

## Índice

- El tipo String
- Entrada/Salida en Java: Flujos
- Entrada/Salida Estándar
- Las clases StringTokenizer y StreamTokenizer
- Lectura/Escritura de Ficheros
  - La Clase File
  - Lectura/Escritura de Ficheros de Texto
  - Lectura/Escritura de Ficheros Binarios
  - Ficheros de Acceso Aleatorio
- Serialización
- Expresiones Regulares

## El tipo String

- Clase que admite operaciones para cadenas de caracteres
- Operaciones básicas del tipo
  - **Literales del tipo**: Se escriben entre `"` como por ejemplo  
`String cad = "Hola"`
  - **Concatenación**: Se utiliza el símbolo `+` por ejemplo  
`String s=s+"a"+"b"`
- Métodos de acceso
  - **Longitud de la cadena**: `int length()`
  - **Subcadena de una cadena**: `String substring(int inicio)` o `String substring(int inicio, int fin)`
  - **Valor de una posición**: `char charAt(int pos)`

## El tipo String

- Métodos de comparación y comprobación de cadenas:
  - Comparación de cadenas: `int compareTo(String otro)`
  - Igualdad de cadenas: `boolean equals(String otro)` si no se tienen en cuenta diferencia entre mayúsculas y minúsculas se utiliza `boolean equalsIgnoreCase(String otro)`
- Otras operaciones:
  - Conversión de cadenas: Se tienen varias posibles transformaciones:
    - `char[] toCharArray()`
    - `String toUpperCase()`
    - `String toLowerCase()`
  - Sustitución: `void replace(char c1, char c2)`
  - Localización de subcadenas:
    - `int lastIndexOf(String cadena)`
    - `int indexOf(String cadena)`

## Entrada/Salida en Java: Flujos

- La entrada/salida de datos en Java se realiza por medio de **flujos** (streams) que conectan el programa con la fuente o destino de los datos.
- Para gestionar la entrada/salida se necesitan distintas clases del paquete `java.io`. En particular son necesarios objetos que dependan de las clases:
  - `InputStream` y `OutputStream`, que manejan **bytes**
  - `Reader` y `Writer`, que manejan **caracteres** (Unicode)
- Para definir una comunicación con un dispositivo se debe empezar por determinar el **origen/destino** de los datos y luego se añaden **características** que permitan manejar los datos.

## Jerarquía InputStream/OutputStream

InputStream	OutputStream
<b>FileInputStream</b>	<b>FileOutputStream</b>
<b>PipedInputSream</b>	<b>PipedOutputStream</b>
<b>ByteArrayInputStream</b>	<b>ByteArrayOutputStream</b>
<b>StringBufferInputStream</b>	
SequenceInputStream	
FilterInputStream	FilterOutputStream
DataInputStream	DataOutputStream
BufferedInputStream	BufferedOutputStream
PushbackInputStream	PushbackOutputStream
LineNumberInputStream	PrintStream
ObjectInputStream	ObjectOutputStream

## Jerarquía Reader/Writer

Reader					
BufferedReader	<b>CharArrayReader</b>	InputStreamReader	FilterReader	<b>PipedReader</b>	<b>StringReader</b>
LineNumberReader		<b>FileReader</b>	PushbackReader		

Writer					
BufferedWriter	<b>CharArrayWriter</b>	OutputStreamWriter	FilterWriter	<b>PipedWriter</b>	<b>StringWriter</b>
		<b>FileWriter</b>			

## Entrada/Salida estándar

- Es necesario utilizar el paquete `java.lang` y dentro de él la clase `System`
- Tres objetos importantes de la clase `System`:
  - `System.in`: Pertenece a la clase `InputStream` dentro de esta clase el principal método es el de lectura `read()` (lectura byte a byte)
  - `System.out`: Pertenece a la clase `PrintStream` y permite la escritura de valores de cualquier tipo a partir de sus métodos `print` y `println`
  - `System.err`: Pertenece a la clase `PrintStream` y sirve para la escritura de mensajes de error

- El método `read` lanza una `IOException` que es necesario capturar.
- La lectura por teclado con el método `read` no es muy conveniente al realizarse una lectura byte a byte. Para ello se crea un objeto de la clase `BufferedReader` mediante la asignación:

```
BufferedReader teclado = new BufferedReader(  
                                new InputStreamReader(System.in));
```

El método más importante de esta clase es el que nos permite leer una línea entera: `String readLine()`



## Lectura/Escritura de Ficheros

- Se dispone de las clases `FileInputStream` y `FileOutputStream` para bytes; y de las clases `FileReader` y `FileWriter` para caracteres.
- Se puede construir un objeto de cualquiera de estas clases a partir de un `String` (nombre del archivo) o de un objeto de la clase `File`.
- Los constructores de los lectores pueden lanzar una `FileNotFoundException` si no encuentran el archivo.
- Los constructores de los escritores pueden lanzar una `IOException`, pero si el fichero no existe lo crean. Por defecto empiezan a sobrescribir un fichero que ya existe (booleano en el constructor).

## La Clase File

- Un objeto de esta clase puede representar un **fichero** o un **directorio**:
  - `File(String nombre)`
  - `File(String dir, String nombre)`
- Métodos para ficheros:
  - `boolean isFile()`: dice si el fichero existe
  - `long length()`: tamaño en bytes del fichero
  - `boolean canRead()`, `boolean canWrite()`: si se puede leer/escribir
  - `void delete()`: borra el fichero
  - `long lastModified()`: fecha de la última modificación
  - `void RenameTo(File f)`: cambio de nombre
- Métodos para directorios: `isDirectory()`, `mkdir()`, `delete()`, `String[] list()`.

## Lectura/Escritura de Ficheros de Texto

- Se realiza a través de las clases `FileReader` y `FileWriter`.
- Se pueden leer las líneas de un fichero utilizando un objeto `BufferedReader`:

```
BufferedReader br = new BufferedReader(  
    new FileReader("fichero.txt"));
```

- Se pueden escribir datos en un fichero de texto utilizando un objeto `PrintWriter` que se construye de la siguiente manera:

```
PrintWriter pr = new PrintWriter(  
    new BufferedWriter(  
        new FileWriter("fichero.txt")));
```

- La clase `PrintWriter` dispone de métodos `print` y `println` para los tipos primitivos del lenguaje

## La Clase StringTokenizer

- Es una clase del paquete `java.util` que permite **descomponer una cadena en tokens** (secuencias de caracteres delimitadas por separadores).
- Se construye a partir de una cadena indicando opcionalmente los separadores:

```
String delim = "\\t\\n.,!?;:'";
```

```
StringTokenizer troceador = new StringTokenizer(cadena,delim);
```

- Métodos para trocear:
  - `boolean hasMoreTokens()`: true si hay más elementos
  - `String nextToken()`: siguiente elemento. `NoSuchElementException` si no hay
  - `int countTokens()`: número de tokens en la cadena

## La Clase StreamTokenizer

- Es más potente: puede controlar el final de fichero, reconoce el final de línea y el tipo de palabras que lee.
- Atributos de la clase:
  - `int ttype` : contiene el tipo del token recién leído: un carácter convertido a entero o una de las constantes `TT_WORD`, `TT_NUMBER`, `TT_EOL`, `TT_EOF`
  - `String sval`: cadena de caracteres si se trata de una palabra (`null` inicialmente)
  - `double nval`: valor si se trata de un número (`0.0` inicialmente)
- El **constructor** recibe un objeto `Reader` como argumento:

```
StreamTokenizer troceador = new StreamTokenizer(  
    new BufferedReader(  
        new FileReader("fichero.txt")));
```

- Métodos de definición de parámetros:
  - `void ordinaryChar(int c)`: lo considera un token independiente
  - `void wordChars(int inf, int sup)`: el rango forman parte de palabras
  - `void whiteSpaceChars(int inf, int sup)`: espacios en blanco separadores
  - `void eolIsSignificant(boolean b)`: true si se trata el fin de línea como token
- Métodos de procesamiento de flujo: `int nextToken()`

## Lectura/Escritura de Ficheros Binarios

- Se realiza a través de las clases `DataInputStream` y `DataOutputStream`, que permiten leer y escribir **datos primitivos** directamente.
- Son clases diseñadas para **trabajar conjuntamente**: una lee lo que la otra escribe. Se utilizan para almacenar datos de manera independiente de la plataforma.

```
DataInputStream dr = new DataInputStream(  
    new BufferedInputStream  
    new FileInputStream("fichero.dat"));
```

- Soportan métodos de las interfaces `DataInput` y `DataOutput`:
  - Escritura de datos primitivos: `writeDouble`, `writeInt`, `writeByte`, `writeChar`, `writeBoolean`, ...
  - Lectura de datos primitivos: `readDouble`, `readInt`, `readByte`, `readChar`, `readBoolean`, ...

## Ficheros de Acceso Aleatorio

- Se utiliza la clase `RandomAccessFile` para localizar o escribir datos en cualquier parte de un fichero.
- Implementa las interfaces `DataInput` y `DataOutput`.
  - `RandomAccessFile(String nombre,String modo)`: el modo puede ser `r` (solo lectura) o `rw` (lectura y escritura).
  - `long getFilePointer()`: devuelve la posición actual del puntero.
  - `void seek(long pos)`: establece el puntero en la posición dada.
  - `long length()`: devuelve la longitud del fichero en bytes.



## Serialización

- Proceso por el que un objeto cualquiera se puede convertir en secuencia de bytes y posteriormente reconstruirlo a partir de ella.
- La clase **debe implementar la interfaz** `Serializable` (sin métodos). Casi todas las clases estándar de Java lo son.
- Para leer y escribir se usan:
  - La clase `ObjectInputStream` con el método `ObjectInputStream readObject()`
  - La clase `ObjectOutputStream` con el método `void writeObject(Object obj)`
- No se serializan:
  - atributos calificados con `transient`
  - variables y objetos `static` (programación específica)

## Control de la Serialización

- Se pueden redefinir los métodos de lectura/escritura de forma consistente.
- El comportamiento por defecto: `stream.defaultReadObject()`, `stream.defaultWriteObject()`.
- Los tipos primitivos se pueden leer/escribir con métodos idénticos a los de `DataInputStream` y `DataOutputStream`
- La interfaz `Externalizable` no tiene ningún comportamiento automático, se deja todo en manos del programador. Hay que definir:
  - `void writeExternal(ObjectOutput out)`
  - `void readExternal(ObjectInput in)`: debe ser capaz de recuperar lo que escribió la anterior.

## Expresiones Regulares

- Se emplean para **facilitar patrones de búsqueda** y se usan para localizar cadenas que cumplen un patrón determinado.
  - Caracteres: `c`, `\n`, `\.` (distinto de `.`)
  - Clases de caracteres: `[c1 c2 ...]`, `[^...]`, `[...&&...]`
  - Clases predefinidas: `\d`, `\D`, `\s`, `\w`, `\p{nombre}` (Lower, Upper, Digit).
  - Límites de los buscadores: `^`, `$`, `\b`
  - Cuatificadores: `X?`, `X*`, `X+`, `X{n}`
  - Sufijos de cuantificación: `?`, `+`
  - Operaciones de establecimiento: `XY`, `X|Y`
  - Agrupamiento: `(X)`, `\n` (empieza en `\1`)

- Uso de una expresión regular:

```
Pattern patron = Pattern.compile(cadenaPatron);  
Matcher m = patron.matcher(cadenaEntrada);  
if (m.matches()) ...
```

- Indicadores de compilación: CASE\_INSENSITIVE, ...

- El objeto Matcher permite ver los **límites de los grupos**:

- `int start(int posGrupo), int end(int posGrupo),`
- `String group(int posGrupo)`, la posición 0 es la cadena completa

- **Búsqueda de todas las coincidencias:**

```
while (m.find())  
{ int inicio = m.start();  
  int fin = m.end();  
  String encaje = cadenaEntrada.substring(inicio,fin);  
  ...}
```

■ Otros métodos útiles:

- `String replaceAll(String sustituta)` (de `Matcher`)

```
Pattern patron = Pattern.compile("[0-9]+");
```

```
Matcher m = patron.matcher(entrada);
```

```
String salida = m.replaceAll("#");
```

- `String[] split(CharSequence entrada)` (de `Pattern`)

```
Pattern patron = Pattern.compile("\\s*\\p{Punct}\\s*");
```

```
String[] tokens = patron.split(entrada);
```