ENTRADA SALIDA EN JAVA

1. ENTRADA Y SALIDA EN JAVA

Una de las operaciones más habituales que tiene que realizar un programa Java es intercambiar datos con el exterior. Para ello, el paquete **java.io** de Java SE incluye una serie de clases que permiten gestionar la entrada y salida de datos en un programa, independientemente de los dispositivos utilizados para el envío/recepción de los datos.

La figura 1 muestra cuáles son las principales clases de este paquete y la operación para la que son utilizadas

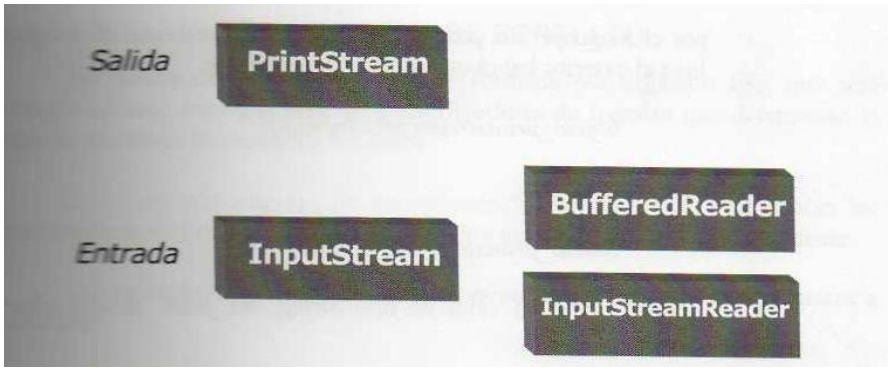


Fig. 1 - Principales clases para realizar la entrada y salida de datos

2. SALIDA DE DATOS

El envío de datos al exterior se gestiona a través de la clase **PrintStream,** utilizándose un objeto de la misma para acceder al dispositivo de salida. Posteriormente, con los métodos proporcionados por esta clase, podemos enviar la información al exterior.

El proceso de envío de datos a la salida debe realizarse siguiendo dos pasos.

1. **Obtención del objeto PrintStream.** Se debe crear un objeto PrintStream asociado al dispositivo de salida, la forma de hacerlo dependerá del dispositivo en cuestión. La clase **System** proporciona el atributo estático ***out*** que contiene una referencia al objeto PrintStream asociado a la salida estándar, representada por la consola.

2. **Envío de datos al stream.** La clase PrintStream dispone de los métodos ***print(String cadena)*** y ***println(String cadena)*** para enviar una cadena de caracteres al dispositivo de salida, diferenciándose uno de otro en que el segundo añade un salto de línea al final de la cadena. Esto explica que para enviar un mensaje a la consola se utilice la expresión:

```
System.out.println("texto de salida");
```

Ambos métodos se encuentran sobrecargados, es decir, existen varias versiones de los mismos para los distintos tipos soportados por el lenguaje.

por el lenguaje. En general, para enviar datos desde un programa Java al exterior habrá que utilizar la expresión:

objeto printstream.println(dato);
objeto printstream.print(dato)

siendo *"dato"* un valor de tipo *String, int, float, double, char o boolean.*

Salida con formato

LOS MÉTODOS FORMAT() Y PRINTF()

A partir de la versión Java 5, la clase *PrintStream* proporciona dos métodos de escritura que permiten aplicar un formato a la cadena de caracteres que se va a enviar a la salida. Se trata de los métodos ***printf()*** **y** ***format()****;* ambos realizan la misma función y tienen exactamente el mismo formato:

format (String formato, Object... datos)
printf (String formato, Object... datos)

El argumento *formato* consiste en una cadena de caracteres con las opciones de formato que van a ser aplicadas sobre los datos a imprimir.

Por otro lado, *datos* representa la información que va a ser enviada a la salida y sobre la que se va a aplicar el formato, siendo el número de estos datos variable. La sintaxis *Object ...datos* indica que se trata de un número variable de argumentos, en este caso puede tratarse de cualquier número de objetos Java.

A modo de ejemplo, dadas las siguientes instrucciones:

```
double cuad=Math.PI*Math.PI;
System.out.printf("El cuadrado de %l$.4f es %2$.2f",Math.PI,
cuad);
```

La salida producida por pantalla será:

El cuadrado de 3,1416 es 9,87

SINTAXIS DE LA CADENA DE FORMATO

La cadena de_formato puede estar formada por un texto fijo, que será mostrado tal cual, más una serie de especifícadores de formato que determinan la forma en que serán formateados los datos.
En el ejemplo anterior, las expresiones %l$.4f y %2$.2f representan los especificadotes de formato para los valores *Pi* y cuadrado de *Pi,* respectivamente.
Los especificadores de formato para números y cadenas deben ajustarse a la sintaxis:

%[posición_argumento$][indicador][mínimo][.num_decimales] conversión

El significado de cada uno de estos elementos es el siguiente:

- ***Posición_argumento***. Representa la posición del argumento sobre el que se va a aplicar el formato. El primer argumento ocupa la posición 1. Su uso es opcional.
- ***Indicador.*** Consiste en un conjunto de caracteres que determina el formato de

salida. Su uso es opcional. Entre los caracteres utilizados cabe destacar:

"-". El resultado aparecerá alineado a la izquierda.

'+'. El resultado incluirá siempre el signo (sólo para argumentos numéricos).

- ***Mínimo***. Representa el número mínimo de caracteres que serán presentados. Su uso también es opcional.
- ***Nmm_decimales***. Número de decimales que serán presentados, por lo que solamente es aplicable con datos de tipo float o double. Obsérvese que este valor debe venir precedido por un punto. Su uso es opcional.
- ***Conversión.*** Consiste en un carácter que indica cómo tiene que ser formateado el argumento. La tabla de la figura 2 contiene algunos de los caracteres de conversión más utilizados.

Carácter	***Función***
's', 'S'	Si el argumento es nuil se formateará como "nuil". En cualquier otro caso se obtendrá argumento.toString()
'c', 'C'	El resultado será un carácter unicode
'd'	El argumento se formateará como un entero en notación decimal
'x', 'X'	El argumento se formateará como un entero en notación hexadecimal
'e', 'E'	El argumento se formateará como un número decimal en notación científica
'f'	El argumento se formateará como un número decimal

Fig. 2 - Caracteres de formato de salida de datos

Para el formato de fechas, los especificadores de formato deben ajustarse a la sintaxis:

%[posicion_argumento$][indicador] [mínimo] conversión

El significado de los distintos elementos es el indicado anteriormente. En este caso, el elemento *conversión* está formado por una secuencia de dos caracteres. El primer carácter es 't' o 'T', siendo el segundo carácter el que indica cómo tiene que ser formateado el argumento.

Las tablas de las figuras 3 y 4 contienen la lista de los caracteres de conversión más utilizados para formato de horas y fechas, respectivamente.

Carácter	***Función***
'H'	Hora del día, formateada *como* un número de dos dígitos comprendido entre 00 y 23
'I'	Hora del día, formateada como un número de dos dígitos comprendido entre 01 y 12
'M'	Minutos de la hora actual, formateados como un número de dos dígitos comprendido entre 00 y 59
'S'	Segundos de la hora actual, formateados como un número de dos dígitos comprendido entre 00 y 59

Fig. 3 - Caracteres para formato de horas

Carácter	*Función*
'B'	Nombre completo del mes
'b'	Nombre abreviado del mes
'A'	Nombre completo del día de la semana
'a'	Nombre abreviado del día de la semana
'y'	Últimos dos dígitos del aflo
'e'	Día del mes formateado como un número comprendido entre 1 y 31

Fig. 4 - Caracteres para formato de fechas

Por ejemplo, dadas las siguientes instrucciones:

```
Calendar c = Calendar. getlnstance();

System.out.printf("%l$tH:%l$tM:%l$tS---%l$td de %l$tB"+
                                  "    de    %l$ty", c);
```

La salida producida por pantalla será la siguiente:

16:27:14---13 de noviembre de 05

3. Entrada de datos

La lectura de datos del exterior se gestiona a través de la clase **InputStream.** Un objeto InputStream está asociado a un dispositivo de entrada, caso de la entrada estándar (el teclado) podemos acceder al mismo a través del atributo estático *in* de la clase System.

Sin embargo, el método *read()* proporcionado por la clase *InputStream* para la lectura de los datos, no nos ofrece la misma potencia que *print* o *println* para la escritura. La llamada a *read()* devuelve el último carácter introducido a través de dispositivo, esto significa que para leer una cadena completa sería necesario hacerlo carácter a carácter, lo que haría bastante ineficiente el código.

Por ello, para realizar la lectura de cadenas de caracteres desde el exterior es preferible utilizar otra de las clases del paquete java.io: la clase **BufferedReader.**

La lectura de datos mediante **BufferedReader** requiere seguir los siguientes pasos en el programa:

1. **Crear objeto InputStreamReader.** Este objeto permite convertir los bytes recuperados del stream de entrada en caracteres. Para crear un objeto de esta clase, es necesario indicar el objeto InputStream de entrada, si la entrada es el teclado este objeto lo tenemos referenciado en el atributo estático *in* de la clase System:

```
InputStreamReader rd;
rd=new InputStreamReader(System.in);
```

2. **Crear objeto BufferedReader.** A partir del objeto anterior se puede construir un BufferedReader que permita realizar la lectura de cadenas:

```
BufferedReader bf;
```

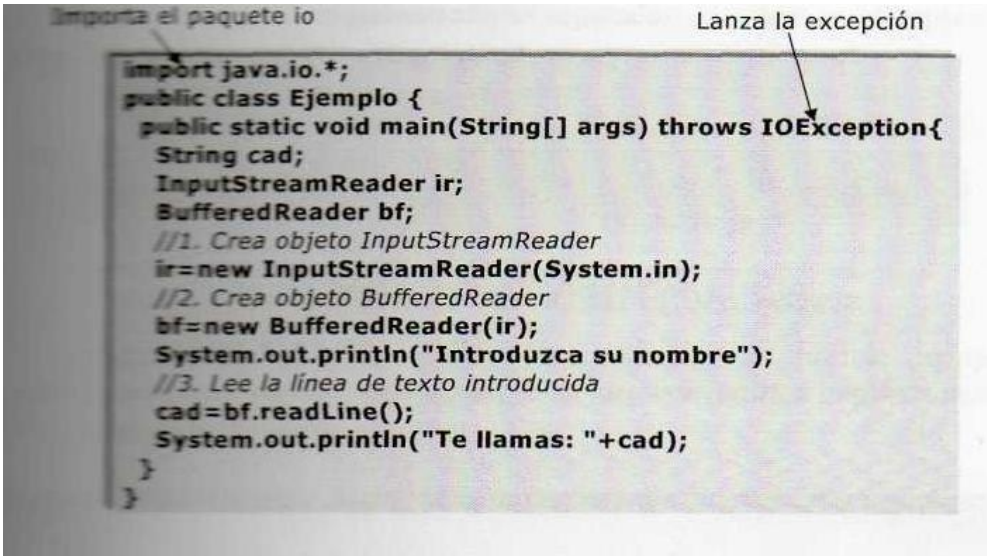
```
bf=new BufferedReader(rd);
```

3. **Invocar al método readLine().** El método *readLine()* de BufferedReader devuelve todos los caracteres introducidos hasta el salto de línea, si lo utilizamos para leer una cadena de caracteres desde el teclado devolverá los caracteres introducidos desde el principio de la línea hasta la pulsación de la tecla "enter":

```
String s=bf.readLine();
```

El ejemplo de la figura 5 se trata de un programa que solicita por teclado la introducción de una cadena de texto para, posteriormente, mostrarla por consola.

Hay que mencionar un punto importante a tener en cuenta cuando se utilizan ciertos métodos de determinadas clases, se trata del hecho de que al invocar a estos métodos el programa puede lanzar una **excepción.** Aunque más adelante se tratará en profundidad el tema de las excepciones, **cuando la llamada a un método de un objeto puede lanzar una excepción el programa que utiliza ese método está obligado a capturarla o a relanzarla.**



Éste es el caso del método *readLine()* de BufferedReader, cuya llamada; puede lanzar la excepción **lOException.** En este ejemplo, se ha optado por relanzar la excepción, incluyendo la expresión ***throws lOException*** en la cabecera del método *main()*.

Cuando se utiliza *readLine()* para leer datos numéricos, hay que tener en cuenta que el método devuelve los caracteres introducidos como tipo String, por lo que deberemos recurrir a los métodos de las clases de envoltorio (se comentan más adelante, **los** métodos estáticos *parseXxx(String) se utilizan* para convertir el dato a número y poder operar con él.

El siguiente código corresponde a un programa que realiza el cálculo del factorial de un número, incluyendo la lectura de dicho número por teclado a través del método *readLine():*

```
import java.io.*;
public class Factorial {
     public static void main (String[] args)
                    throws IOException{
          string cad;
          long result=1;
          BufferedReader bf=new BufferedReader(
                    new InputStreamReader(System.in)};
          System.out.println("Introduzca el número");
          cad=bf.readline(); //Obtiene el número en
```

```
                        //formato cadena
        long num=Long.parseLong(cad); //Convierte la cadena
                                      //a número
        for(int i=1; i<=num; i++){
           result*=i;
        }
        System.out.println ( "El factorial de "+num+
                                " es "+result } ;
        }
}
```

4. CLASES DE ENVOLTORIO

CLASES DE ENVOLTORIO

Para cada uno de los tipos de datos básicos, Java proporciona una clase que lo representa. A estas clases se las conoce como **clases de envoltorio,** y sirven para dos propósitos principales:

- **Encapsular un dato básico en un objeto**, es decir, proporcionar un mecanismo para "envolver" valores primitivos en un Objeto para que los primitivos puedan ser incluidos en actividades reservadas para los objetos, como ser añadido a un vector o devuelto desde un método.

- Proporcionar un conjunto de funciones útiles para los primitivos. La mayoría de estas funciones están relacionadas con varias conversiones: convirtiendo primitivos a String y viceversa y convirtiendo primitivos y objetos String en diferentes bases, tales como, binario, octal y hexadecimal.

Hay una clase de envoltorio para cada primitivo en Java. Por ejemplo, la clase de envoltura para int es Integer, la clase Float es de float y así todas. Recuerda que los nombres de los datos primitivos es simplemente el nombre en minuscula, a excepción de int que es Integer y char que es Character. La siguiente tabla nos muestra la lista de clases de envoltura en la API de Java.

Primitivo	Clase de envoltura	Argumentos del constructor
boolean	Boolean	boolean o String
byte	Byte	byte o String
char	Character	char
double	Double	double o String
float	Float	float, double o String
int	Integer	int o String
long	Long	long o String
short	Short	short o String

Encapsulamiento de un tipo básico. Construcción de los envoltorios.

Todas las clases envoltorio excepto Character provee dos constructores: uno que toma un dato primitivo y otro que toma una representacion String del tipo que esta siendo construido. Por ejemplo:

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

o

```
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");
```

La clase Carácter suministra sólo un constructor que toma un char como argumento, ejemplo:

```
Character c1 = new Character('c');
```

Los constructores para los envoltorios booleanos toman también un valor booleano trae o false, una cadena sensible a mayúsculas con el valor trae o false. Hasta Java 5, un objeto boolenao no podía ser usado como una expresión en un test booleano, por ejemplo:

```
Boolean b = new Boolean("false");
if (b) // no compila en Java 1.4 o anterior
```

En Java 5, un objeto booleano puede ser usado en el test, porque el compilador automaticamente desempacará (unbox) el Boolean a booleano.

El método valueOf()

El método estatico **valueOf()** es suministrado en la mayoría de las clases de envoltorio para darte la capacidad de crear objetos envoltorio. También toma una cadena como representación del tipo de Java como primero argumento. Este método toma un argumento adicional, int radix, que indica la base (por ejemplo, binario, octal, o hexadecimal) del primer argumento suministrado, por ejemplo:

```
Integer i2 = Integer.valueOf("101011", 2);   // convierte 101011
                                             // a 43 y
                                             // asigna el valor
                                             // 43 al objeto
                                             // Integer i2
```

o

```
Float f2 = Float.valueOf("3.14f");           // asigna 3.14 al objeto
                                             // Float f2
```

Usando las herramientas de conversión para envoltorios

Como dijimos antes, la segunda gran función de un envoltorio es la conversión. Los siguientes métodos son los más usados.

xxxValue()

Los métodos **xxxValue().** Se utilizan cuando se necesita convertir el valor de un envoltorio numérico a primitivo. Todos los métodos de esta familia no tienen argumentos. Hay 36 métodos. Cada una de las seis clases envoltorio, tiene seis métodos, así que cualquier número envoltorio puede ser convertido a cualquier tipo primitivo, por ejemplo:

```
Integer i2 = new Integer(42);        // Crea un nuevo objeto envoltorio
byte b = i2.byteValue();             // convierte el valor de i2 a byte

short s = i2.shortValue();           // otro método de xxxValue
                                     //para números enteros
double d = i2.doubleValue();         // otro más
```

o

```
Float f2 = new Float(3.14f);         // crea un nuevo objeto envoltorio
short s = f2.shortValue();           // convierte el valor de f2's a short
System.out.println(s);       // el resultado es 3 (truncado, no
redondeado)
```

parseXxx() y valueOf()

Los seis métodos **parseXxx()** (uno por cada tipo de envoltorio) son muy parecidos a los de **valueOf().** Los dos toman una cadena como argumenta arrojando NumberFormatException (comunmente denominado NFE) si el argumento cadena no esta apropiadamente formateado, y puede convertir objetos de cadena a diferentes bases (radix), cuando el dato primitivo es cualquiera de los cuatro tipos de números enteros. La diferencia entre ambos métodos es:

- parseXxx() devuelve el primitivo.
- valueOf() devuelve el recién creado objeto envoltorio del tipo invocado en el método.

Aquí se muestran algunos de los ejemplos de estos métodos en acción:

```
double d4 = Double.parseDouble("3.14");      // convierte un String
                                             // a un primitivo
System.out.println("d4 = " + d4);    // el resultado es d4 = 3.14

Double d5 = Double.valueOf("3.14");          // crea un objeto
                                             //Double
System.out.println(d5 instanceof Double); // el resultado es "true"
```

Los siguientes ejemplos de envoltura usan el parámetro radix (en esta caso binario):

```
long L2 = Long.parseLong("101010", 2);       // binary String a

System.out.println("L2 = " + L2);            // resultado es: L2 = 42

Long L3 = Long.valueOf("101010", 2);         // binary String a
                                             // Long objecto
System.out.println("L3 value = " + L3);      // resultado es:
                                             // L3 value = 42
```

toString

La clase alfa, o sea, la primera clase de Java, Object, tiene un metodo toString(). Asi que todas las clases de Java tienen también este método. La idea de toString() es permitirte conseguir una representación mas significativa del objeto. Por ejemplo, si tienes una coleccion de varios tipos de objetos, puedes hacer un bucle a través de la colección e imprimir algunas de las representaciones más significativas de cada objeto usando toString(), que está presente en todas las clases. Hablaremos mas de toString() en las colecciones en el capitulo correspondiente, ahora vamos a centrarnos en como funciona toString() en las clases envoltorio, que como ya sabemos, están marcadas como clases finales (ya que son inmutables). **Todas las clases envoltorio devuelven una cadena con el valor del objeto primitivo del objeto,** por ejemplo:

```
Double d = new Double("3.14");
System.out.println("d = "+ d.toString() ); // result is d = 3.14
```

Todas las clases envoltorio suministran un método sobrecargado, que toma un número primitivo de un tipo apropiado (Double.toString() toma un double, Long.toString() toma un long, etc...) y naturalmente devuelve una cadena.

```
String d = Double.toString(3.14); // d = "3.14"
```

Para terminar, los enteros y los long suminstran un tercer método, es estático. Su primer argumento es el dato primitivo, y el segundo argumento es el radix. El radix le dice al método como coger el primer argumento, lo que significa que si el radix es 10 (base 10, por defecto), lo convertirá a la base sumnistrada, y devolverá una cadena, por ejemplo:

```
String s = "hex = "+ Long.toString(254,16); // s = "hex = fe"
```

toXxxString() (Binario, Hexadecimal, Octal)

Las clases envoltorio Long e Integer te permiten convertir números en base 10 a otras bases. Estas conversiones toman un int o un long y devuelven una cadena que representa la conversión del número. Por ejemplo:

```
String s3 = Integer.toHexString(254);     // convert 254 to hex
System.out.println ("254 is " + s3);      // result: "254 is fe"
String s4 = Long.toOctalString(254);      // convert 254 to octal
System.out.print("254(oct) ="+ s4); // result: "254(oct) =376"
```

Estudiar la siguiente tabla es la mejor manera de diferenciar entre **xxxValue(), parseXxx() y valueOf()**..

Método	Boolean	Byte	Character	Double	Float	Integer	Long	Short
byte Value		x		x	x	x	x	x
double Value		x		x	x	x	x	x
float Value		x		x	x	x	x	x
int Value		x		x	x	x	x	x
long Value		x		x	x	x	x	x
short Value		x		x	x	x	x	x
parseXxx s,n		x		x	x	x	x	x
parse Xxx s,n (con radix)		x				x	x	x
valueOf s,n	x	x		x	x	x	x	x
valueOf s,n (con radix)		x				x	x	x
toString s	x	x	x	x	x	x	x	x
toString s (primitive)	x	x	x	x	x	x	x	x
toString s (primitive, radix)						x	x	

En esencia, los metodos para la conversóon de envoltorios son:

Primitive xxxValue(), convierte un envoltorio a primitivo.

primitive parseXxx(String), convierte una cadena a un primitivo.

Envoltorio valueOf(String), convierte cadena a envoltorio.

La s, significa static, y la n, NFE.

Autoboxing

Con la introducción de autoboxing (y unboxing) en Java 5 muchas de las operaciones de envoltura que realizaban los programadores manualmente ahora se hacen de manera automática**.**

El *autoboxing* representa otra de las nuevas características del lenguaje incluidas a partir de la versión Java 5 siendo, probablemente, una de las más prácticas.

Consiste en la encapsulación automática de un dato básico en un objeto de envoltorio, mediante la utilización del operador de asignación.

Por ejemplo, según se ha explicado anteriormente, para encapsular un dato entero de tipo *int* en un objeto *Integer,* deberíamos proceder del siguiente modo:

```
int p = 5;
Integer n = new Integer (p);
```

Utilizando el *autoboxing* la anterior operación puede realizarse de la siguiente forma:

```
int p = 5; Integer n = p;
```

Es decir, la creación del objeto de envoltorio se produce implícitamente al asignar el dato a la variable objeto.

De la misma forma, para obtener el dato básico a partir del objeto de envoltorio no será necesario recurrir al método *xxxValue(),* esto se realizará implícitamente al utilizar la variable objeto en una expresión. Por ejemplo, para recuperar el dato encapsulado en el objeto *n* utilizaríamos:

```
int a = n;
```

A la operación anterior se le conoce como *autounboxing* y, al igual que el *autoboxing,* solamente pueden ser utilizadas a partir de la versión Java 5.

El *autoboxing/autounboxing* permite al programador despreocuparse de tener que realizar de forma manual el encapsulado de un tipo básico y su posterior recuperación, reduciendo el número de errores por esta causa.

El siguiente método representa un ejemplo más de la utilización del *autoboxing/autounboxing,* en él se hace uso de esta técnica para mostrar por pantalla los números recibidos en un array de objetos Integer:

```
void muestraNumeros{Integer [] nums){ int suma=0;

for(Integer n: nums)

      {
            suma+=n; //autounboxing
            System.out.println{"El número vale "+n);
      1
      System.out.println{"La suma total es "+suma);
```