1. Programación de la consola

Una aplicación de consola es un programa informático diseñado para ser utilizado a través de una interfaz en el modo de solo texto. Algunos ejemplos de ello son la interfaz de línea de comandos de algunos sistemas operativos (Unix, GNU/Linux, etcétera), la consola Win32 en Microsoft Windows y la terminal en MacOS de Apple.

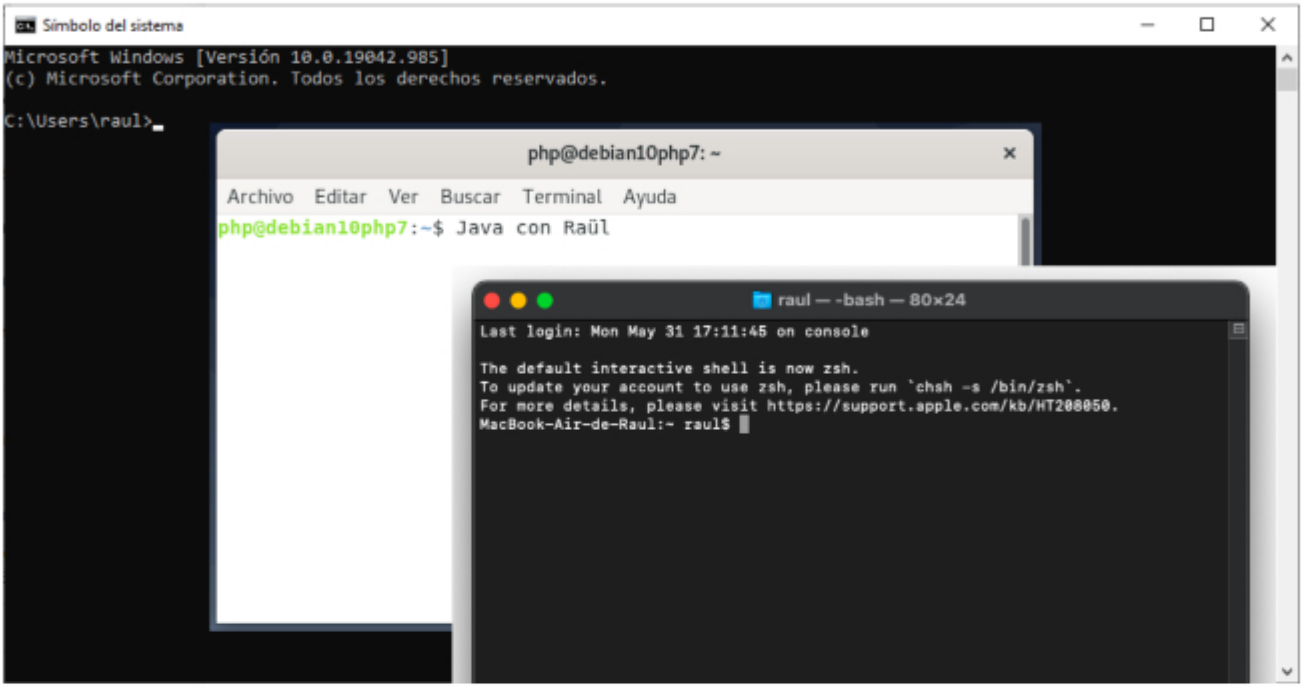

Figura 3.1. Ejemplos de consolas de terminales en diferentes sistemas operativos

Un usuario generalmente interactúa con una aplicación de consola usando solo un teclado y una pantalla, a diferencia de las aplicaciones de IGU, que normalmente requieren el uso de un ratón (en inglés, *mouse*) o de algún otro dispositivo señalador (TouchPad o similar).

1.1. Aplicaciones de consola

Muchas aplicaciones de consola, como las terminales o intérpretes de línea de comandos, son herramientas del sistema, pero también existen numerosos programas de interfaz de usuario basada en texto. Estas aplicaciones pueden ser actualizaciones de aplicaciones en modo texto que aparecieron cuando solo había sistemas operativos en modo texto, como el DOS o Unix, entre otros. También puede tratarse de aplicaciones nuevas que se considera que no requieren de entorno gráfico, y que solo en modo texto solucionan el problema que la empresa desee resolver.

A medida que la velocidad y la facilidad de uso de las aplicaciones de IGU han mejorado con el tiempo, el uso de las aplicaciones de consola ha disminuido en gran medida, pero no ha desaparecido. Algunos usuarios simplemente prefieren las aplicaciones basadas en consola, mientras que algunas organizaciones aún dependen de las aplicaciones de consola existentes para manejar las tareas clave del procesamiento de datos.

La capacidad de crear aplicaciones de consola se mantiene como una característica de los entornos de programación modernos, porque simplifica enormemente el proceso de aprendizaje de un nuevo lenguaje de programación al eliminar la complejidad de una IGU.

1.2. Concepto de flujo

En Java, dentro del paquete java.io se define la abstracción de flujo de datos (del inglés, *stream*) para tratar la comunicación de información entre el programa y el exterior. De este modo, entre una fuente (por ejemplo, un teclado) y un destino (por ejemplo, la pantalla) fluye una secuencia de datos.

IMPORTANTE

En Java, el concepto de **paquete** sirve para agrupar clases (en una misma carpeta) dentro de un contenedor que permite, a su vez, agrupar las distintas partes de un mismo programa con ciertas funcionalidades y elementos comunes. En nuestro caso, el paquete java.io es una biblioteca del sistema que nos evita asumir la complejidad de utilizar el teclado y la pantalla del sistema, pues nos permite usar directamente las clases que nos proporciona.

Los flujos de datos actúan a modo de interfaz con el dispositivo o la clase asociada. Esto permite:

- Realizar operaciones independientes del tipo de datos y/o del dispositivo.
- Gozar de mayor flexibilidad (por ejemplo, redirección, combinación, etcétera).
- Contar con más diversidad de dispositivos (fichero, pantalla, teclado, red…).
- Tener mayor variedad en las formas de comunicación.

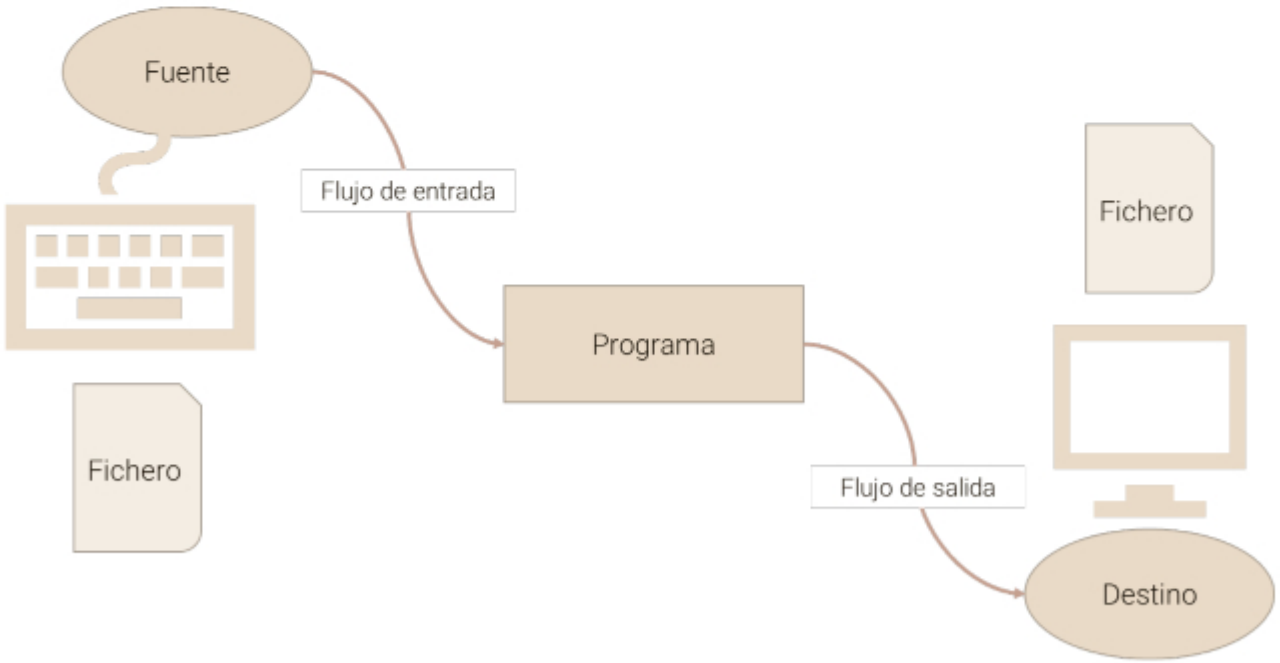

Figura 3.2. Posibles flujos tanto de entrada como de salida en un programa de Java

A. Tipos de flujos

Empezamos viendo las clases que tenemos dentro del paquete java.io. Para flujos de bytes, disponemos de las clases InputStream y OutputStream. Y para los flujos de caracteres disponemos de las clases Reader y Writer. Así pues, podemos pasar, en cualquier momento, de un flujo de bytes a uno de caracteres con las clases puente de InputStreamReader y OutputStreamWriter.

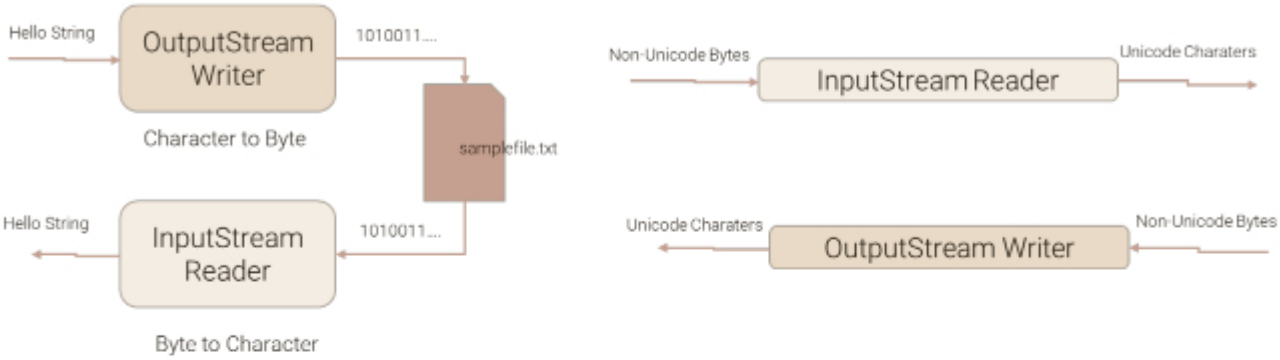

Figura 3.3. Las clases puente para convertir flujos de bytes a caracteres UNICODE o viceversa

B. Flujos predefinidos

Dentro del paquete java.lang disponemos de:

		Métodos
System.in	Es una instancia de la clase InputStream, que es el flujo de bytes de entrada del sistema estándar.	• read() permite leer un byte de la entrada como entero. • skip(n) ignora n bytes de la entrada. • available() número de bytes disponibles para leer en la entrada.
System.out	Es una instancia de la clase PrintStream: flujo de bytes de salida del sistema estándar.	• print() permite escribir un byte por la salida estándar, sin saltar de línea. • println() igual que el anterior, pero esta sí que salta a la siguiente línea y vuelve al principio de la línea, con un retorno de carro. • printf() es la salida por pantalla formateada. Permite escribir un byte por la salida estándar, sin saltar de línea, dándole formato especifico de salida. • flush() vacía el buffer de salida escribiendo su contenido.

System.err	También es una instancia de la clase PrintStream, que tiene un funcionamiento similar a System.out. Se utiliza para enviar mensajes a la salida de error estándar (por ejemplo, a la consola de error).	Se pueden utilizar los mismos métodos de System.out.

Tabla 3.1. Flujos en java.lang

C. Clases relativas a flujos

Las clases del paquete java.io relativas a flujos son:

BufferedInputStream	Permite leer datos a través de un flujo con un buffer intermedio.
BufferedOutputStream	Implementa los métodos para escribir en un flujo a través de un buffer.
FileInputStream	Permite leer bytes de un fichero o descriptor.
FileOutputStream	Permite escribir bytes en un fichero o descriptor.
StreamTokenizer	Recibe un flujo de entrada, lo analiza (del inglés, *parse*) y divide en diversos pedazos (del inglés, *tokens*), lo que permite leer uno en cada momento.
StringReader	Es un flujo de caracteres cuya fuente es una cadena de caracteres o string.
StringWriter	Es un flujo de caracteres cuya salida es un buffer de cadena de caracteres, que puede utilizarse para construir un string.

Tabla 3.2. Principales clases relativas a flujos

D. Utilización de flujos

Lectura de entrada	1. Abriremos un flujo a una fuente de datos (creación del objeto stream). Que puede ser un Teclado o un Fichero o Socket remoto, entre otros. Los iremos utilizando a lo largo del curso. 2. Mientras existan datos disponibles, iremos leyendo estos hasta llegar al final. 3. Por último, cerraremos el flujo (utilizando el método close).
Escritura de salida	1. .Abriremos un flujo a una fuente de datos (creación del objeto stream). Que puede ser una Pantalla o un Fichero o un Socket local, de una conexión de red. 2. Mientras existan datos disponibles. iremos escribiendo los datos hasta el final. 3. Por último, cerraremos el flujo (utilizando el método close).

Tabla 3.3. Utillización de flujos de entrada y salida

Como vamos a empezar a utilizar los flujos estándar, es el propio sistema quien se encargará de abrirlos y cerrarlos sin necesidad de que nos encarguemos nosotros de hacerlo.

1.3. Entrada desde el teclado

La clase Scanner de Java nos provee de métodos para leer valores de entrada de varios tipos y está localizada en el paquete java.util. Estos valores de entrada pueden proceder de varias fuentes, incluyendo valores que se introduzcan por el teclado o datos almacenados en un archivo. Para ello tenemos que crear un objeto de la clase Scanner asociado al dispositivo de entrada.

Si el dispositivo de entrada es el teclado, escribiremos:

```
Scanner teclado = new Scanner(System.in);
```

Veremos la creación de objetos con más detalle en breve. De momento, has de utilizar estas instrucciones para crear el objeto 'teclado' asociado al teclado representado por System.in

public ***Scanner*** *(InputStream source)*	Crea un nuevo Scanner a partir de un flujo de entrada de datos como es el caso de System.in (para poder leer desde el teclado).
public String ***next*** *()* *public String* ***next*** *(String pattern)*	Devuelve el siguiente elemento leído desde el teclado como un String (si coincide con el patrón especificado). Lanza NoSuchElementException si no quedan más elementos por leer.
public String ***nextLine*** *()*	Se lee el resto de la línea completa, descartando el salto de línea. Devuelve el resultado como un String. Lanza NoSuchElementException si no quedan más elementos por leer.
public int ***nextInt*** *()* *public long* ***nextLong*** *()* *public short* ***nextShort*** *()* *public byte* ***nextByte*** *()* *public float* ***nextFloat*** *()* *public double* ***nextDouble*** *()* *public boolean* ***nextBoolean*** *()*	Devuelve el siguiente elemento como un int siempre que se trate de un int. Ídem para long, short, byte, float, double y boolean. Lanza InputMismatchException en caso de no poder obtener un valor del tipo apropiado. Lanza NoSuchElementException si no quedan más elementos por leer.
public boolean ***hasNext*** *()*	Devuelve true si queda algún elemento por leer.
public boolean ***hasNextLine*** *()*	Devuelve true si queda alguna línea por leer.
public boolean ***hasNextInt*** *()* *public boolean* ***hasNextLong*** *()* *public boolean* ***hasNextShort*** *()* *public boolean* ***hasNextByte*** *()* *public boolean* ***hasNextFloat*** *()* *public boolean* ***hasNextDouble*** *()* *public boolean* ***hasNextBoolean*** *()*	Devuelve true si el siguiente elemento por obtener se puede interpretar como un int. Ídem para long, short, byte, float, double y boolean.
public Scanner ***useLocale*** *(Locale l)*	Establece la configuración local del Scanner a la configuración especificada por el Locale l.

Tabla 3.4. Principales constructores y métodos de la clase Scanner