

## 6 Programación orientada a objetos avanzada

Objetivo: Aplicar los conceptos avanzados de la programación orientada a objetos para la resolución de problemas complejos.





JAVA 2 Curso de programación. Francisco Javier Ceballos,  
2da edición Alfaomega, 2003.

2

## 6.1 Multihilos

Un hilo es un único flujo de ejecución dentro de un proceso. Un proceso es un programa en ejecución dentro de su propio espacio de direcciones.

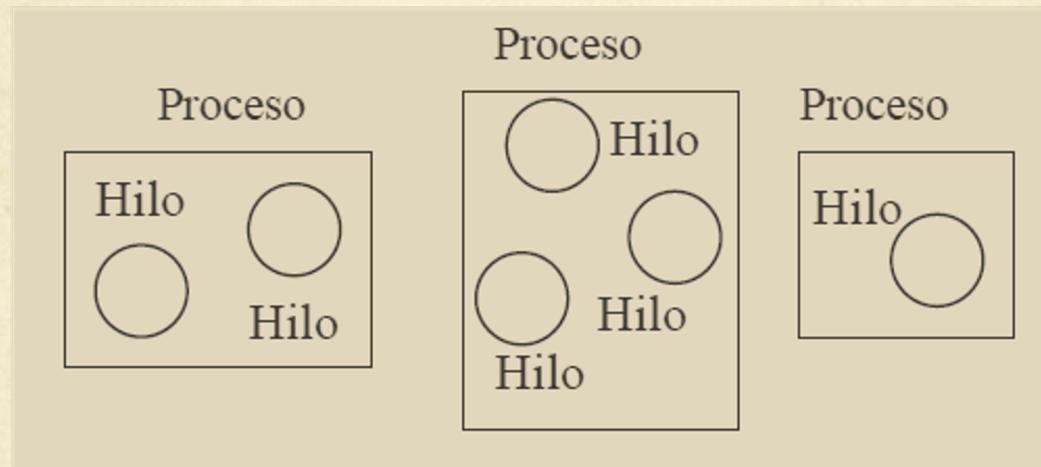
Por lo tanto, un hilo es una secuencia de código en ejecución dentro del contexto de un proceso, esto es debido a que los hilos no pueden ejecutarse solos, requieren la supervisión de un proceso.

La Máquina Virtual Java (JVM) es capaz de manejar multihilos, es decir, es posible crear varios flujos de ejecución de manera simultánea.

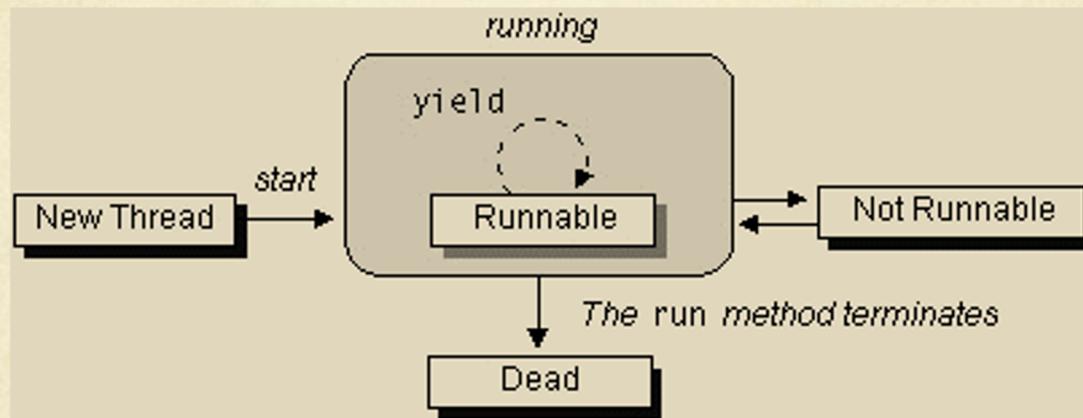
La JVM gestiona detalles como asignación de tiempos de ejecución o prioridades de los hilos, de forma similar a como gestiona un Sistema Operativo múltiples procesos.

La diferencia básica entre un proceso del Sistema Operativo y un hilo de Java es que estos últimos se ejecutan dentro de la JVM, que es, a su vez, un proceso del Sistema Operativo y, por tanto, los hilos que se ejecutan dentro de la máquina virtual comparten todos los recursos (memoria, variables y objetos).

Los procesos pueden compartir los recursos asignados, a este tipo de procesos se les llama procesos ligeros (lightweight process). La siguiente figura muestra la relación entre hilos y procesos, es decir, el Sistema Operativo ejecuta varios procesos y estos procesos a su vez ejecutan varios hilos.



## Ciclo de vida de un hilo



Como se puede observar, el campo de acción de un hilo lo compone la etapa `Runnable`, es decir, cuando se está ejecutando (corriendo) el proceso ligero.

Java proporciona soporte para hilos a través de una interfaz y un conjunto de clases. La interfaz de Java y las clases que proporcionan algunas funcionalidades sobre hilos son:

- Thread
- Runnable (interfaz)
- ThreadDeath
- ThreadGroup
- Object

Todas las clases mencionadas así como la interfaz son parte del paquete Java.lang.

## Clase Thread

Es la clase responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase solo se hereda de esta clase.

El método run define la acción de un hilo y, por lo tanto, se conoce como el cuerpo del hilo. La clase Thread también define los métodos start y stop, los cuales permiten iniciar y detener la ejecución del hilo, entre otros métodos útiles.

Para añadir la funcionalidad deseada a cada hilo creado es necesario redefinir el método `run()`. Este método es invocado cuando se inicia el hilo (mediante una llamada al método `start()` de la clase `Thread`). El hilo se inicia con la llamada al método `run()` y termina cuando termina este método llega a su fin.

**Ejemplo 6.1**

```
public class EjConThread extends Thread {  
    public EjConThread(String nombre) {  
        super(nombre);  
    }  
    public void run() {  
        for(int i = 0 ; i < 5 ; i++) {  
            System.out.println("IteraciOn " + (i+1) + " de " + getName());  
        }  
        System.out.println("Termina el " + getName());  
    }  
    public static void main(String[] args) {  
        new EjConThread("Primer hilo").start();  
        new EjConThread("Segundo Hilo").start();  
        System.out.println("Termina el hilo principal");  
    }  
}
```

Del ejemplo anterior se puede observar que el método run contiene el bloque de ejecución del hilo. El método main crea dos objetos del tipo EjConThread mandando llamar al método start (en cada caso). El método start inicia un nuevo hilo y manda llamar al método run.

Se observa que la ejecución de los tres hilos (el método principal y los hilos generados) es asíncrona.

## Interfaz Runnable

La interfaz Runnable permite producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase por medio de Runnable, solo es necesario implementar la interfaz.

La interface Runnable proporciona un método alternativo al uso de la clase Thread, para los casos en los que no es posible hacer que la clase definida herede de la clase Thread, es decir, cuando la clase definida hereda de alguna otra clase. Debido a que no existe herencia múltiple, la clase definida no podría heredar a la vez de la clase Thread y otra más. En este caso, la clase debe implantar la interface Runnable.

Las clases que implementan la interfaz Runnable proporcionan un método run que es ejecutado por un objeto hilo asociado que es creado aparte. Esta es una herramienta muy útil y, a menudo, es la única salida que tenemos para incorporar multihilo dentro de las clases.

**Ejemplo 6.2**

```
public class EjConRunnable implements Runnable {  
  
    public void run() {  
        for(int i = 0 ; i < 5 ; i++) {  
            System.out.println("IteraciOn " + (i+1) + " de " +  
                               Thread.currentThread().getName());  
        }  
        System.out.println("Termina el " +  
                           Thread.currentThread().getName());  
    }  
  
    public static void main(String[] args) {  
        new Thread(new EjConRunnable(), "Primer hilo").start();  
        new Thread(new EjConRunnable(), "Segundo Hilo").start();  
        System.out.println("Termina el hilo principal");  
    }  
}
```

Cuando se utiliza la interfaz Runnable los hilos se instancian de manera distinta. Primero se crea un objeto de la clase que implementa Runnable y después se crea una instancia de la clase Thread, pasando como parámetros el objeto creado y el nombre que tendrá el hilo. Al final se manda llamar el método start de la clase Thread y, a su vez, start mandará llamar al método run() del objeto enviado como parámetro.

Por último, en el método run para obtener el nombre del hilo que se está ejecutando se hace una llamada al método estático currentThread(), que devuelve el hilo que se está ejecutando y, a su vez, se invoca el método que obtiene su nombre.

## Clase ThreadDeath

ThreadDeath deriva de la clase Error, la cual proporciona medios para manejar y notificar errores.

Cuando el método stop de un hilo es invocado, una instancia de ThreadDeath es lanzada por el hilo como un error. Así, cuando se mata al thread de esta forma, muere de forma asíncrona. El thread moriría cuando reciba realmente la excepción ThreadDeath.

Sólo se debe recoger el objeto ThreadDeath si se necesita para realizar una limpieza específica para la ejecución asíncrona, lo cual es una situación bastante inusual. Si se recoge el objeto, debe ser relanzado para que el hilo en realidad muera.

## Clase ThreadGroup

ThreadGroup se utiliza para manejar un grupo de hilos de manera unida (en conjunto). Esto proporciona una manera de controlar de modo eficiente la ejecución de una serie de hilos.

Esta clase proporciona métodos como stop, suspend y resume para controlar la ejecución del grupo (todos los hilos del grupo).

Los hilos de un grupo pueden, a su vez, contener otros grupos de hilos permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo pero no al padre del grupo.

**Ejemplo 6.3**

```
public class EjThreadGroup extends Thread {  
  
    public EjThreadGroup(ThreadGroup g, String n){  
        super(g,n);  
    }  
  
    public void run(){  
        for (int i = 0 ; i < 10 ; i++){  
            System.out.print(getName());  
        }  
    }  
}
```

**Ejemplo 6.3**

```
public static void listarHilos(ThreadGroup grupoActual) {  
    int numHilos;  
    Thread [] listaDeHilos;  
  
    numHilos = grupoActual.activeCount();  
    listaDeHilos = new Thread[numHilos];  
    grupoActual.enumerate(listaDeHilos);  
    System.out.println("\nNúmero de hilos activos = " + numHilos + "\n");  
    for (int i = 0 ; i < numHilos ; i++) {  
        System.out.println("\nHilo activo " + (i+1) + " = "  
                           + listaDeHilos[i].getName());  
    }  
}
```

## Ejemplo 6.3

```
public static void main(String[] args) {  
    ThreadGroup grupoHilos = new ThreadGroup("Grupo con prioridad normal");  
  
    Thread hilo1 = new EjThreadGroup(grupoHilos, "Hilo 1 con prioridad mAxima");  
    Thread hilo2 = new EjThreadGroup(grupoHilos, "Hilo 2 con prioridad normal");  
    Thread hilo3 = new EjThreadGroup(grupoHilos, "Hilo 3 con prioridad normal");  
    Thread hilo4 = new EjThreadGroup(grupoHilos, "Hilo 4 con prioridad normal");  
    Thread hilo5 = new EjThreadGroup(grupoHilos, "Hilo 5 con prioridad normal");  
  
    hilo1.setPriority(Thread.MAX_PRIORITY);  
    grupoHilos.setMaxPriority(Thread.NORM_PRIORITY);
```

## Ejemplo 6.3

```
System.out.println("Prioridad del grupo = " +  
    grupoHilos.getMaxPriority());
```

```
System.out.println("Prioridad del Thread = " + hilo1.getPriority());  
System.out.println("Prioridad del Thread = " + hilo2.getPriority());  
System.out.println("Prioridad del Thread = " + hilo3.getPriority());  
System.out.println("Prioridad del Thread = " + hilo4.getPriority());  
System.out.println("Prioridad del Thread = " + hilo5.getPriority());
```

```
hilo1.start();  
hilo2.start();  
hilo3.start();  
hilo4.start();  
hilo5.start();
```

```
listarHilos(grupoHilos);
```

```
}
```

```
}
```

21

## Clase Object

Esta clase proporciona métodos cruciales dentro de la arquitectura multihilos de Java. Estos métodos son *wait*, *notify* y *notifyAll*.

El método *wait* hace que el hilo de ejecución espere en estado dormido (sleep). El método *notify* informa a un hilo en espera (*wait*) de que continúe con su ejecución. El método *notifyAll* es similar a *notify* excepto que se aplica a todos los hilos en espera.

Los métodos anteriores solo pueden ser llamados desde un método o bloque sincronizado (o bloque de sincronización).

## Ciclo de vida del hilo

### Estado new

Un hilo esta en el estado *new* la primera vez que se crea y hasta que el método *start* es llamado. Los hilos en estado *new* ya han sido inicializados y están listos para empezar a trabajar, pero aún no han sido notificados para que empiecen a realizar su trabajo.

## Estado runnable

Cuando se llama al método *start* de un hilo nuevo, el método *run* es invocado y el hilo entra en el estado *runnable* y, por tanto, el hilo se encuentra en ejecución.

Es importante tener en cuenta que los hilos manejan diferentes prioridades y, dependiendo de ellas, el hilo se ejecutará o estará en espera hasta que se libere un recurso.

## Estado not running

Se aplica a todos los hilos que están detenidos por alguna razón. Cuando un hilo está en este estado, se encuentra listo para ser usado y es capaz de volver al estado runnable en cualquier momento dado.

Los hilos pueden pasar al estado not running por diferentes métodos (suspend, sleep y wait) o por algún bloqueo de I/O.

Dependiendo de la manera en que el hilo paso al estado *not running*, se puede regresar al estado *runnable*:

- Cuando un hilo está suspendido, se invoca al método resume.
- Cuando un hilo está durmiendo, se mantendrá así el número de milisegundo especificado.
- Cuando un hilo está en espera, se activará cuando se haga una llamada a los métodos notify o notifyAll.
- Cuando un hilo está bloqueado por I/O, regresará al estado runnable cuando la operación I/O sea completada.

## Estado dead

Un hilo entra en estado dead cuando ya no es un objeto necesario. Los hilos en estado dead no pueden ser resucitados, es decir, ejecutados de nuevo. Un hilo puede entrar en estado dead por dos causas:

- El método run termina su ejecución.
- Se realiza una llamada al método stop.

## Planificador

Java tiene un Planificador (Scheduler) que controla todos los hilos que se están ejecutando en todos sus programas y decide cuáles deben ejecutarse (por tener una prioridad más alta) y cuáles deben encontrarse preparados para su ejecución (en estado wait o suspend).

La decisión de qué hilos se ejecutan primero y cuales tienen que esperar la realiza java con base en la prioridad que tiene asignado cada hilo.

28

## Prioridad

Cada hilo tiene una prioridad, que no es otra cosa que un valor entero entre 1 y 10, de modo que cuanto mayor el valor, mayor es la prioridad.

El planificador determina que hilo debe ejecutarse en función de la prioridad asignada a cada uno de ellos. Cuando se crea un hilo en Java, éste hereda la prioridad de su padre. Una vez creado el hilo es posible modificar su prioridad en cualquier momento utilizando el método `setPriority`.

Las prioridades de un hilo varían en un rango de enteros comprendido entre `MIN_PRIORITY` y `MAX_PRIORITY` (estas variables están definidas en la clase `Thread`).

El planificador ejecuta primero el hilo de prioridad superior (en estado `runnable`) y sólo cuando éste se detiene, comienza la ejecución de en hilo de prioridad inferior. Si dos hilos tienen la misma prioridad, el programador elige entre ellos de manera alternativa (forma de competición).

El hilo seleccionado se ejecutará hasta que:

- Un hilo con prioridad mayor pase a ser “Ejecutable”.
- En sistemas que soportan tiempo-compartido, termina su tiempo.
- Abandone, o termine su método run.

Luego, un segundo hilo puede ejecutarse, y así continuamente hasta que el intérprete abandone.

El algoritmo del sistema de ejecución de hilos que sigue Java es de tipo preventivo. Si en un momento dado un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando pasa a ser “Ejecutable”, entonces el sistema elige a este nuevo hilo.

## Métodos sincronizables

Los métodos sincronizables (`synchronized`) son aquellos a los que es imposible acceder si otro objeto está haciendo uso de ellos, es decir, dos o objetos no pueden acceder a un método sincronizable al mismo tiempo.

Estos tipos de métodos son muy utilizados en hilos (thread) para evitar la pérdida de información o ambigüedades en la misma.

32

Los hilos se ejecutan como procesos paralelos (independientes). Cuando se ejecutan varios hilos de manera simultanea, estos pueden intentar acceder al mismo tiempo a un método, para, por ejemplo, obtener o modificar el valor de un atributo.

Como los hilos se ejecutan en paralelo, el acceso al recurso no es controlado y, por lo tanto, puede haber pérdida de información.

Por ejemplo, si se tiene un hilo que está guardando información en un arreglo por medio de un método y, al mismo tiempo, otro hilo ejecuta el mismo método y comienza a escribir en el mismo arreglo, esto generaría pérdida de datos y ambigüedad en la información para el primer hilo.

Estas complicaciones se pueden evitar bloqueando el acceso al método mientras algún proceso lo esté ejecutando. Lo anterior se puede lograr haciendo que el método en cuestión sea sincronizable (synchronizable).

Así, al estar sincronizados los métodos de una clase, si un recurso está ocupando un método, éste es inaccesible hasta que sea liberado, es decir, dos hilos no pueden acceder a un mismo método al mismo tiempo.

**Ejemplo 6.4**

```
public class Cuenta extends Thread {  
  
    private static long saldo = 0;  
  
    public Cuenta (String nombre){  
        super(nombre);  
    }  
  
    public void run() {  
        if (getName().equals("Deposito 1") ||  
            getName().equals("Deposito 2")) {  
            this.depositarDinero(100);  
        } else {  
            this.extraerDinero(50);  
        }  
        System.out.println("Termina el " + getName());  
    }  
}
```

**Ejemplo 6.4**

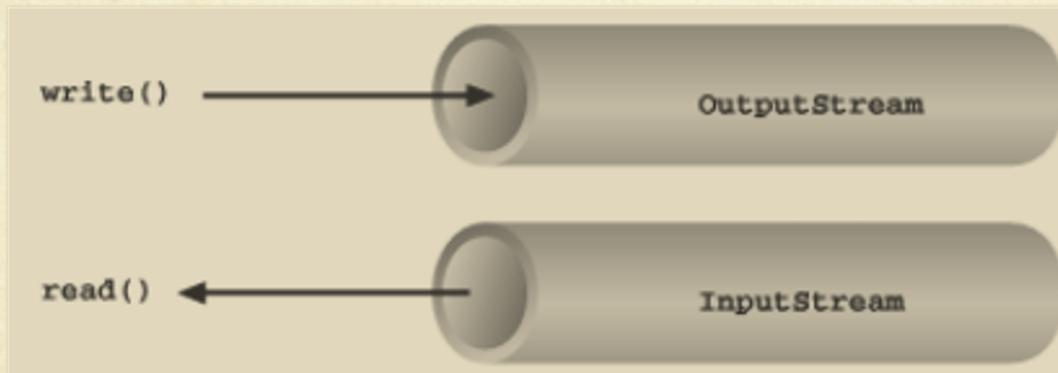
```
public synchronized void extraerDinero(int cantidad) {  
    try {  
        if (saldo <= 0){  
            System.out.println(getName() + " espera depOsito.  
                                \nSaldo = " + saldo);  
            sleep(5000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println(e);  
    }  
    saldo -= cantidad;  
    System.out.println(getName() + " extrajo " + cantidad + "  
                        pesos.\nSaldo restante = "+ saldo);  
    notifyAll();  
}
```

**Ejemplo 6.4**

```
public synchronized void depositarDinero(int cantidad) {  
    saldo += cantidad;  
    System.out.println("Se depositaron " + cantidad + " pesos");  
    notifyAll();  
}  
  
public static void main(String[] args) {  
    new Cuenta("Acceso 1").start();  
    new Cuenta("Acceso 2").start();  
    new Cuenta("Deposito 1").start();  
    new Cuenta("Deposito 2").start();  
    System.out.println("Termina el hilo principal");  
}  
}
```

## 6.2 Flujos de datos

Las entradas y las salidas de datos en java se manejan mediante streams (flujos de datos). Un stream es una conexión entre el programa y la fuente (lectura) o el destino (escritura) de los datos. La información se traslada en serie a través de esta conexión.



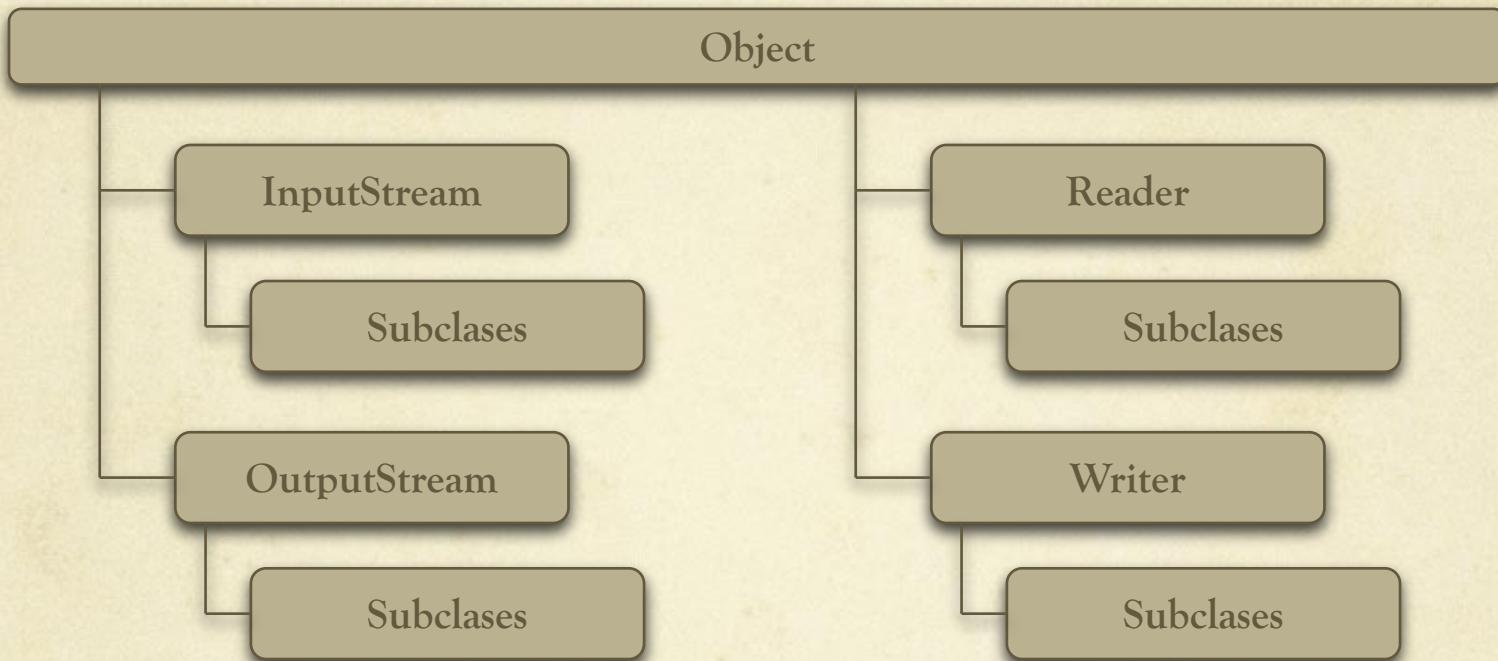
Existen 4 jerarquías de clases relacionadas con la entrada y salida de datos.

Por un lado se encuentran las clases derivadas de `InputStream` (para lectura) y de `OutputStream` (para escritura). Estas clases manejan streams de bytes.

Por otro lado se encuentra las clases derivadas de `Reader` y `Writer`, que manejan caracteres en vez de bytes.

Todas las clases de Java relacionadas con la entrada y salida se agrupan en el paquete `java.io`.

## Jerarquía de clases de los flujos de E/S



40

## Clase File

La clase file permite manejar archivos o carpetas, es decir, crear y borrar tanto archivos como carpetas, entre otras funciones.

Cabe resaltar que cuando se crea una instancia de la clase File no se crea ningún archivo o directorio, solo se instancia un objeto de este tipo. La creación de archivos o carpetas se realizan al momento de invocar el método respectivo: `createNewFile()` o `mkdir()`.

Algunos métodos que posee la clase File son los siguientes:

- exists()
- createNewFile()
- mkdir()
- delete()
- renameTo()
- list()

42

**Ejemplo 2.1**

```
import java.io.*;
class Escribe {
    public static void main(String [] args) {
        try {
            File archivo = new File("archivo.txt");
            System.out.println(archivo.exists());
            boolean seCrea = archivo.createNewFile();
            System.out.println(seCrea);
            System.out.println(archivo.exists());
        } catch(IOException e) { }
    }
}
```

Al ejecutarse, el código emite la siguiente salida:

false  
true  
true

43

## Lectura y escritura de archivos

Para leer o escribir flujos de bytes desde o hacia un archivo se utilizan las clases `FileOutputStream` y `FileInputStream`, que son subclases de `OutputStream` e `InputStream`, respectivamente.

Para leer o escribir flujos de caracteres desde o hacia un archivo se utilizan las clases `FileWriter` y `FileReader` que son subclases de `Writer` y `Reader`, respectivamente.

## FileOutputStream

La clase FileOutputStream permite escribir bytes en un archivo. Esta clase hereda los métodos de la clase OutputStream y, además, posee sobrecarga de constructores:

FileOutputStream (String nombre)

FileOutputStream (String nombre, boolean añadir)

FileOutputStream (File archivo)

El primer constructor abre un flujo de salida hacia el archivo especificado (nombre). El segundo constructor también abre un flujo de salida hacia el archivo especificado y, si el archivo existe, permite agregar datos al mismo (añadir = true). El tercer constructor abre un flujo de salida a partir de un objeto File.

**Ejemplo 6.6**

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

public class ClaseOutputStream {
    public static void main (String [] args){
        FileOutputStream fos = null;
        byte[] buffer = new byte[81];
        int nBytes;
        try {
            System.out.println("Escribir el texto a guardar en el archivo:");
            nBytes = System.in.read(buffer);
            fos = new FileOutputStream("fos.txt");
            fos.write(buffer,0,nBytes);
        } catch (IOException ioe){
            System.out.println("Error: " + ioe.toString());
        }
    }
}
```

**46**

## Ejemplo 6.6

En el ejemplo anterior se define un arreglo llamado buffer de 81 bytes donde se va a almacenar lo que se capture del teclado.

Después se crea el flujo de bytes mediante la clase `FileOutputStream` y se redirige al archivo de nombre `fos.txt`.

Al final, se hace uso del método `write` de la clase `FileOutputStream` para escribir los datos del buffer en el archivo, que recibe como primer parámetro la referencia a la matriz que contiene los bytes a escribir, como segundo parámetro la posición de la matriz del primer byte y como último parámetro el número de bytes a escribir.

El método `write` es un método sobrecargado. Los otros usos del mismo se pueden consultar en el API de java.

## Ejemplo 6.6

Es una buena costumbre cerrar el flujo de datos cuando éste ya no se vaya a utilizar. Para ello se puede ocupar el bloque finally

```
finally {  
    try {  
        if (fos != null)  
            fos.close();  
    } catch (IOException ioe){  
        System.out.println("Error : " + ioe.toString());  
    }  
}
```

48

## FileInputStream

FileInputStream permite leer flujos de bytes desde un archivo. Hereda de la clase InputStream y, además, posee diferentes constructores (sobrecarga):

FileInputStream(String nombre)

FileInputStream(File archivo)

donde, el primer constructor abre un flujo de entrada desde el archivo especificado por nombre y el segundo constructor abre un flujo de entrada a partir de un objeto del tipo File.

**Ejemplo 6.7**

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

public class ClaseInputStream {
    public static void main (String [] args){
        FileInputStream fis = null;
        byte[] buffer = new byte[81];
        int nbytes;
        try {
            fis = new FileInputStream("leer.txt");
            nbytes = fis.read(buffer, 0, 81);
            String texto = new String(buffer, 0, nbytes);
            System.out.println(texto);
        }
    }
}
```

**Ejemplo 6.7**

```
catch (IOException ioe) {  
    System.out.println("Error: " + ioe.toString());  
} finally {  
    try {  
        if (fis != null) fis.close();  
    } catch (IOException ioe) {  
        System.out.println("Error al cerrar el archivo.");  
    }  
}  
}  
}
```

Para el ejemplo anterior, se define un flujo que va a leer desde el archivo con nombre leer.txt (si no existe el archivo se genera una excepción). También se crea un buffer de 81 bytes.

FileInputStream lee el texto desde el archivo y lo almacena en el buffer creado. Se lee el archivo hasta que se terminen los caracteres del mismo o hasta que se llene el buffer (desde la posición 0 hasta la 81), lo primero que ocurra. El método read devuelve el número de bytes leídos o -1 si se finalizó de leer el archivo.

## FileWriter

La clase `FileWriter` hereda de `Writer` y permite escribir caracteres en un archivo.

La clase `BufferedWriter` también deriva de la clase `Writer` y permite añadir un buffer para realizar una escritura eficiente de caracteres.

La clase `PrintWriter`, que también deriva de `Writer`, permite escribir de forma sencilla en un archivo de texto, ya que posee los métodos `print` y `println`, idénticos a los de `System.out`. El método `close()` cierra el stream.

Para escribir una cadena de caracteres en un archivo de texto se puede utilizar el siguiente código:

```
// Se instancia la clase que permite crear/leer archivos  
// indicando el nombre del archivo  
FileWriter fw = new FileWriter("archivo.txt");  
  
// Se crea un buffer de escritura de caracteres hacia el archivo  
BufferedWriter bw = new BufferedWriter(fw);  
  
// Se instancia la clase que permite escribir en el archivo  
// indicando el buffer de escritura  
PrintWriter salida = new PrintWriter(bw);  
  
// Se escribe en el archivo  
salida.println(texto);  
  
// Se cierra el buffer y el archivo  
salida.close();
```

## Ejemplo 6.8

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.IOException;

public class ClaseFileWriter{
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();
            FileWriter fw = new FileWriter("archivo.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter salida = new PrintWriter(bw);
            salida.println(texto);
            salida.close();
        }
    }
}
```

55

**Ejemplo 6.8**

```
catch (IOException ioe){  
    System.out.println("\n\nError al abrir o guardar el archivo:");  
    ioe.printStackTrace();  
} catch (Exception e){  
    System.out.println("\n\nError al leer de teclado:");  
    e.printStackTrace();  
}  
}
```

## Ejemplo 6.8 (2)

```
try{  
    System.out.println("Este programa escribe un arreglo de dos dimensiones");  
    System.out.println("en un archivo de texto llamado for.txt");  
    FileWriter fw = new FileWriter("for.txt",true);  
    BufferedWriter bw = new BufferedWriter(fw);  
    PrintWriter salida = new PrintWriter(bw);  
    int [][] arreglo = new int[5][];  
    int a = 0;  
    for (int i = 0 ; i < arreglo.length ; i++ ){  
        arreglo[i] = new int[2*i];  
        for (int j = 0 ; j < arreglo[i].length ; j++){  
            arreglo[i][j] = a++;  
            salida.print(arreglo[i][j]);  
        }  
        salida.println("");  
    }  
    salida.close();  
}
```

## FileReader

Las clases Reader se utilizan para obtener los caracteres ingresados desde la entrada estándar.

La clase FileReader hereda de Reader y permite leer caracteres de un archivo.

InputStreamReader es una clase derivada de Reader que convierte los streams de bytes a streams de caracteres, es decir, lee bytes y los convierte en caracteres. System.in es el objeto de la clase InputStream para recibir datos desde la entrada estándar del sistema (el teclado).

**Ejemplo 6.9**

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

public class ClaseFileReader{
    public static void main (String [] escribir){
        String texto = "";
        try {
            BufferedReader br;
            FileReader fr = new FileReader("leer.txt");
            br = new BufferedReader(fr);
            System.out.println("El texto contenido en el archivo leer.txt es:");
            String linea = br.readLine();
            while (linea != null ) {
                System.out.println(linea);
                linea = br.readLine();
            }
        }
    }
}
```

**Ejemplo 6.9**

```
catch (IOException ioe){  
    System.out.println("\n\nError al abrir o guardar el archivo:");  
    ioe.printStackTrace();  
} catch (Exception e){  
    System.out.println("\n\nError al leer de teclado:");  
    e.printStackTrace();  
}  
}
```

60

## BufferedReader

La clase BufferedReader, que también deriva de la clase Reader, añade un buffer para realizar una lectura eficiente de caracteres. Dispone del método readLine que permite leer una línea de texto y tiene como valor de retorno un String.

System.in es el objeto de la clase InputStream para recibir datos desde la entrada estándar del sistema (el teclado).

Para leer una cadena de texto utilizando esta clase se realiza lo siguiente:

```
String texto = "";  
BufferedReader br;  
br = new BufferedReader(new InputStreamReader(System.in));  
texto = br.readLine();
```

## Ejemplo 6.10

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class LeeTeclado {
    public static void main (String [] args){
        try {
            String texto = "";
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir el texto deseado:");
            texto = br.readLine();
            System.out.println("El texto escrito fue: " + texto);
        } catch (IOException ioe){
            System.out.println("Error al leer caracteres: \n" + ioe);
        }
    }
}
```

## StringTokenizer

La clase StringTokenizer permite separar una cadena de texto por palabras (espacios) o por algún otro carácter. La clase StringTokenizer pertenece al paquete java.util.

El código para separar una cadena de texto es el siguiente:

```
StringTokenizer st = new StringTokenizer(texto);
while(st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

El método countTokens() devuelve el número de tokens que se pueden extraer de la frase.

## Ejemplo 6.11

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.PrintWriter;
import java.util.StringTokenizer;

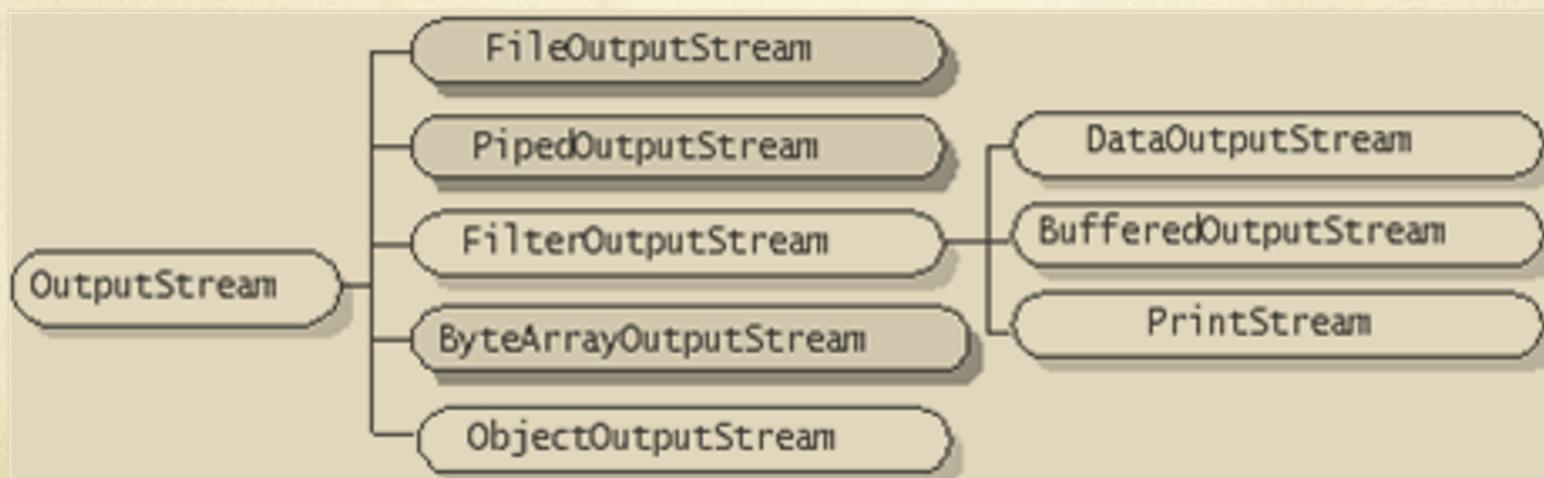
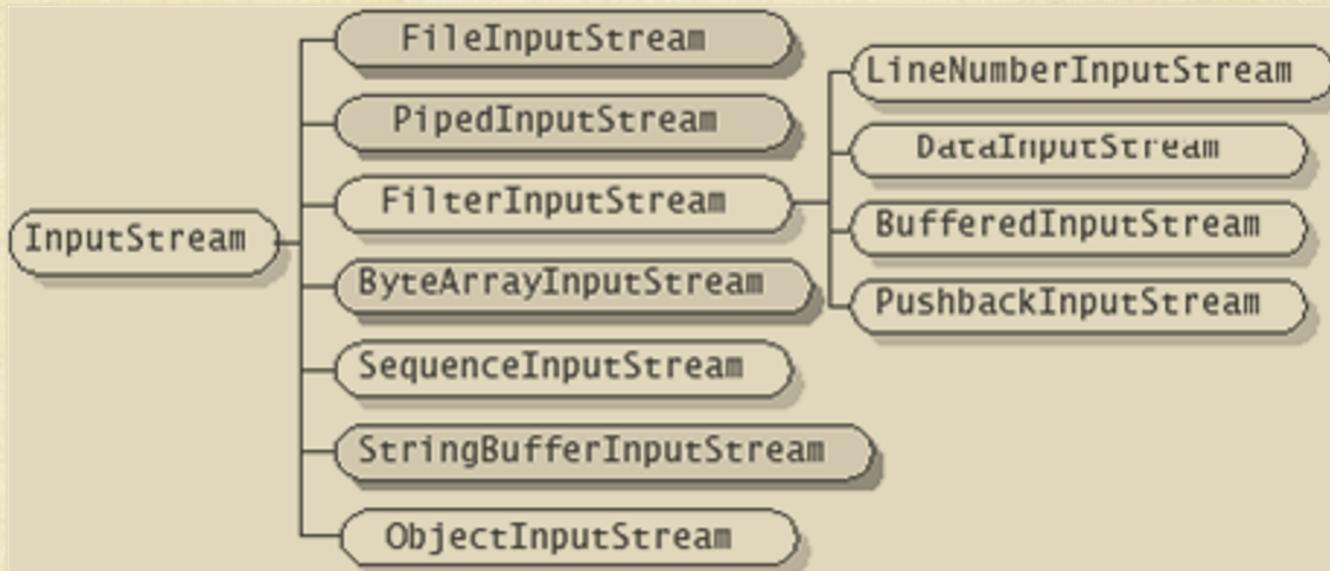
public class LeeTecladoCompleto {
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();
            System.out.println("\n\nEl texto separado por espacios es:");
            StringTokenizer st = new StringTokenizer(texto);
            while(st.hasMoreTokens()) {
                System.out.println(st.nextToken());
            }
        }
    }
}
```

## Ejemplo 6.11

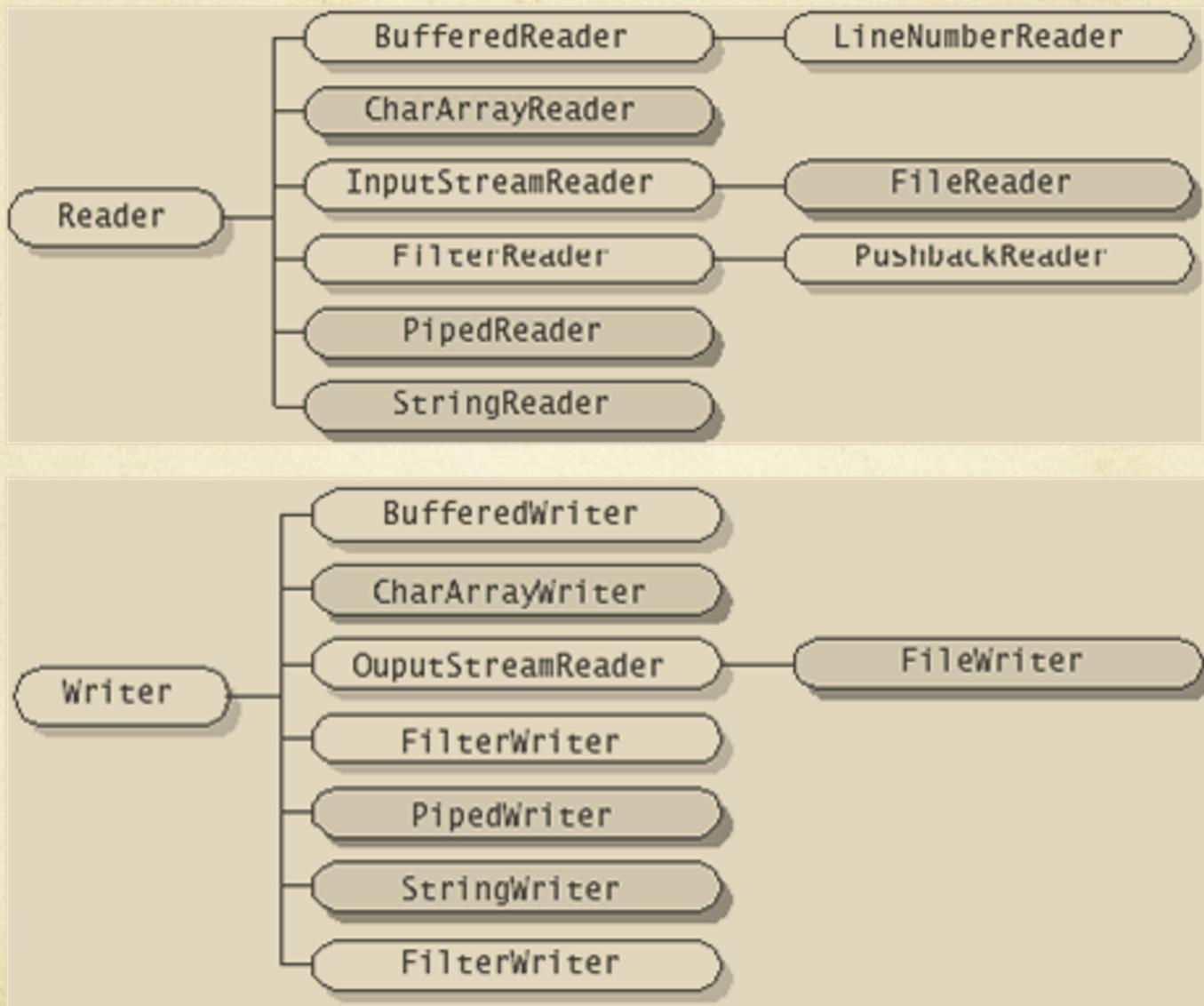
```
br.readLine();
System.out.println("\n\nEl texto completo es:");
System.out.println(texto);

FileWriter fw = new FileWriter("archivo.txt");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter salida = new PrintWriter(bw);
salida.println(texto);
salida.close();
System.out.println("\n\nEl texto se guardó en archivo.txt");
} catch (IOException ioe){
    System.out.println("\n\nError al abrir o guardar el archivo:");
    ioe.printStackTrace();
} catch (Exception e){
    System.out.println("\n\nError al leer de teclado:");
    e.printStackTrace();
}
}
```

## Jerarquía de clases para lectura y escritura de datos



## Jerarquía de clases para lectura y escritura de datos



## Scanner

La clase Scanner permite leer de la entrada estándar. Pertenece al paquete java.util.

Los métodos principales de esta clase son next() y hasNext(). El método next() obtiene el siguiente elemento del flujo de datos. El método hasNext() verifica si el flujo de datos todavía posee elementos, en caso afirmativo regresa true, de lo contrario regresa false.

El delimitador de la clase scanner, por defecto, es el espacio en blanco. Es posible cambiar el delimitador utilizando el método:

```
useDelimiter(String cadena);
```

## Ejemplo 6.12

```
import java.util.Scanner;

public class EjScanner {
    public static void main(String [] args) {
        try {
            String cad = "";
            Scanner s = new Scanner(System.in);
            System.out.println(s.nextLine());
            s.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

## Console

La clase **Console** permite recibir datos de la línea de comandos (entrada estándar). Se encuentra dentro del paquete `java.io`.

Entre los métodos importantes que posee están:

- `readLine()`: lee una cadena de caracteres hasta que encuentra el salto de línea (enter).
- `readPassword()`: lee una cadena de caracteres hasta que encuentra el salto de línea (enter), ocultando los caracteres.

## Ejemplo 6.13

```
import java.io.Console;

public class EjConsole {
    public static void main(String [] args){
        Console con = System.console();
        System.out.print("Usuario: ");
        String s = con.readLine();
        System.out.println(s);
        System.out.print("Contraseña: ");
        char [] s2 = con.readPassword();
        System.out.println(s2);
    }
}
```

## Formato de cadenas

Es posible dar formato a las cadenas de texto, para ello se cuenta con un conjunto de expresiones regulares:

- `%n` inserta un salto de línea
- `%s` inserta una cadena de caracteres
- `%d` inserta un número entero
- `%f` inserta un número flotante

La sintaxis para dar formato a un tipo de dato es la siguiente:

`%[indiceArg$][banderas][tamaño].[precisión]tipoDato`

**Ejemplo 6.14**

```
public class Formato {  
    public static void main(String [] args){  
        String n = "Jav";  
        int a = 25;  
        float y = 100.344f;  
        System.out.println("Formato de cadenas");  
        String s = String.format("%s %5d%on %o%on %f",n,a,a,y);  
        System.out.println(s);  
    }  
}
```

## Ejemplo 6.15

```
public class Formato2 {  
    public static void main(String [] args){  
        int i1 = -123;  
        int i2 = 12345;  
  
        System.out.println("PRINTF");  
        System.out.printf(">%1$(7d< \n",i1);  
        System.out.printf(">%0,7d< \n",i2);  
        System.out.printf(">%+-7d< \n",i1);  
        System.out.printf(">%2$b + %1$5d< \n",i1,false);  
  
        System.out.println("FORMAT");  
        System.out.format(">%1$(7d< \n",i1);  
        System.out.format(">%0,7d< \n",i2);  
        System.out.format(">%+-7d< \n",i1);  
        System.out.format(">%2$b + %1$5d< \n",i1,false);  
    }  
}
```

**Tarea 6.1**

## Validar usuario

Crear un programa en java que permita validar si un usuario está dado de alta en el banco de datos de un sistema. En caso de que el usuario ya esté en el sistema, el programa debe validar las credenciales de éste para permitir el acceso al mismo.

Los usuarios y sus contraseñas se almacenan en un archivo de texto.

**Tarea 6.2**

### Revisar texto

Una vez que el usuario esté dentro del sistema, es posible que imprima algún pensamiento en el mismo. Para ello puede escribir y/o leer un archivo de publicaciones general (todos pueden escribir en él).

Las publicaciones se guardan en un archivo de texto. El archivo, como es accesible para todos, se debe ingresar en un método sincronizado (para evitar ambigüedades).

## Sockets

Los sockets son un sistema de comunicación entre procesos de diferentes máquinas en una red.

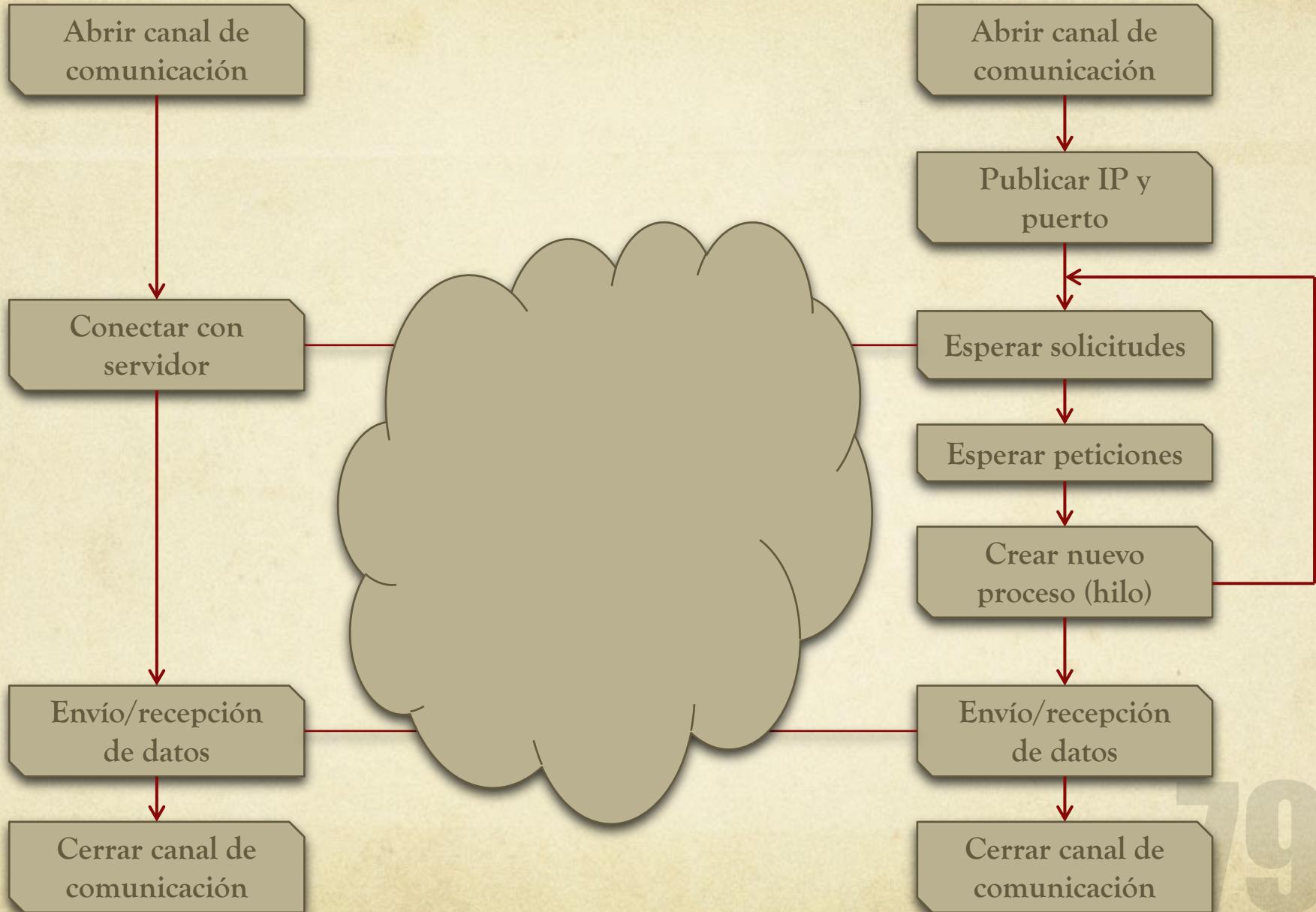
Más exactamente, un socket es un punto de comunicación por el cual un proceso puede enviar o recibir información (flujos de datos).

Los sockets se clasifican en sockets de flujo o sockets de datagramas dependiendo de el servicio que utiliza: TCP (Transfer Control Protocol) o UDP (User Datagram Protocol).

El protocolo TCP se conoce como orientado a la conexión y es muy fiable. El protocolo UDP no es tan confiable ya que puede haber pérdida de información debido a que no se asegura que el mensaje llegue a su destino.

# Diagrama de dialogo entre sockets (Cliente-Servidor)

6.2 Flujos de datos



Por lo general, un servicio o programa se ejecuta en una computadora sobre un puerto específico, a la espera de solicitudes del servicio. El programa que está a esperando peticiones se llama servidor.

Por otro lado, existe un programa que se intenta conectar con el servidor, para ello debe especificar la ruta donde se está ejecutando el servicio (dirección IP) así como el puerto por el que el servicio está a la escucha. Este programa se conoce como cliente.

## Arquitectura Cliente-Servidor



81

Cuando el servidor acepta la conexión del cliente se abre un nuevo socket sobre otro puerto para mantener activa la conexión con el cliente, mientras sigue recibiendo peticiones por el puerto establecido, a la espera de otros clientes.

En el cliente se lleva a cabo un proceso similar, se crea un socket con la conexión establecida hacia el servidor por el que ambos pueden seguir comunicados (flujo de datos). El socket en el cliente también se levanta en otro puerto.

Una vez establecida la conexión con un cliente, el servidor sigue a la escucha de otras peticiones



## Sockets en java

En java, la clase Socket se encuentra definida en el paquete java.net, la cual implementa una parte de la comunicación bidireccional entre un programa java y otro. Estos sockets pueden comunicarse a través de la red de forma totalmente independiente de la plataforma donde se ejecute.

Además, java.net también posee una clase llamada ServerSocket, que implementa un socket que es capaz de estar al escucha de peticiones de conexiones por parte de los clientes.

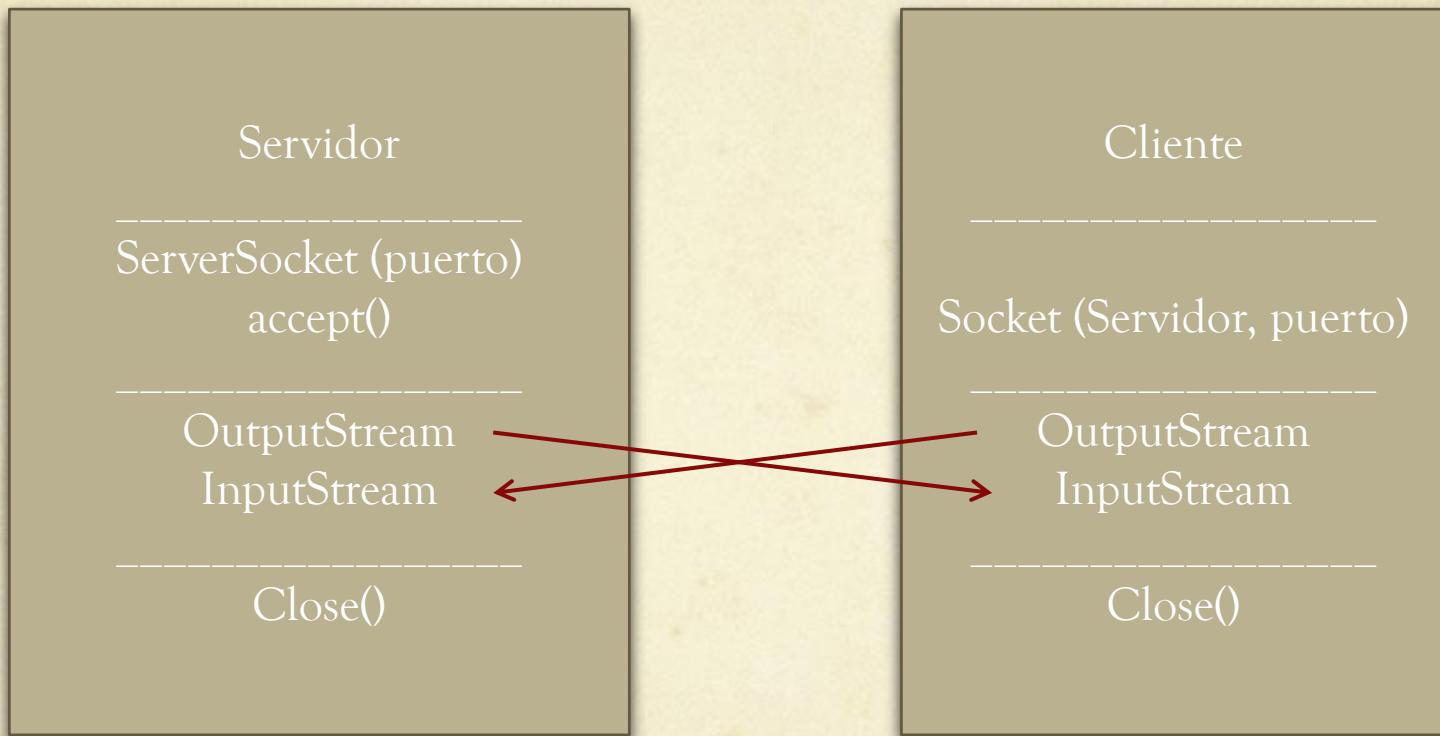
## Modelo de comunicaciones

El modelo de comunicaciones más básico entre sockets es el que envuelve a un servidor y un cliente.

En este caso, el servidor espera la petición del cliente en cierto puerto durante cierto tiempo. Cuando el cliente realice la petición, el servidor abrirá la conexión.

Por otra parte, el cliente se intentará conectar con el servidor por el puerto establecido y, de ser aceptada la conexión, ambos pueden intercambiar información mediante flujos de datos (`InputStream` y `OutputStream`).

## Modelo de comunicación básico



## Rango de puertos IP

Existen una cantidad enorme de puertos (65535) para cada dirección IP. Sin embargo, hay puertos que no se pueden utilizar porque están designados a servicios del sistema, por eso es importante conocer los rangos de puertos existentes y saber cuáles se pueden utilizar.

- Puertos conocidos o reservados (0 al 1023): Están reservados para procesos del sistema (demonios).
- Puertos registrados (1024 al 49151): Son de libre utilización.
- Puertos dinámicos o privados (49152 al 65535): Son de tipo temporal.

## Cliente

La sintaxis para crear un socket cliente en java es la siguiente:

```
Socket cliente;  
try {  
    cliente = new Socket("Servidor", puerto);  
} catch (IOException ioe){  
    System.out.println(e.getMessage());  
}
```

donde Servidor se refiere al nombre (o ip) del servidor a conectar y el puerto por donde está a la escucha de peticiones.

## Flujos de entrada

`DataInputStream` permite crear flujos de entrada para recibir los datos que envíe el servidor. Su sintaxis es la siguiente:

```
Socket cliente;  
DataInputStream entrada;  
try {  
    cliente = new Socket("Servidor", puerto);  
    entrada = new DataInputStream(cliente.getInputStream());  
} catch(IOException e) {  
    System.out.println( e );  
}
```

La clase `DataInputStream` permite leer diferentes tipos de datos primitivos: `read()`, `readChar()`, `readInt()`, `readDouble()` y `readLine()`.

## Flujos de salida

Las clases PrintStream y DataInputStream permiten crear flujos de salida para enviar información. La sintaxis para crear un flujo de datos de salida por medio de la clase PrintStream es la siguiente:

```
Socket cliente;  
DataInputStream entrada;  
PrintStream salida;  
try {  
    cliente = new Socket("Servidor", puerto);  
    salida = new PrintStream(cliente.getOutputStream());  
} catch(IOException e) {  
    System.out.println( e );  
}
```

90

La sintaxis para crear un flujo de datos de salida por medio de la clase `DataOutputStream` es la siguiente:

```
Socket cliente;
DataInputStream entrada;
DataOutputStream salida;
try {
    cliente = new Socket("Servidor", puerto);
    salida = new DataOutputStream(cliente.getOutputStream());
} catch(IOException e) {
    System.out.println( e );
}
```

Tanto la clase `PrintStream` como `DataOutputStream` permite escribir cualquier tipo de dato primitivo.

Al final, es necesario cerrar todos los flujos de entrada y de salida, así como los sockets generados. Primero se deben cerrar los flujos de datos y al final el socket.

```
Socket cliente;
DataInputStream entrada;
DataOutputStream salida;
try {
    cliente = new Socket("Servidor", puerto);
    entrada = new DataInputStream(cliente.getInputStream());
    salida = new DataOutputStream(cliente.getOutputStream());
    entrada.close();
    salida.close();
    cliente.close();
} catch(IOException e) {
    System.out.println(e);
}
```

## Servidor

Para crear un servidor con sockets es necesario crear un socket cliente y un socket servidor. La sintaxis es la siguiente:

```
Socket servicio;  
ServerSocket servidor;  
try {  
    servidor = new ServerSocket(puerto);  
    servicio = servidor.accept();  
} catch (IOException ioe){  
    System.out.println(e.getMessage());  
}
```

En el ejemplo anterior se ve claramente como cada vez que el servidor acepta una petición, crea un socket nuevo para seguir comunicado con el cliente.

## Flujos de entrada

DataInputStream permite recibir flujos de entrada envíe el cliente. La sintaxis es la misma que para el cliente:

```
Socket servicio;  
ServerSocket servidor;  
DataInputStream entrada;  
try {  
    servidor = new ServerSocket(puerto);  
    servicio = servidor.accept();  
    entrada = new DataInputStream(servidor.getInputStream());  
} catch(IOException e) {  
    System.out.println(e);  
}
```

## Flujos de salida

Las clases PrintStream y DataInputStream permiten crear flujos de salida para enviar información. La sintaxis para crear un flujo de datos de salida por medio de la clase PrintStream es la siguiente:

```
Socket servicio;  
ServerSocket servidor;  
PrintStream salida;  
try {  
    servidor = new ServerSocket(puerto);  
    servicio = servidor.accept();  
    salida = new PrintStream(servidor.getOutputStream());  
} catch(IOException e) {  
    System.out.println( e );  
}
```

Al final, al igual que en el cliente es necesario cerrar todos los flujos de entrada, de salida y los sockets generados. Primero se deben cerrar los flujos de datos y al final el socket.

```
Socket servicio;
ServerSocket servidor;
DataInputStream entrada;
PrintStream salida;
try {
    servidor = new ServerSocket(puerto);
    servicio = servidor.accept();
    entrada = new DataInputStream(servidor.getInputStream());
    salida = new PrintStream(servidor.getOutputStream());
    salida.close();
    entrada.close();
    servidor.close();
    servicio.close();
} catch(IOException e) {
    System.out.println(e);
}
```

## Ejemplo 6.16

```
import java.io.OutputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class SocketServidor {
    static final int PUERTO = 5432;

    public SocketServidor() {
        try {
            ServerSocket servidor = new ServerSocket(PUERTO);
            System.out.println("Esperando peticiones por el puerto " + PUERTO);
            for (int clientes = 0 ; clientes < 5; clientes++) {
                Socket servicio = servidor.accept();
                System.out.println("Se aceptó la conexión del cliente "
                        + clientes);
                OutputStream escribir = servicio.getOutputStream();
                DataOutputStream flujoDatosSalida =
                    new DataOutputStream(escribir);
                flujoDatosSalida.writeUTF("Bienvenido cliente " + clientes);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

**Ejemplo 6.16**

```
DataStream flujoDatosEntrada =  
    new DataInputStream(servicio.getInputStream());  
System.out.println("El cliente " + clientes + " dice: "  
    + flujoDatosEntrada.readUTF());  
  
flujoDatosSalida.close();  
flujoDatosEntrada.close();  
servicio.close();  
}  
System.out.println("Demasiados clientes por hoy.");  
} catch(Exception e) {  
    System.out.println(e.getMessage());  
    e.printStackTrace();  
}  
}  
  
public static void main(String [] arg) {  
    new SocketServidor();  
}  
}
```

Ejemplo 6.16

```
public class SocketCliente {  
    static final String SERVIDOR = "localhost";  
    static final int PUERTO = 5432;  
    public SocketCliente() {  
        try{  
            Socket con = new Socket(SERVIDOR, PUERTO);  
            InputStream leer = con.getInputStream();  
            DataInputStream flujoDatosEntrada =  
                new  
DataInputStream(leer);  
            System.out.println(flujoDatosEntrada.readUTF());  
            DataOutputStream flujoDatosSalida =  
                new DataOutputStream(con.getOutputStream());  
            flujoDatosSalida.writeUTF("Gracias por aceptarme.");  
            flujoDatosEntrada.close();  
            flujoDatosSalida.close();  
            con.close();  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    public static void main(String [] arg) {  
        new SocketCliente();  
    }  
}
```

### 6.3 Resolución de problemas complejos

Elaborar programas utilizando los conceptos vistos anteriormente:

- Hilos (Threads)
- Flujos (entrada y salida) de datos.

100

Tema 6: Aplicar los conceptos avanzados de la programación orientada a objetos para la resolución de problemas complejos.

- Subtema 6.1: Explicar el concepto de hilo y su funcionamiento.
- Subtema 6.2: Describir las diferentes maneras en que los datos son manipulados.
- Subtema 6.3: Elaborar programas utilizando los conceptos vistos anteriormente.

101

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

```
namespace SP {
    public partial class Form2 : Form {
        string [] datos;
        Logica logica;
        Form1 f1;
        public Form2(string [] campos, Form1 form) {
            InitializeComponent();
            datos = campos;
            logica = new Logica();
            f1 = form;
        }
    }
```

Análisis

102

```
private void button1_Click(object sender, EventArgs e) {  
    if (textBox1.Text.ToString() == "") {  
        MessageBox.Show("Favor de insertar un Título");  
    } else {  
        if (textBox2.Text.ToString() == "") {  
            MessageBox.Show("Favor de insertar las horas");  
        } else {  
            datos[3] = textBox1.Text.ToString();  
            datos[5] = textBox2.Text.ToString();  
            int res = logica.actualizar(datos);  
            if (res == 0) {  
                MessageBox.Show("Campo " + datos[0] + " actualizado.");  
                f1.button6_Click_1(sender, e);  
            } else {  
                MessageBox.Show("Error al actulizar los datos.\nError = "  
                               + logica.error);  
            }  
            this.Close();  
        }  
    }  
}
```

```
private void Form2_Load(object sender, EventArgs e) {  
    label2.Text = datos[0];  
    textBox1.Text = datos[3];  
    textBox2.Text = datos[5];  
}  
}  
}
```

104