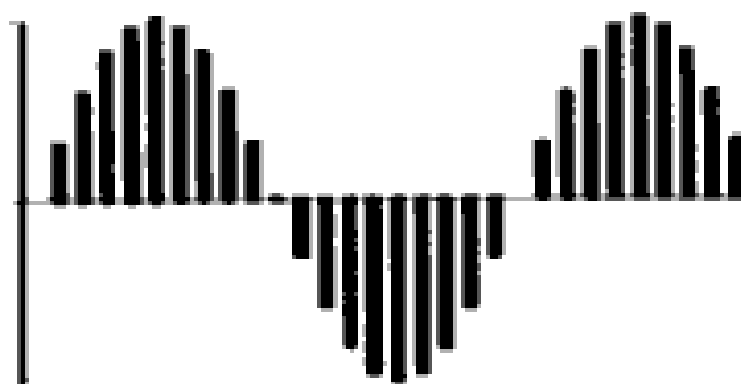


Archivo  
Binario  
(.wav)



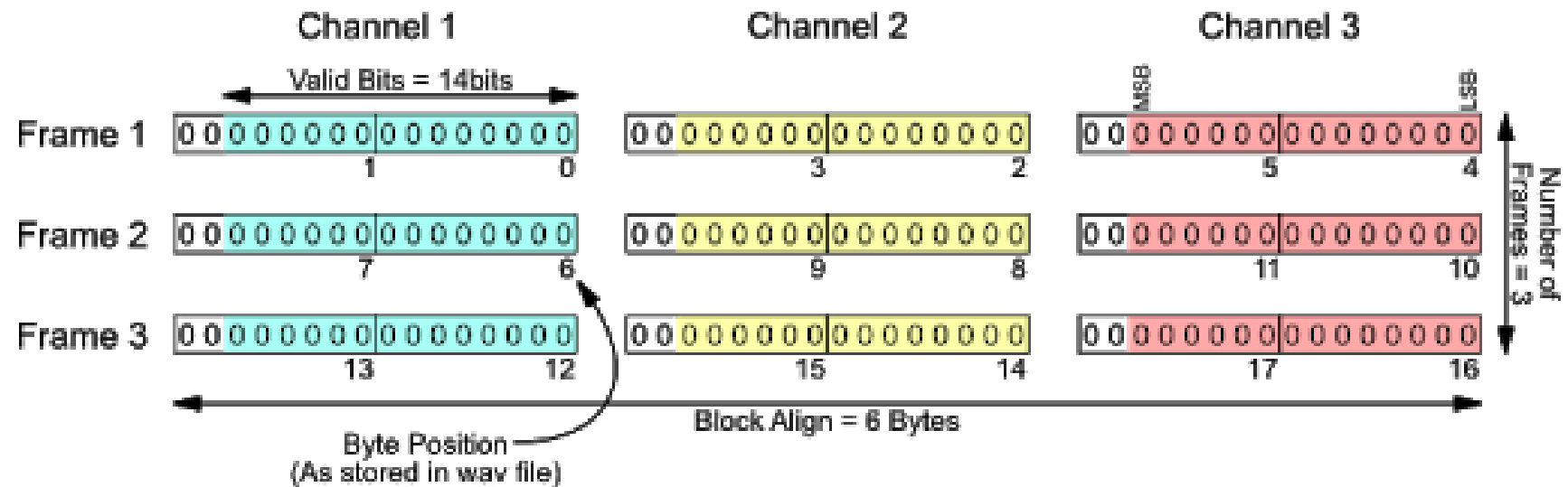


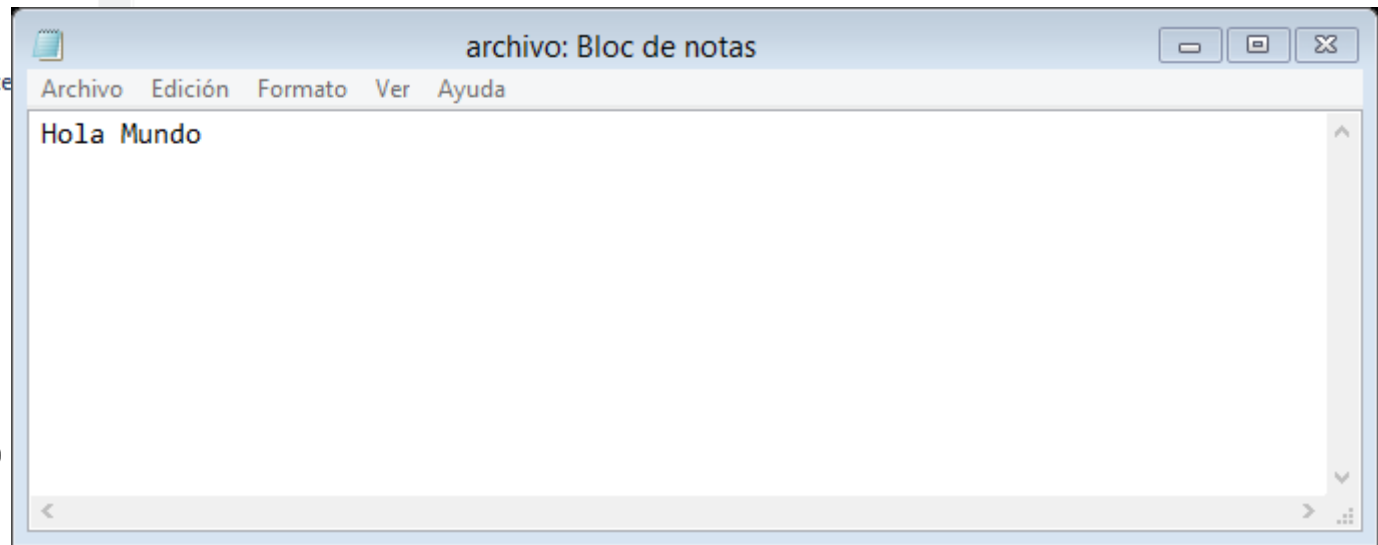
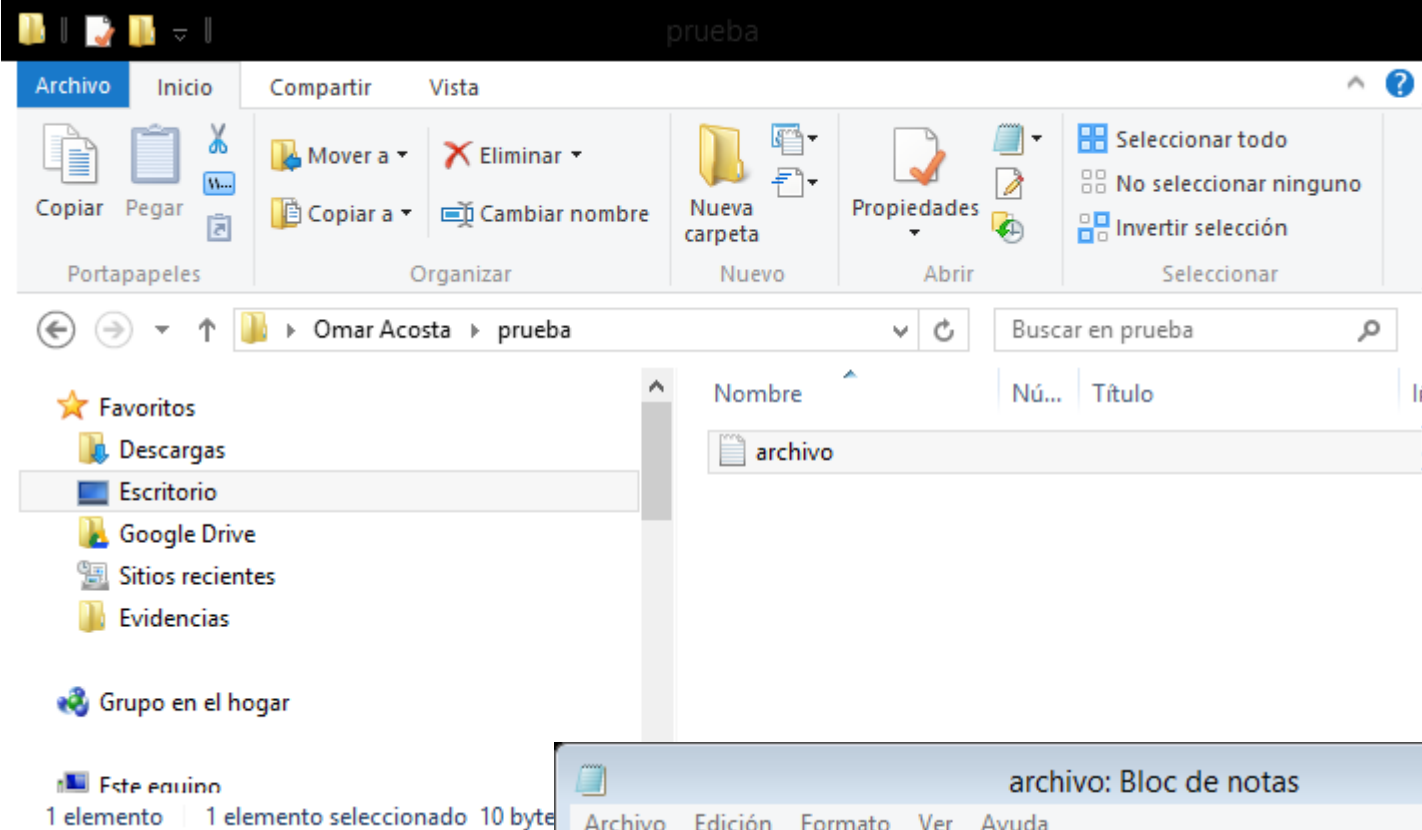
Señal muestreada



Max = 11111111111111

Min = 00000000000000





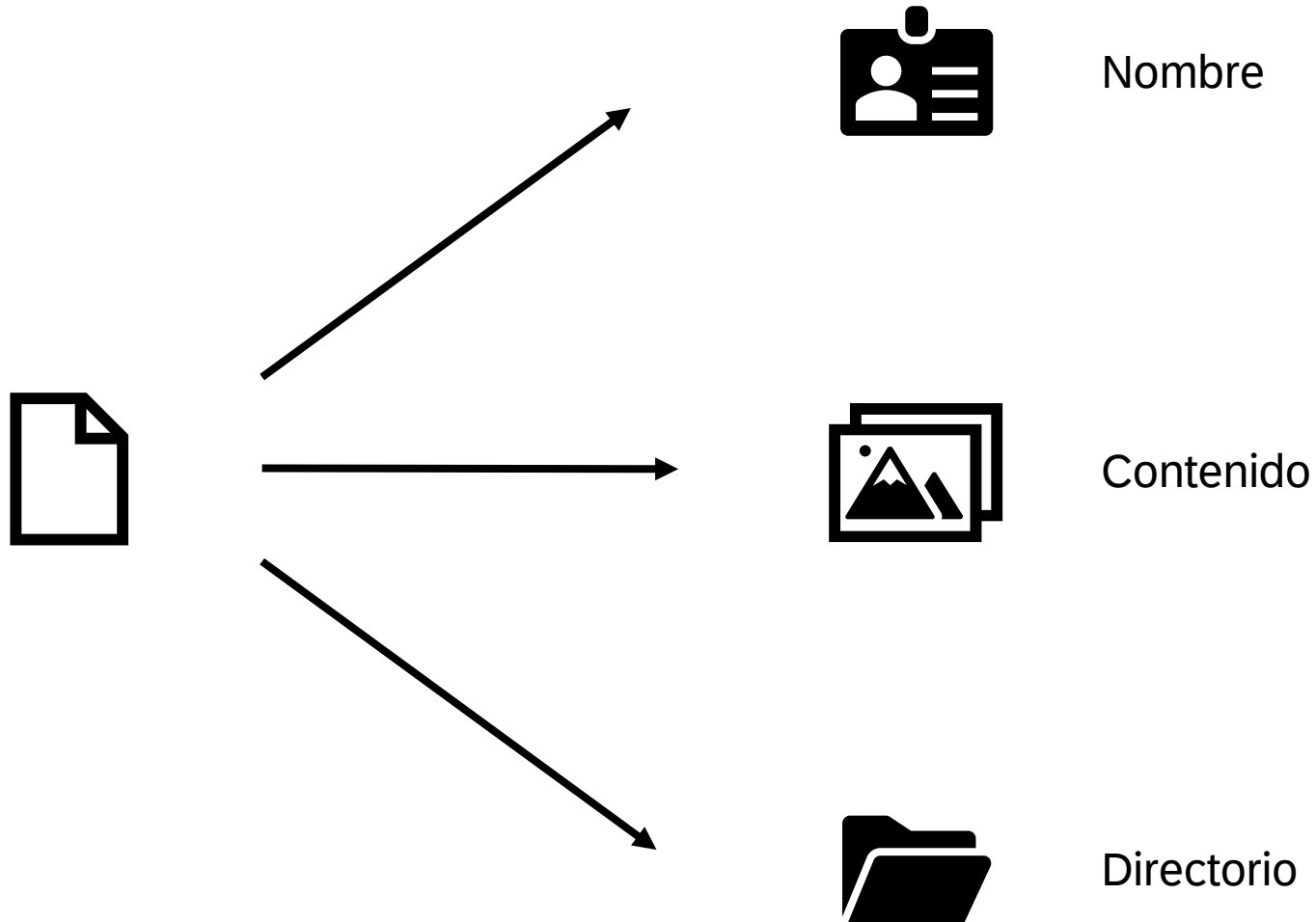
# Archivo De Texto (.txt)

# Archivos binarios vs texto

1. Los archivos binarios generalmente ocupan menos espacio.
2. Los archivos de texto son mas amigables para el uso humano.

**Crear un nuevo archivo**

Para crear un archivo, vamos a necesitar tres componentes:





# Crear un archivo de Texto

La clase **PrintWriter** define los métodos necesarios para crear y escribir a un archivo de texto.

- La clase se encuentra en el paquete **java.io**

Para poder abrir un archivo, debemos:

- Instanciar un objeto de la clase **PrintWriter**, usando el nombre del archivo como argumento.
- Rodear nuestro programa en bloques **try** y **catch**, para manejar las excepciones.

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class IO_Demo {

    public static void main(String[] args) {


        String filename = "out.txt";
        PrintWriter outputStream = null;

        try {
            outputStream = new PrintWriter(filename);
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        }

        System.out.println("Enter three lines of text:");
        Scanner keyboard = new Scanner(System.in);
        for (int i=0; i<3; i++) {
            String line = keyboard.nextLine();
            outputStream.println(i + ": " + line);
        }

        outputStream.close();
        keyboard.close();

        System.out.println("Finished");
    }
}
```



El archivo se generará en el  
folder del proyecto.

# Creating a Text File

Sample  
screen  
output

```
Enter three lines of text:  
A tall tree  
in a short forest is like  
a big fish in a small pond.  
Those lines were written to out.txt
```

## Resulting File

```
1 A tall tree  
2 in a short forest is like  
3 a big fish in a small pond.
```

*You can use a text editor  
to read this file.*

**!** SAVING...

Saving content. Please do not turn off your console.



# Tips al programar con archivos de texto

Cuando nuestros programas trabajan con archivos, es buena idea incluir informarle al usuario.

- Esto nos ayuda a evitar corrupción de información si interrumpimos el programa.

Recordemos la función del botón “Grabar”

- ¿En dónde has visto esta funcionalidad?
- Instalas actualizaciones de Windows
- Instalas aplicaciones en un celular
- Grabas el progreso en un videojuego
- etc!

**Modificar un archivo existente**

# Archivos de Texto

Hay ocasiones en donde requerimos agregar o modificar información en un archivo sin perder lo que ya teníamos!

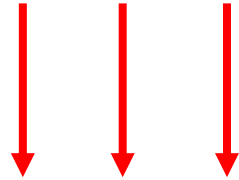
```
outputStream =  
    new PrintWriter(new FileOutputStream(fileName, true));
```

El método `println` agrega la información al final del archivo.

```
String fileName = "out.txt";
PrintWriter outputStream = null;

try {
    outputStream = new PrintWriter(filename);
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Crea un  
archivo nuevo



```
String fileName = "out.txt";
PrintWriter outputStream = null;

try {
    outputStream = new PrintWriter(new FileOutputStream(fileName, true));
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Modifica un archivo ya existente



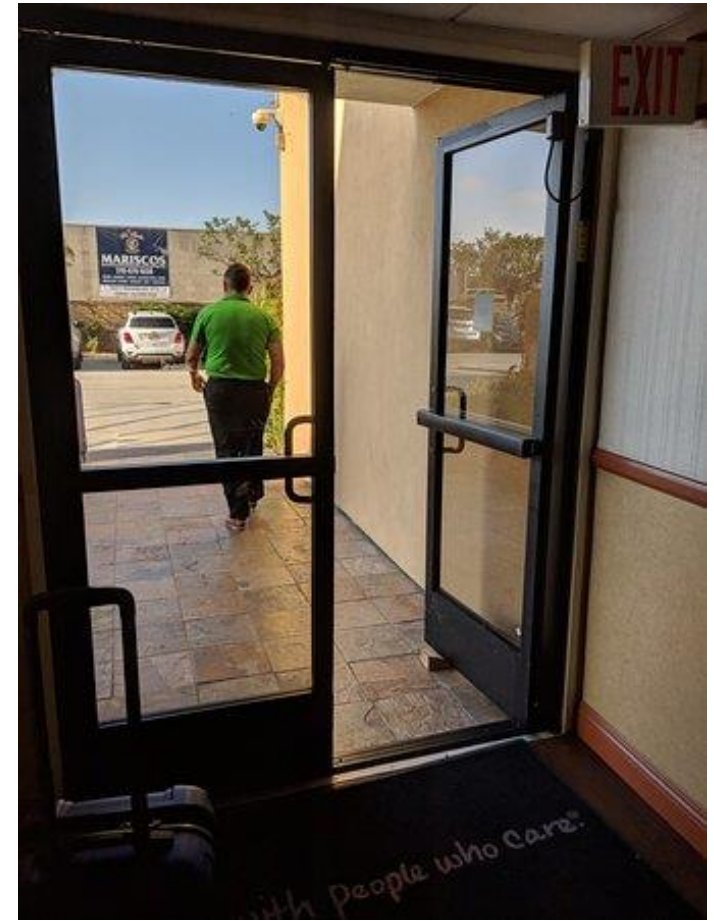
**Leer un archivo de texto**

# Leer de un archivo de Texto

Vamos a ver un ejemplo donde un archivo de texto se lee y se imprime en la pantalla.

OJO en los siguientes puntos:

1. Línea de código con la que se abre el archivo.
2. El objeto **Scanner**.
3. El método **hasNextLine()**
4. Al finalizar, el objeto debe cerrarse.



```
package file;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class IO_Demo {

    public static void main(String[] args) {
        String fileName = "out.txt";
        Scanner inputStream = null;
        try {
            inputStream = new Scanner(new File(fileName));
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
            System.exit(0);
        }

        while (inputStream.hasNextLine()) {
            String line = inputStream.nextLine();
            System.out.println(line);
        }
        inputStream.close();
    }
}
```

# Reading from a Text File

Sample  
screen  
output

The file out.txt  
contains the following lines:

```
1 A tall tree  
2 in a short forest is like  
3 a big fish in a small pond.
```

# Otros métodos

*Scanner\_Object\_Name*.hasNext()

Returns true if more input data is available to be read by the method `next`.

*Scanner\_Object\_Name*.hasNextDouble()

Returns true if more input data is available to be read by the method `nextDouble`.

*Scanner\_Object\_Name*.hasNextInt()

Returns true if more input data is available to be read by the method `nextInt`.

*Scanner\_Object\_Name*.hasNextLine()

Returns true if more input data is available to be read by the method `nextLine`.

# La Clase File

# La clase **File**

La clase permite una manera en la que podemos representar archivos de una manera general.

- El objeto **File** representa un archivo existente en algún dispositivo de almacenamiento de la computadora.

# La ruta (*path*) del archivo

Los archivos en los ejemplos asumen que el programa está en el mismo folder que el programa.

Podemos especificar la ubicación de un archivo con la **dirección absoluta**, o la **dirección relativa**.

- La dirección absoluta indica la ubicación del archivo considerando el disco, y los folders.  
Ejemplo: (C:\\User\\...\\out.txt)
- La dirección relativa asume que el archivo se encuentra en el folder del proyecto, y solo es necesario especificar su ubicación dentro del folder (“out.txt”).

Dependiendo del Sistema Operativo, la dirección del archivo puede variar

- Sistemas basados en Unix, ocupan sólo un dash “/” para indicar saltos entre folders.
- Sistemas basados en Windows, ocupan 2 dash-invertidos “\\”.



# Methods of the Class File

`public boolean canRead()`

Tests whether the program can read from the file.

`public boolean canWrite()`

Tests whether the program can write to the file.

`public boolean delete()`

Tries to delete the file. Returns true if it was able to delete the file.

`public boolean exists()`

Tests whether an existing file has the name used as an argument to the constructor when the File object was created.

`public String getName()`

Returns the name of the file. (Note that this name is not a path name, just a simple file name.)

`public String getPath()`

Returns the path name of the file.

`public long length()`

Returns the length of the file, in bytes.

# Caso de Estudio

## Archivo separado por commas

Los archivos separados por comas, o CSV, son archivos de texto que se usan para almacenar una lista de elementos. Ejemplo:

```
SKU,Quantity,Price,Description
4039,50,0.99,SODA
9100,5,9.50,T-SHIRT
1949,30,110.00,JAVA PROGRAMMING TEXTBOOK
5199,25,1.50,COOKIE
```

# CSV

Utilizamos la instrucción SPLIT para generar un arreglo en donde cada posición corresponde a un elemento de la lista.

```
String line = "4039,50,0.99,SODA"
String[] ary = line.split(",");
System.out.println(ary[0]);           // Outputs 4039
System.out.println(ary[1]);           // Outputs 50
System.out.println(ary[2]);           // Outputs 0.99
System.out.println(ary[3]);           // Outputs SODA
```

```
1 String fileName = "Transactions.txt";
2 try
3 {
4     Scanner inputStream = new Scanner(new File(fileName));
5     // Read the header line
6     String line = inputStream.nextLine();
7     // Total sales
8     double total = 0;
9     // Read the rest of the file line by line
10    while (inputStream.hasNextLine())
11    {
12        // Contains SKU,Quantity,Price,Description
13        line = inputStream.nextLine();
14        // Turn the string into an array of strings
15        String[] ary = line.split(",");
16        // Extract each item
17        String SKU = ary[0];
18        int quantity = Integer.parseInt(ary[1]);
19        double price = Double.parseDouble(ary[2]);
20        String description = ary[3];
21        // Output item
22        System.out.printf("Sold %d of %s (SKU: %s) at $%1.2f each.\n",
23            quantity, description, SKU, price);
24        // Compute total
25        total += quantity * price;
26    }
27    System.out.printf("Total sales: $%1.2f\n",total);
28    inputStream.close();
29 }
30 catch(FileNotFoundException e)
31 {
32     System.out.println("Cannot find file " + fileName);
33 }
34 catch(IOException e)
35 {
36     System.out.println("Problem with input from file " + fileName);
37 }
```

# Archivos Binarios

Java tiene un mecanismo llamado *serialización de Objetos (Object Serialization)* que permite que cualquier objeto sea representado como una secuencia de bytes.

Este proceso es independiente de la JVM, por lo que permite que un objeto sea serializado y deserializado en plataformas completamente diferentes.

# Archivos Binarios

Para esto, hacemos uso de las clases `ObjectInputStream` para serializar objetos, y `ObjectOutputStream` para deserializar objetos.

Veamos un ejemplo:

Tenemos una clase llamada **Estudiante**. Nótese que la clase está utilizando la interfaz “java.io.Serializable”. Esto está permitiendo a cualquier objeto de la clase **Estudiante** serializarse.

Ojo con la variable **transient int id**. La palabra reservada “transient” hace que el valor de la variable no pueda serializarse.

```
12 public class Estudiante implements java.io.Serializable {
13
14     private int edad;
15     private String nombre;
16     private transient int id;
17
18     public Estudiante(int edad, String nombre){
19         this.edad = edad;
20         this.nombre = nombre;
21     }
22
23     private void setEdad(int edad) {
24         this.edad = edad;
25     }
26
27     private void setNombre(String nombre){
28         this.nombre = nombre;
29     }
30 }
```

```
16 public class SerializeDemo {
17
18     public static void main(String[] args) {
19
20         //Declaramos el objeto Omar
21         Estudiante Omar = new Estudiante(26, "Omar Acosta");
22
23         try {
24             //generamos un nuevo archivo llamado employee.ser para grabar el
25             //objeto serializado
26             FileOutputStream fileOut = new FileOutputStream("employee.ser");
27             //generamos un objeto de la clase ObjectOutputStream
28             ObjectOutputStream out = new ObjectOutputStream(fileOut);
29             //serializamos el objeto Omar
30             out.writeObject(Omar);
31             out.close();
32             fileOut.close();
33             System.out.printf("Serialized data is saved in employee.ser");
34         }
35         catch (IOException i) {
36             i.printStackTrace();
37         }
38
39     }
40
41 }
```



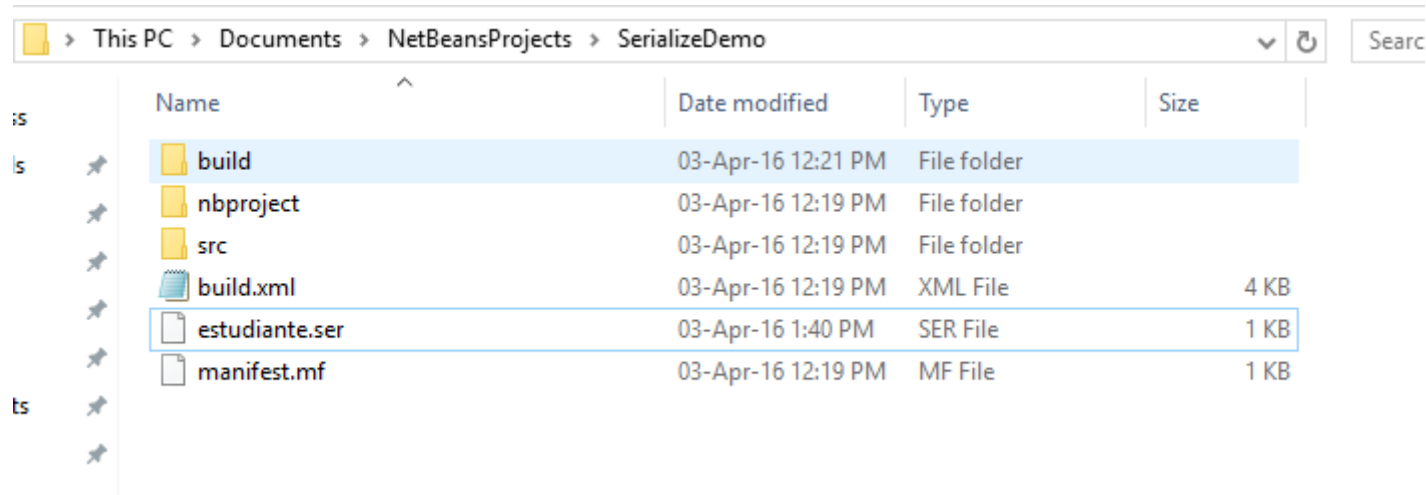
# Serialización de Clases

En el código anterior vemos un objeto de la clase **Estudiante** instanciarse. Después, se abre un buffer del archivo “**estudiante**”.a

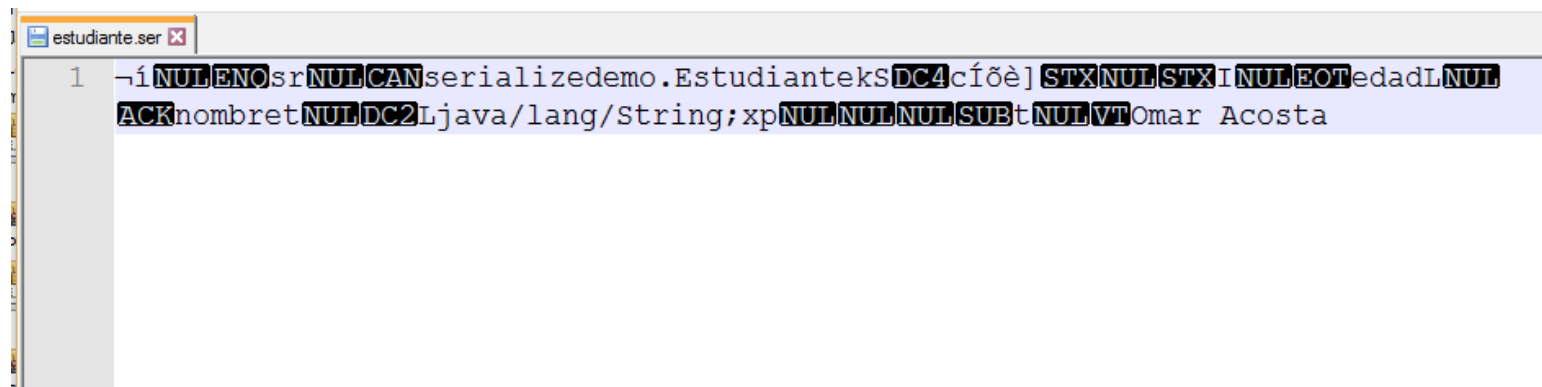
A través del método **writeObject** de la clase **ObjectOutputStream**, serializamos el objeto **Estudiante Omar**.

Esto genera un nuevo archivo en la ruta determinada

Vemos el archivo de texto en la ruta del proyecto. Si lo abrimos, podemos leer la información serializada.



	Name	Date modified	Type	Size
	build	03-Apr-16 12:21 PM	File folder	
	nbproject	03-Apr-16 12:19 PM	File folder	
	src	03-Apr-16 12:19 PM	File folder	
	build.xml	03-Apr-16 12:19 PM	XML File	4 KB
	estudiante.ser	03-Apr-16 1:40 PM	SER File	1 KB
	manifest.mf	03-Apr-16 12:19 PM	MF File	1 KB



```
1 -íNULENOsrNULCANserializedemo.EstudiantekSDC4cíõè]STXNULSTXI NULEOTedadLNUL  
ACKnombretNULDC2Ljava/lang/String;xpNULNULNULSUBtNULVT Omar Acosta
```

# Archivos Binarios

Supongamos que ahora queremos leer el archivo. Esto puede ser en una próxima ejecución del programa.

Para la lectura del objeto serializado utilizaremos la clase **ObjectInputStream**.

```
9 public class ReadSerializedData
10 {
11     public static void main(String [] args)
12     {
13         Estudiante e = null;
14         try
15         {
16             FileInputStream fileIn = new FileInputStream("estudiante.ser");
17             ObjectInputStream in = new ObjectInputStream(fileIn);
18             e = (Estudiante) in.readObject();
19             in.close();
20             fileIn.close();
21         } catch (IOException i)
22         {
23             i.printStackTrace();
24             return;
25         } catch (ClassNotFoundException c)
26         {
27             System.out.println("Estudiante class not found");
28             c.printStackTrace();
29             return;
30         }
31         System.out.println("Deserialized Estudiante...");
32         System.out.println("Name: " + e.getNombre());
33         System.out.println("Edad: " + e.getEdad());
34     }
35 }
```

# Archivos Binarios

Nótese como se busca el archivo “estudiante.ser” y comienza su lectura a través de la clase **ObjectInputStream**.

Ahora, se invoca el método **readObject()** que tiene un valor de retorno de tipo **Object**, y se realiza un casting a una variable de tipo “Estudiante”.

Nótese la firma del método **readObject()**. Recordemos que todas las clases heredan la clase base **Object**, por lo que a través del *dynamic binding* podemos hacer un casting a la clase **Estudiante**.

```
    */  
    public final Object readObject()  
        throws IOException, ClassNotFoundException  
    {
```

```
run:
```

```
Deserialized Estudiante...
```

```
Name: Omar Acosta
```

```
Edad: 26
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
|
```