



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Programación orientada a objetos
Tema 6:
FLUJO DE ENTRADA Y SALIDA

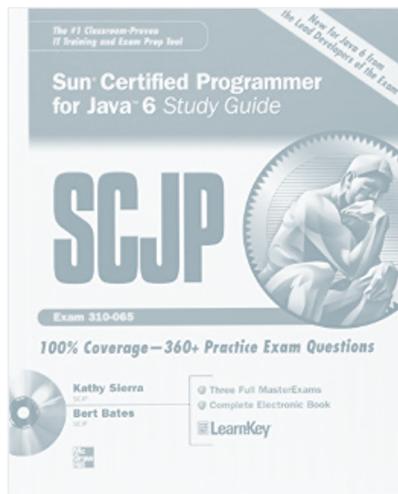
6 Flujo de entrada y salida

Objetivo: Construir programas con el principio de flujo de entrada y salida para procesar información a partir de un problema resuelto.

6 Flujo de entrada y salida

- 6.1 Fundamentos de entrada y salida.
- 6.2 Jerarquía de clases de los flujos de datos.
- 6.3 Manipulación de archivos y carpetas.
- 6.4 Flujos de entrada de datos.
 - 6.4.1 Lectura de archivo.
 - 6.4.2 Lectura de teclado.
- 6.5 Flujos de salida de datos (escritura de archivo).
- 6.6 Procesamiento del flujo.

Bibliografía



*Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill*

6 Flujo de entrada y salida

Los programas necesitan comunicarse con su entorno, tanto para obtener datos e información que deben procesar, como para almacenar de manera permanente los resultados obtenidos.

El manejo de archivos se realiza a través de streams o flujos de datos desde una fuente (origen) hacia un repositorio (destino).

Los flujos de datos de entrada permiten obtener información a partir de un repositorio de datos donde se encuentran almacenados.

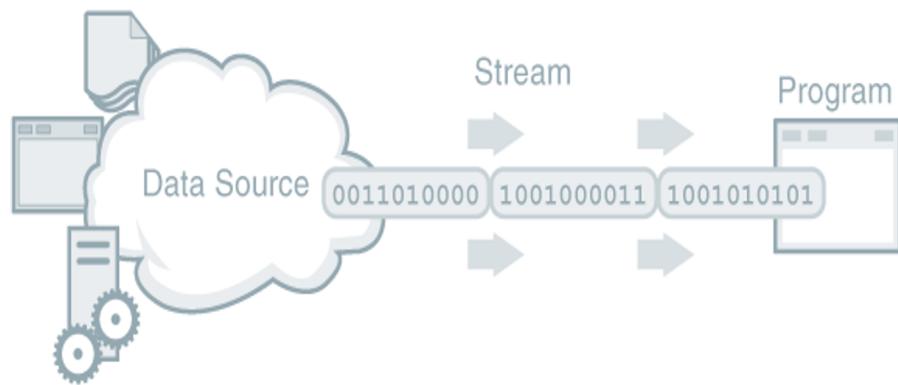
Los flujos de datos de salida permiten almacenar información en un repositorio de datos para su conservación de manera indefinida en un medio no volátil.

6.1 Fundamentos de entrada y salida.

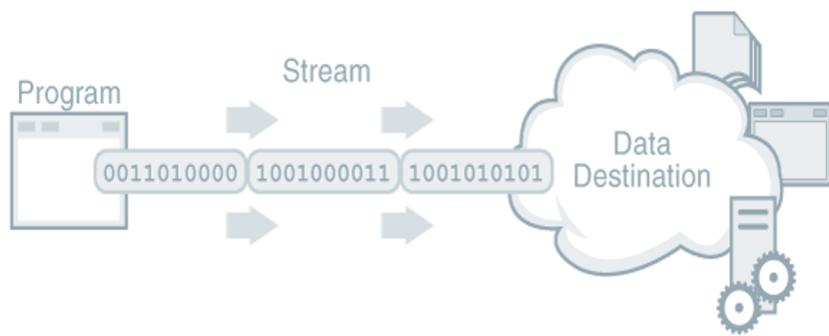
Las entradas y las salidas de datos en Java se manejan mediante streams (o flujos) de datos.

Un stream de datos 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.

Cuando la fuente inicia el flujo de datos, se conoce como flujo de datos de entrada.



Cuando el repositorio termina el flujo de datos, se conoce como flujo de datos de salida.



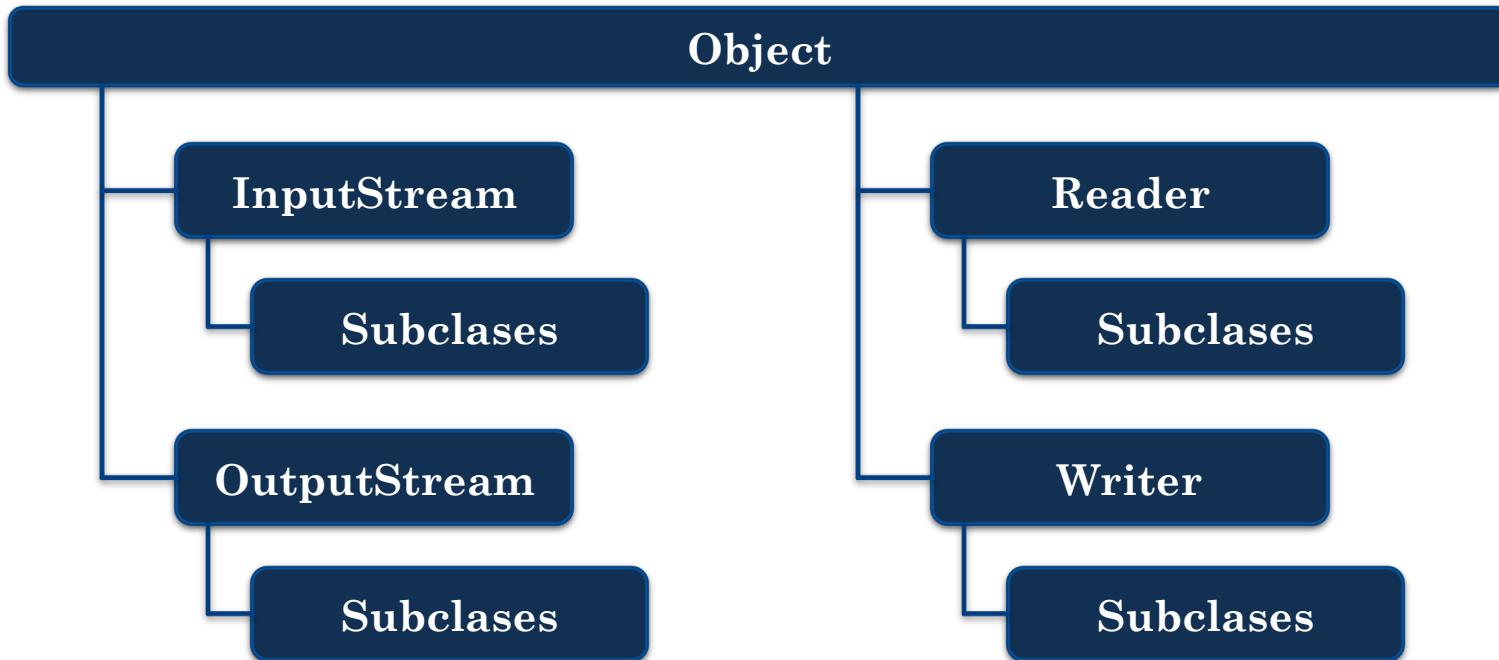
6.2 Jerarquía de clases de los flujos de datos.

En Java existen 4 jerarquías de clases relacionadas con los flujos de entrada y salida de datos.

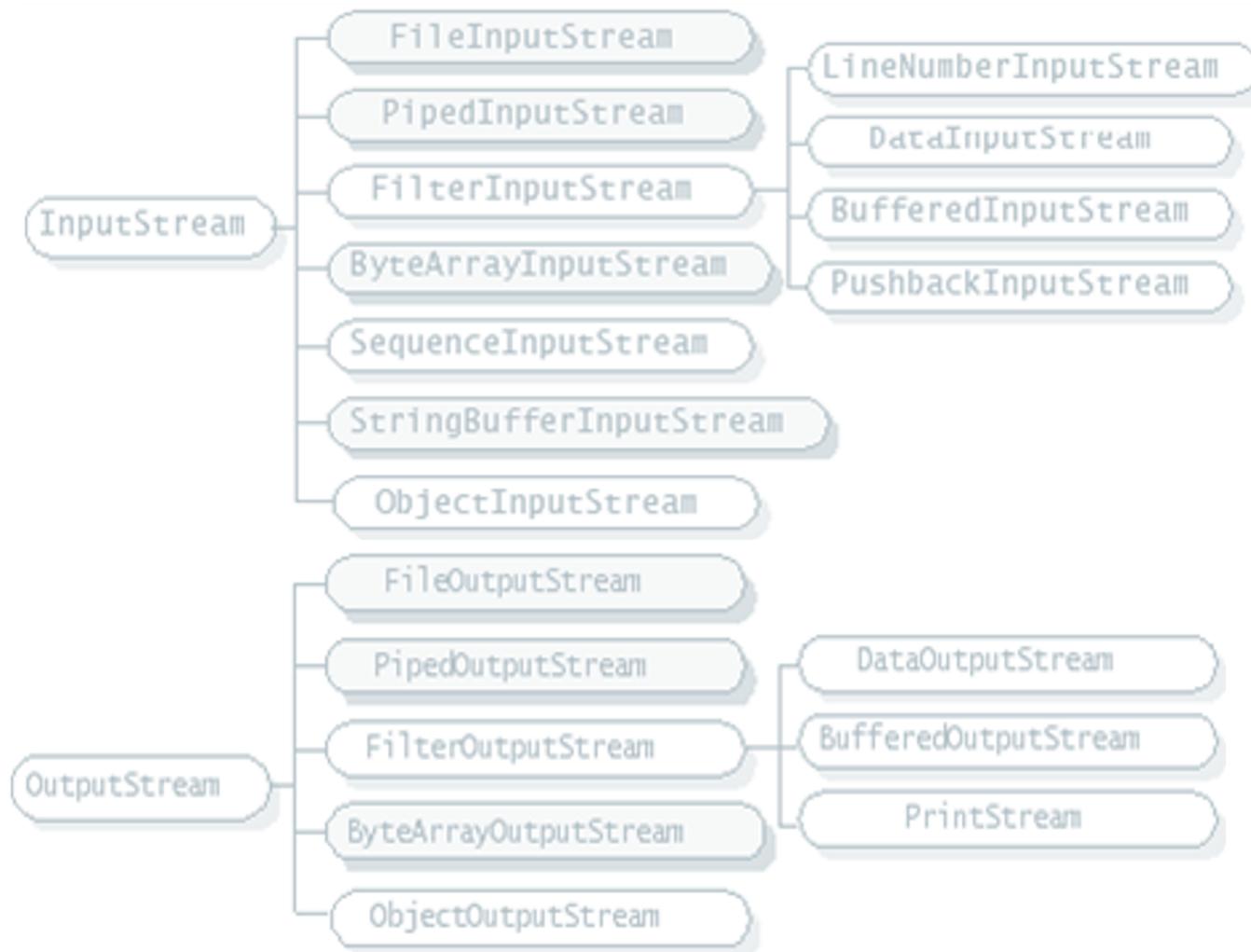
- Flujos de bytes: las clases derivadas de InputStream (para lectura) y de OutputStream (para escritura), las cuales manejan los flujos de datos como stream de bytes.
- Flujos de caracteres: las clases derivadas de Reader (para lectura) y Writer (para escritura), las cuales manejan stream de caracteres.

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

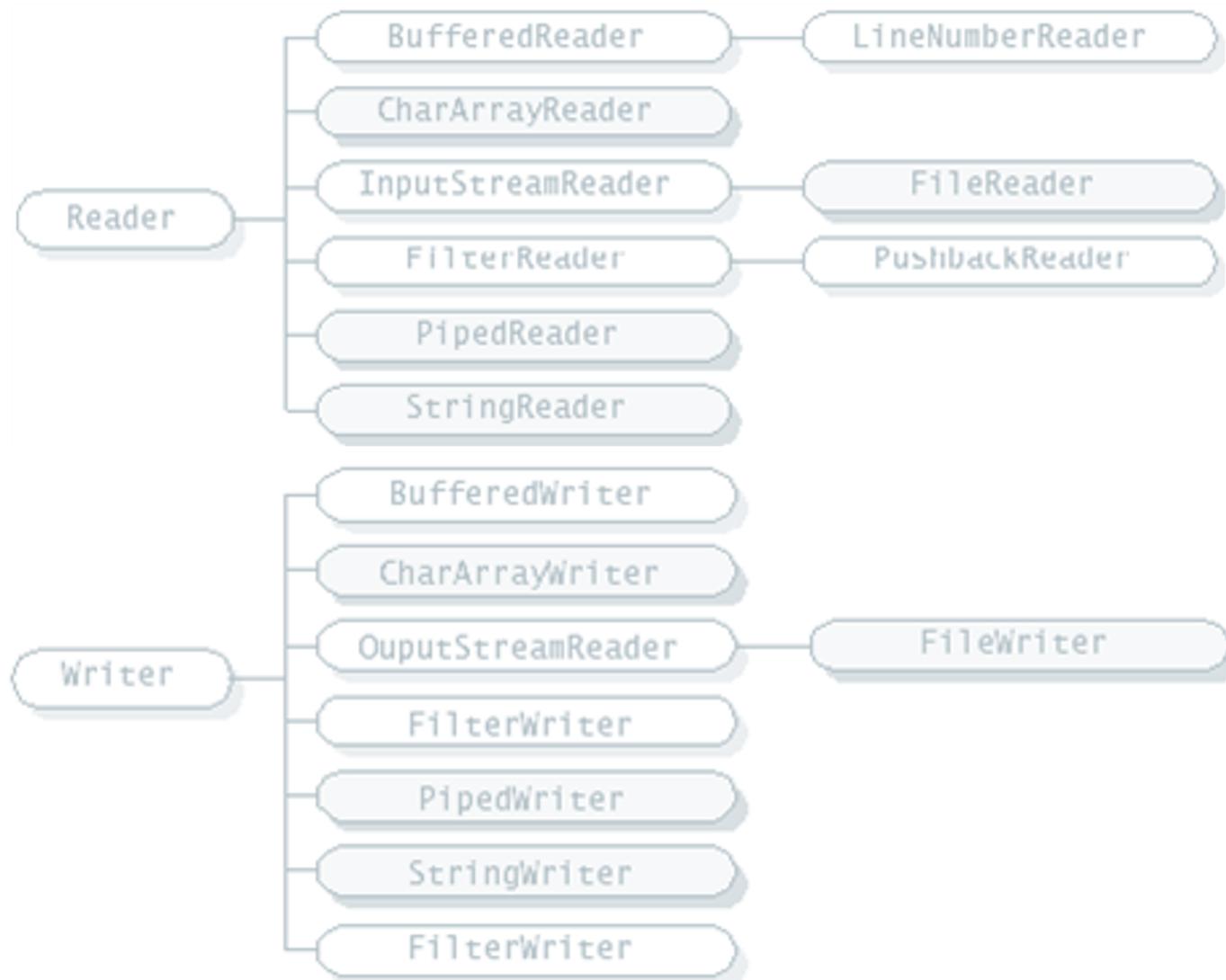
Jerarquía de clases de los flujos de E/S



Jerarquía de clases de Input y Output Stream



Jerarquía de clases de Reader y Writer



6.3 Manipulación de archivos y carpetas.

Archivos

Un archivo es un objeto en una computadora que puede almacenar información, configuraciones o comandos, el cual puede ser manipulado como una entidad por el sistema operativo o por cualquier programa o aplicación.

Un archivo debe tener un nombre único dentro de la carpeta que lo contiene. Normalmente, el nombre de un archivo contiene un sufijo (extensión) que permite identificar el tipo del archivo.

Clase File

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

Cuando se crea una instancia de la clase File no se crea ningún archivo o directorio, solo se crea una referencia hacia un objeto de este tipo.

La creación de archivos o carpetas se realizan de manera explícita, invocando a los métodos respectivos.

Los métodos más útiles que posee la clase File son:

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

Ejemplo 1

```
import java.io.File;
import java.io.IOException;

class CreateFile {
    public static void main(String [] args) {
        try {
            File file = new File("file.txt");
            System.out.println("Exists? " + file.exists());
            boolean wasCreated = file.createNewFile();
            System.out.println("It was created? " + wasCreated);
            System.out.println("Exists? " + file.exists());
        } catch(IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Ejemplo 2

```
import java.io.File;

class MakeDir {
    public static void main(String [] args) {
        try {
            File dir = new File("directory");
            System.out.println("Exists? " + dir.exists());
            boolean wasCreated = dir.mkdir();
            System.out.println("It was created? " + wasCreated);
            System.out.println("Exists? " + dir.exists());
        } catch(SecurityException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Ejemplo 3

```
import java.io.File;

class Rename {
    public static void main(String [] args) {
        try {
            File file = new File("elif.txt");
            File rename = new File("file.txt");
            if (rename.exists()) {
                System.out.println("It was renamed? "
                    + rename.renameTo(file));
            }
            File dir = new File("yrotcerid");
            rename = new File("directory");
            if (rename.exists()) {
                System.out.println("It was renamed? "
                    + rename.renameTo(dir));
            }
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Ejemplo 4

```
import java.io.File;
import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

class ListFiles {
    public static void main(String [] args) {
        try {
            File dir = new File("directory");
            boolean wasCreated = dir.mkdir();
            if (wasCreated){
                File file = new File("directory/file.txt");
                wasCreated = file.createNewFile();
            }
        }
    }
}
```

Ejemplo 4

```
if (wasCreated){
    Date date = new Date();
    DateFormat format =
        new SimpleDateFormat("dd-MM-yyyy");
    File ren = new File("directory/file_"
        + format.format(date)
        + ".txt");
    file.renameTo(ren);
    String [] files = dir.list();
    for (String f : files){
        System.out.println(f);
    }
}
} catch (IOException e){
    System.out.println(e.getMessage());
} catch(SecurityException e) {
    System.out.println(e.getMessage());
}
}
```

6.4 Flujos de entrada de datos.

Para leer flujos de bytes desde un repositorio de datos se utiliza la jerarquía de clases InputStream.

Para leer flujos de caracteres desde una fuente se utiliza la jerarquía de datos de Reader.

6.4.1 Lectura de archivo.

La lectura de datos se realiza desde una fuente (repositorio de datos) hacia un destino. Se pueden leer datos a través de flujos de bytes o a través de flujos de caracteres.

Como se mencionó en la diapositiva anterior, para leer datos en forma de flujos de bytes se utiliza la jerarquía de clases `InputStream`, mientras que para leer datos en forma de flujos de caracteres se utiliza la jerarquía de clases de `Reader`.

FileInputStream

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

- `FileInputStream(String nombre)`
- `FileInputStream(File archivo)`

Ejemplo 5

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

public class FileInputStreamClass {
    public static void main (String [] args){
        FileInputStream fis = null;
        byte[] buffer = new byte[81];
        int nbytes;
        try {
            fis = new FileInputStream("FileInputStreamClass.java");
            nbytes = fis.read(buffer, 0, 81);
            while (nbytes != -1) {
                String text = new String(buffer, 0, nbytes);
                System.out.println(text);
                nbytes = fis.read(buffer, 0, 81);
            }
        }
    }
}
```

Ejemplo 5

```
        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.");
            }
        }
    }
}
```

En el ejemplo anterior se define un flujo que va a leer datos desde el archivo definido (si no existe el archivo se genera una excepción).

También se crea un buffer de 81 bytes. El objeto 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.

FileReader

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

BufferedReader

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

Ejemplo 6

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

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

Ejemplo 6

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

6.4.2 Lectura de teclado.

La lectura de datos desde el teclado se realiza mediante un flujo de bytes. En java, `System.in` es un objeto de la clase `InputStream` que permite recibir datos desde la entrada estándar del sistema (el teclado).

InputStreamReader

La clase `InputStreamReader` permite crear objetos que convierten un flujo de bytes en un flujo de caracteres, es decir, permiten leer datos desde una fuente en forma de bytes y los transforman a caracteres para ser manipulados en forma de `Reader`, de ahí la composición de su nombre de bytes (`Stream`) a caracteres (`Reader`).

Ejemplo 7

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

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

StringTokenizer

La clase StringTokenizer permite separar una cadena de texto por palabras por algún token (separador). El token por defecto es el espacio en blanco aunque el constructor de la clase está sobrecargado para admitir algún otro separador (cadena). La clase StringTokenizer pertenece al paquete java.util.

Ejemplo 8

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

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

Ejemplo 8

```
// Se separa el texto por espacios (defecto)
StringTokenizer st = new StringTokenizer(text);
while(st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
br.readLine();
System.out.println("\n\nEl texto completo es:");
System.out.println(text);
} catch (Exception e){
    System.out.println("\n\nError al leer de teclado:");
    e.printStackTrace();
}
}
```

Scanner

La clase Scanner permite leer datos desde la entrada estándar a través del método nextLine(). Pertenece al paquete java.util.

Scanner también posee los métodos 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.

Scanner usa un delimitador para saber si la cadena tiene más elementos. El delimitador por defecto es el espacio en blanco, aunque es posible cambiarlo utilizando el método `useDelimiter(String cadena)`.

Ejemplo 9

```
import java.util.Scanner;

public class ScannerClass {
    public static void main(String [] args) {
        try {
            String text = "";
            Scanner s = new Scanner(System.in);
            do {
                text = s.nextLine();
                System.out.println(text);
            } while (!text.equals("au revoir"));
            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 10

```
import java.io.Console;

public class ConsoleClass {
    public static void main(String [] args){
        Console con = System.console();
        System.out.print("Usuario: ");
        String user = con.readLine();
        System.out.print("Contraseña: ");
        char [] pass = con.readPassword();

        System.out.println("Los datos ingresados son:");
        System.out.println(user);
        System.out.println(pass);
    }
}
```

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

*java.util.Formatter

Ejemplo 11

```
public class StringFormat {  
    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%n %o%n %f",n,a,a,y);  
        System.out.println(s);  
    }  
}
```

Ejemplo 12

```
public class PrintFormat {  
    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);  
    }  
}
```

6.5 Flujos de salida de datos (escritura de archivo).

Para escribir flujos de bytes en un repositorio de datos se utiliza la jerarquía de clases OutputStream.

Para escribir flujos de caracteres hacia un destino se utiliza la jerarquía de clases Writer.

FileOutputStream

La clase FileOutputStream permite escribir flujos de 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)

Ejemplo 13

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

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

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.

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

Por otra parte, es una buena costumbre cerrar el flujo de datos al final de la ejecución del programa. Debido a que la escritura de datos puede fallar, el bloque finally es el mejor lugar para cerrar la conexión.

FileWriter

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

BufferedWriter

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

PrintWriter

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

Ejemplo 14

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

Ejemplo 14

```
        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();
        } finally {
            try {
                if (salida != null)
                    salida.close();
            } catch (Exception e){
                System.out.println("Error : " + e.toString());
            }
        }
    }
```

Tarea

Log in

Crear un programa en java que permita registrar usuarios (sin repetir nombre de usuario) o iniciar sesión si un usuario ya está registrado en el sistema. Los usuarios y sus contraseñas se almacenan en un archivo de texto.

6.6 Procesamiento del flujo.

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) a través de la red.

Los sockets se clasifican en sockets de flujo o sockets de datagramas dependiendo del servicio que utilizan: 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.

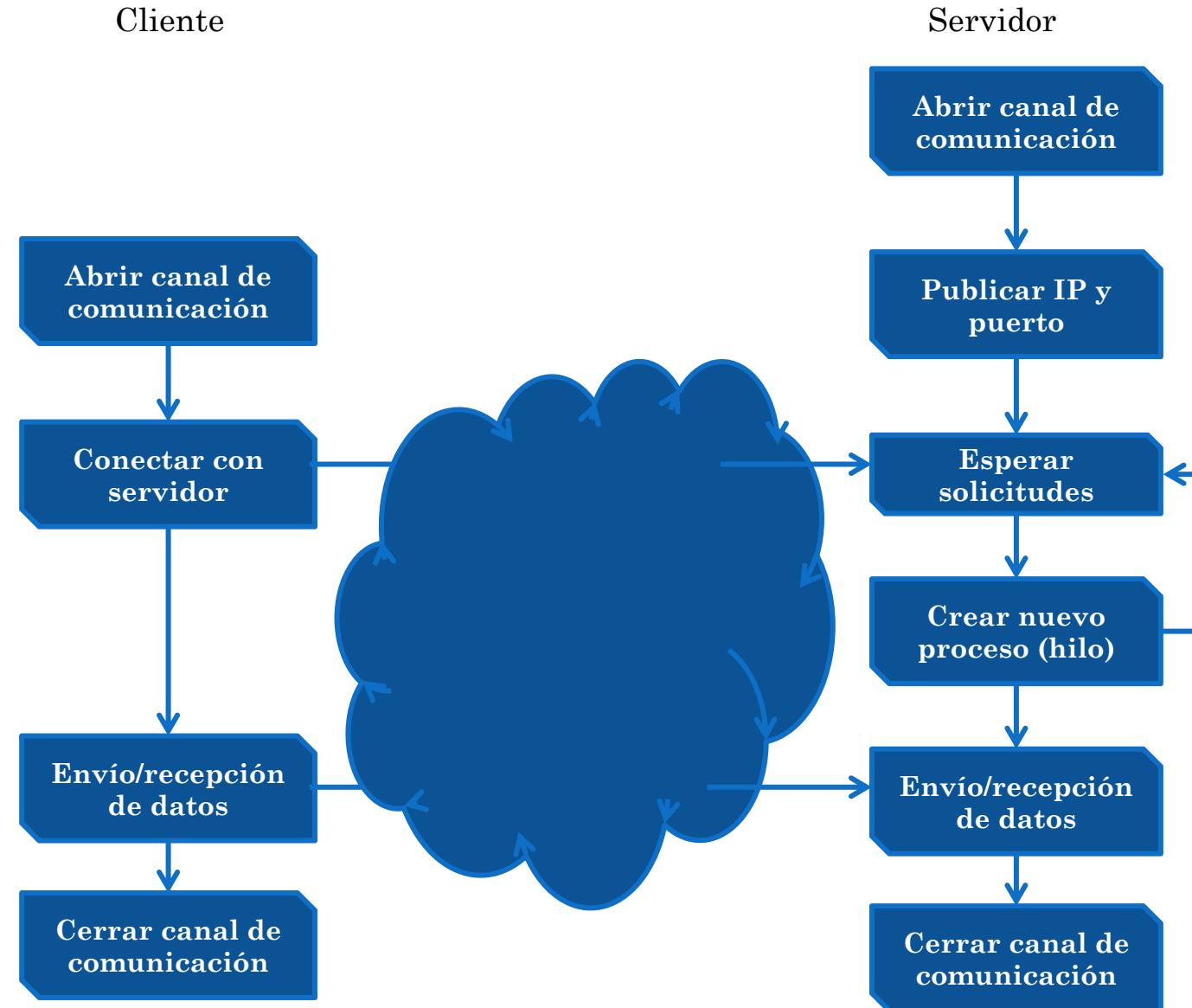
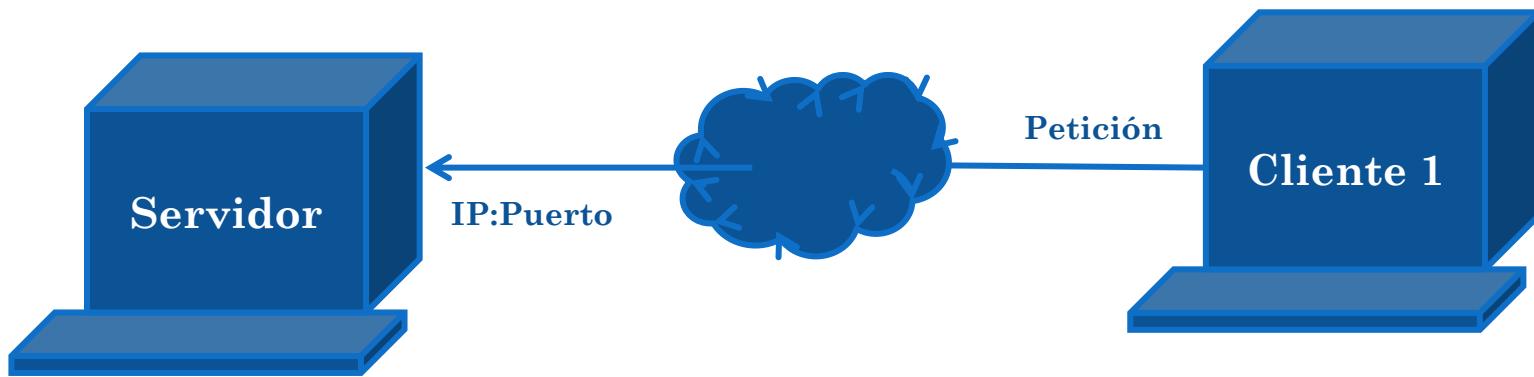


Diagrama de dialogo entre sockets (Cliente-Servidor)

Un servicio se ejecuta en una computadora sobre un puerto específico, a la espera de peticiones. Este programa (servicio) se conoce como 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

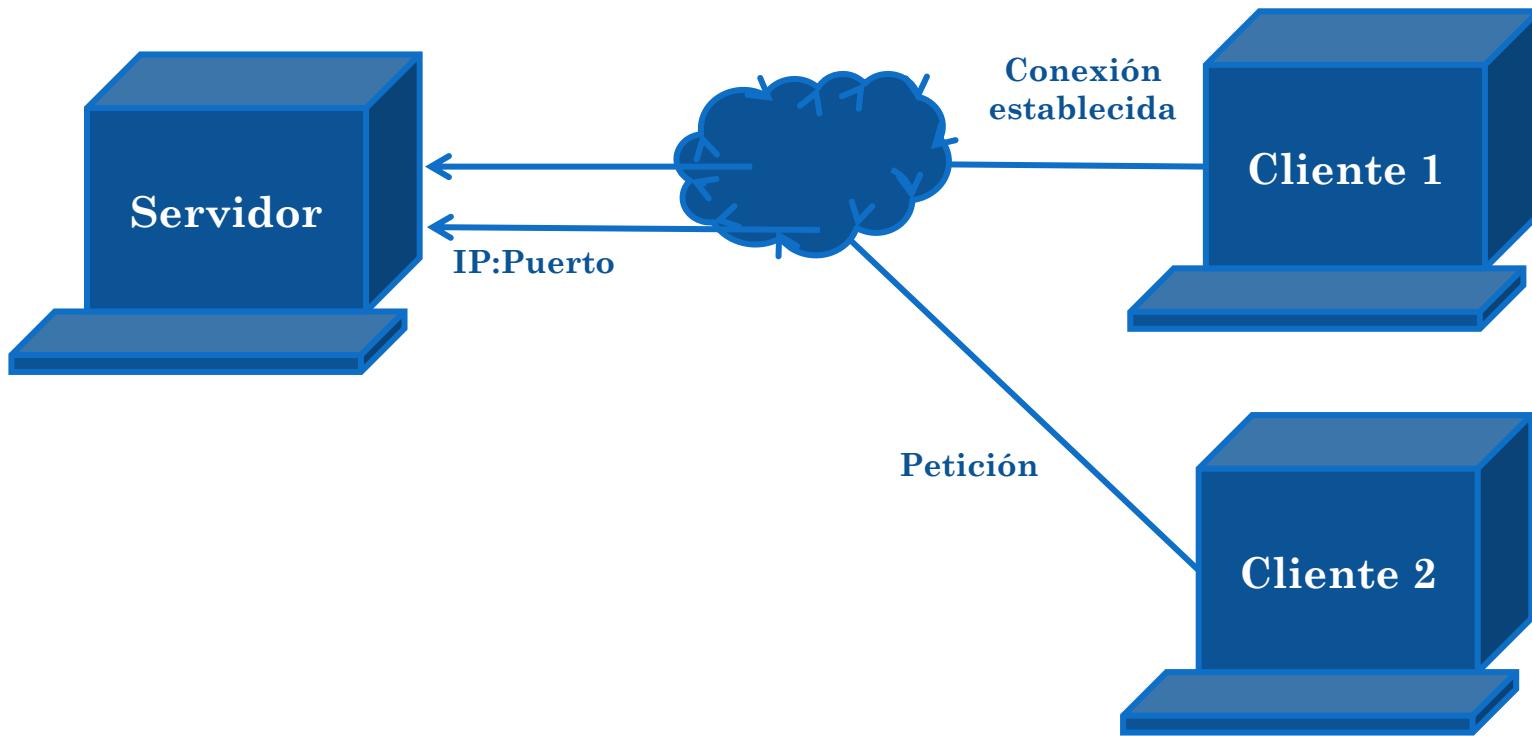


6 Flujo de entrada y salida.

En el servidor, cuando se acepta una conexión 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 crea un socket que intenta establecer una conexión hacia el servidor por el que ambos pueden seguir comunicados (flujo de datos). El socket en el cliente también se ejecuta 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 y otro. Estos sockets pueden comunicarse a través de la red de forma totalmente independiente de la plataforma donde se ejecute.

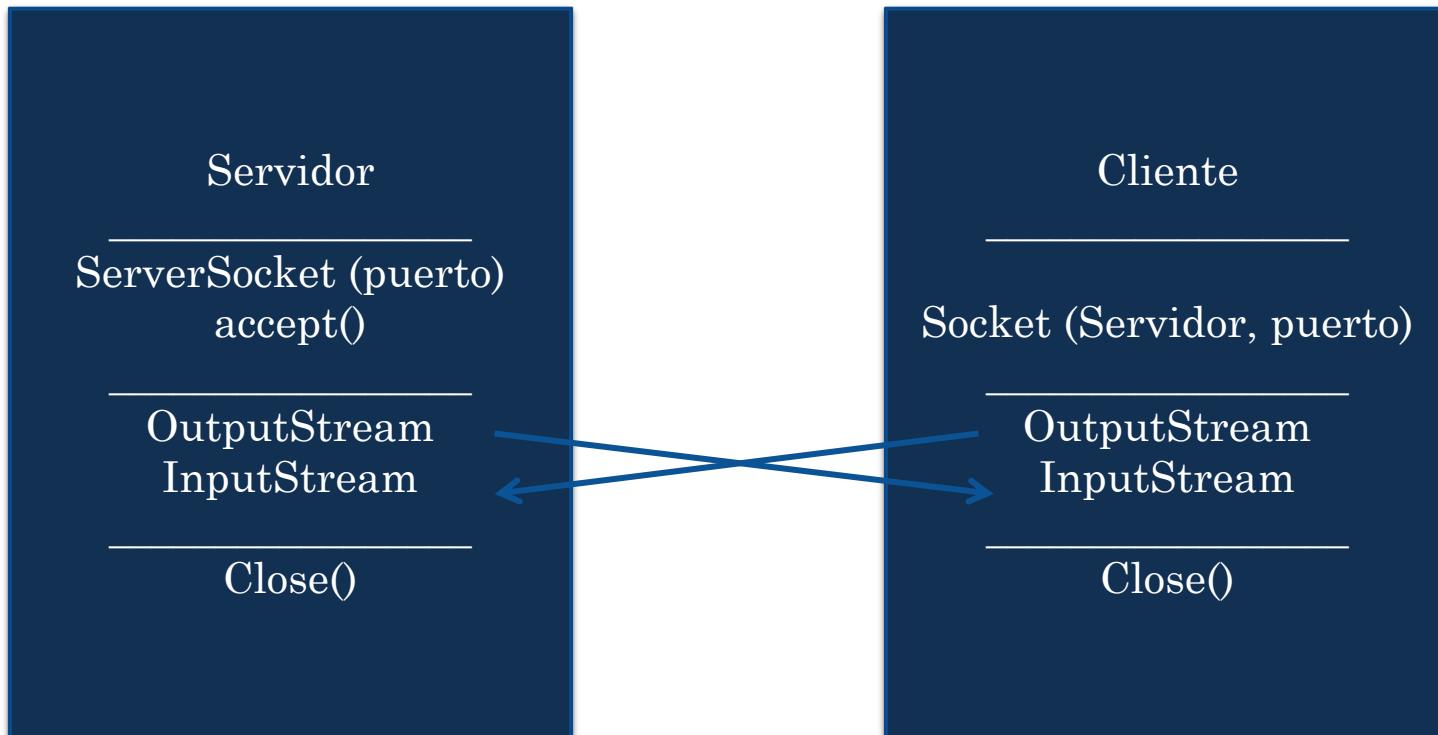
Además, java.net también posee la clase ServerSocket, que implementa un servidor para estar a al escucha de peticiones de conexiones por parte de los clientes.

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

El servidor espera la petición del cliente por un puerto específico. Cuando el cliente realice la petición, el servidor abrirá una conexión (flujo de datos).

El cliente realiza la petición al servidor por el puerto establecido y, de ser aceptada, se abre una conexión por la que ambos pueden intercambiar información mediante flujos de datos (InputStream y OutputStream).

Modelo de comunicación básico entre sockets



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.

- 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());  
}
```

La cadena Servidor se refiere al nombre (o ip) del equipo que va a estar a la espera de conexiones y el puerto por donde estará a la escucha.

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());  
}
```

Cada vez que el servidor acepta una petición, crea un socket nuevo para seguir a la escucha de solicitudes.

Flujos de entrada

DataStream permite crear flujos de entrada para recibir los datos que envíe un nodo.

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

Flujos de salida

Las clases PrintStream y DataOutputStream permiten crear flujos de salida para enviar información.

Ambas clases permiten escribir cualquier tipo de dato primitivo en forma de flujos de bytes.

Al final del programa es necesario cerrar todos los flujos de datos (tanto de entrada como de salida), así como los sockets creados.

Primero se deben cerrar los flujos de datos para que toda la información llegue a los nodos y, al final, se debe cerrar el canal de comunicación (el socket o los sockets).

Ejemplo 14

```
import java.io.InputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.net.Socket;

public class Client {
    static final String SERVER = "localhost";
    static final int PORT = 5432;
    public Client() {
        Socket con = null;
        DataInputStream dis = null;
        DataOutputStream dos = null;
        try{
            con = new Socket(SERVER, PORT);
            InputStream read = con.getInputStream();
            dis = new DataInputStream(read);
            System.out.println(dis.readUTF());
            dos = new DataOutputStream(con.getOutputStream());
            dos.writeUTF("Gracias por aceptarme.");
        }
    }
}
```

Ejemplo 14

```
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (dis != null) dis.close();
                if (dos != null) dos.close();
                if (con != null) con.close();
            } catch (Exception e){
                System.out.println(e.getMessage());
            }
        }
    }

public static void main(String [] arg) {
    new Client();
}
```

Ejemplo 14

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

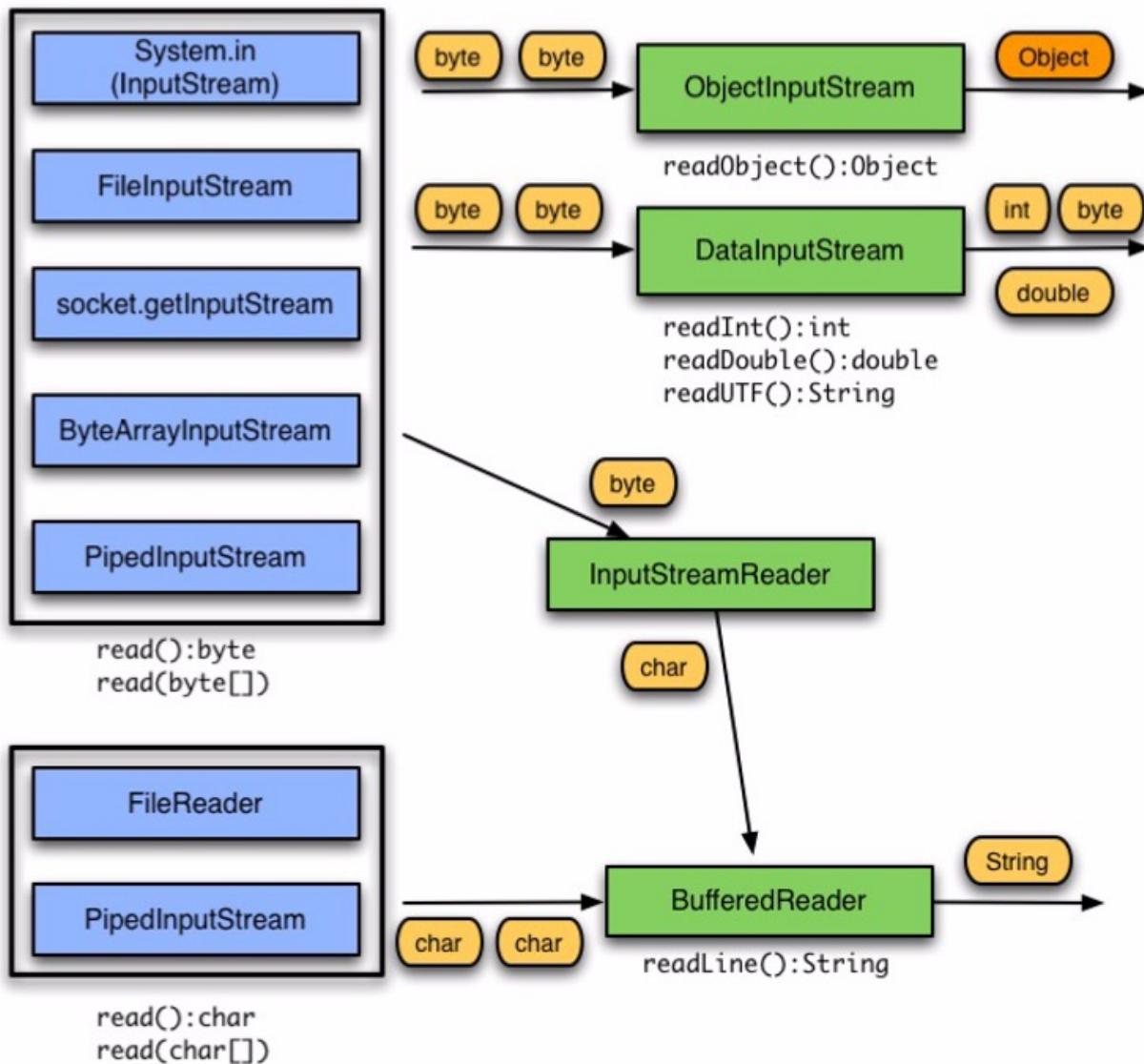
public class Server {
    static final int PORT = 5432;

    public Server() {
        Socket service = null;
        DataOutputStream fds = null;
        DataInputStream dis = null;
        try {
            ServerSocket server = new ServerSocket(PORT);
            System.out.println("Escuchando por el puerto " + PORT);
            for (int clients = 0 ; clients < 5; clients++) {
                service = server.accept();
                System.out.println("Conexión del cliente " + clients);
                OutputStream writting = service.getOutputStream();
                fds = new DataOutputStream(writting);
                fds.writeUTF("Bienvenido cliente " + clients);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

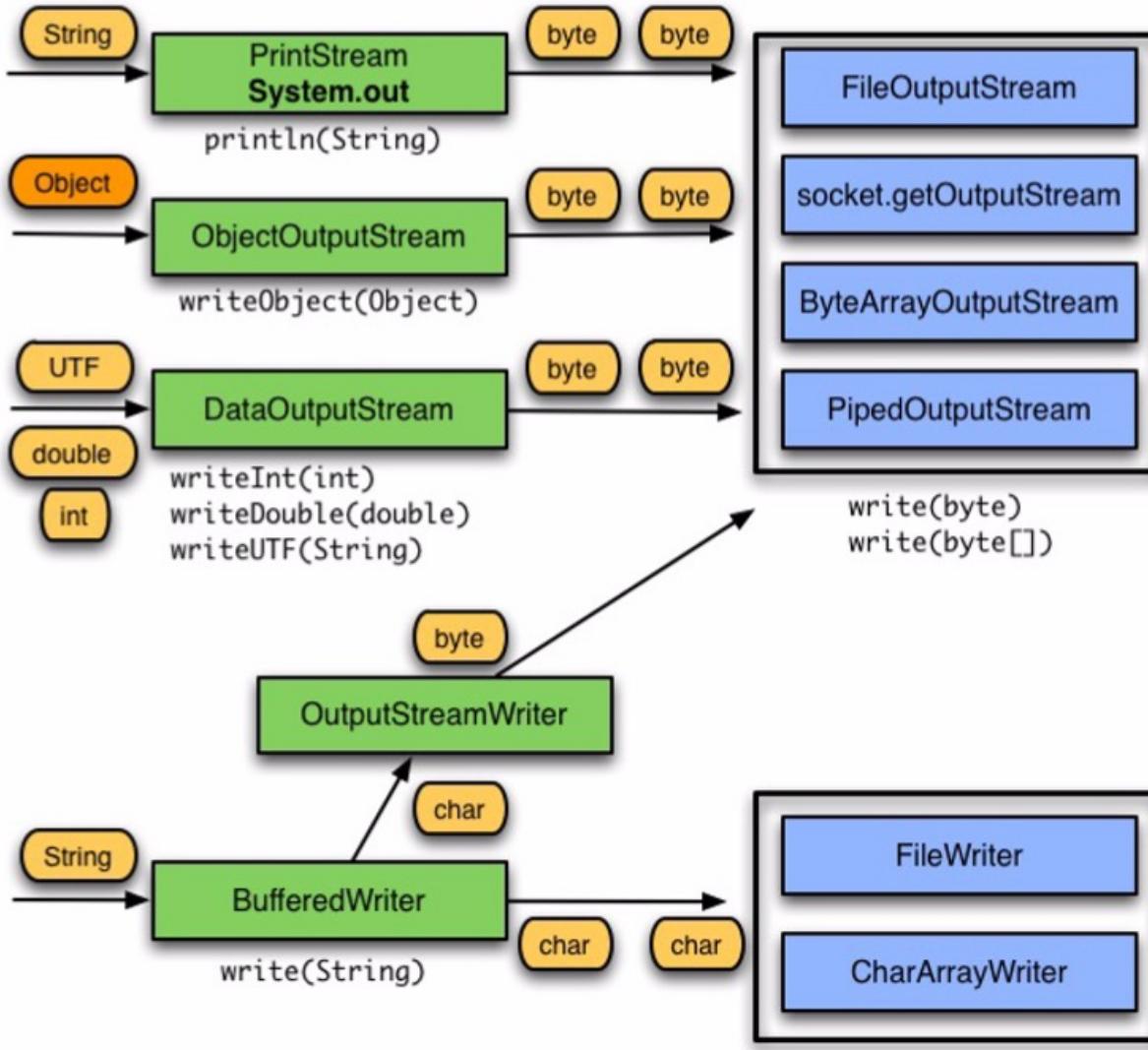
Ejemplo 14

```
        dis = new DataInputStream(service.getInputStream());
        System.out.println("Cliente " + clients + " dice: " + dis.readUTF());
    }
    System.out.println("Demasiados clientes por hoy.");
} catch(Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
} finally {
    try {
        if (fds != null) fds.close();
        if (dis != null) dis.close();
        if (service != null) service.close();
    } catch (Exception e){
        System.out.println(e.getMessage());
    }
}
}

public static void main(String [] arg) {
    new Server();
}
```



Combinaciones de flujos de entrada de datos



Combinaciones de flujos de salida de datos

DAO

Data Access Object (DAO) es un patrón de persistencia de datos, es decir, permite acceder o almacenar información en algún medio de almacenamiento permanente (repositorio de datos).

Los objetos de acceso a datos (DAO) permiten abstraer y encapsular todos los accesos a una fuente de datos.

DAO esconde por completo los detalles de implementación de la fuente de datos de los clientes. Este patrón permite adaptarse a diferentes esquemas de almacenamiento sin afectar a los clientes o los componentes que lo utilizan.

En esencia, un DAO actúa como un adaptador (intermediario) entre el componente de software y la fuente de datos.

Los participantes dentro de este patrón son:

- El objeto de negocio (Business Object).
- El objeto de acceso a datos (Data Access Object).
- La fuente de datos (Data Source).
- El objeto de transferencia (Transfer Object).

Business Object

Representa la información (datos) del cliente. Este objeto requiere acceso a la fuente de datos para leer o almacenar la información que posee.

Data Access Object

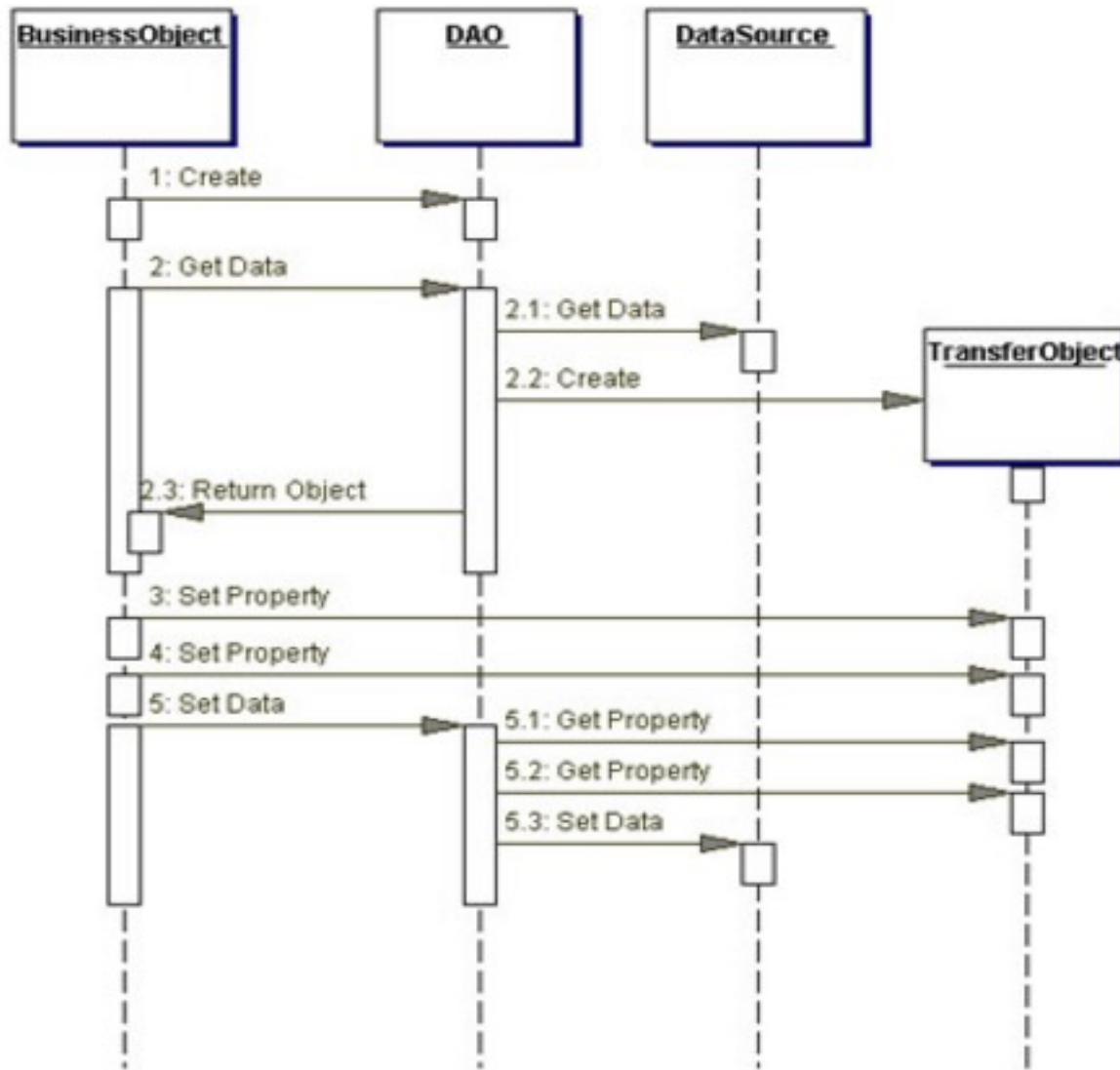
Representa al objeto primario del patrón. AbstRAE la implementación de acceso a datos básica para permitir que el objeto de negocio acceda de manera transparente a la fuente de datos.

Data Source

Representa la implementación de la fuente de datos, la cual puede ser una base de datos, un archivo, una conexión remota, etc.

Transfer Object

Representa la transferencia de datos. Se utiliza como soporte de datos. Los DAO utilizan los objetos de transferencia para regresar información al cliente o enviar información a la fuente de datos. Es el objeto donde se transporta la información.



Patrón DAO

```
public class User {  
    private String userName;  
    private String password;  
  
    public User(){  
  
        public User(String userName, String password){  
            setUserName(userName);  
            setPassword(password);  
        }  
  
        public void setUserName(String userName){  
            this.userName = userName.trim();  
        }  
        public String getUserName(){  
            return this.userName;  
        }  
        public void setPassword(String password){  
            this.password = password.trim();  
        }  
        public String getPassword(){  
            return this.password;  
        }  
        public String toString(){  
            return this.userName + ":*****";  
        }  
    }  
}
```

Transfer Object

Data Source



uno | hola
tres | 12345
cinco | qwerty

```
import java.util.List;

public interface DAO {
    public List<User> getAllUsers();
    public User findUser(String userName);
}
```

```
import java.util.StringTokenizer;

public class UserDAO implements DAO{
    private String fileName;
    private List<User> users;

    public UserDAO() {
    }

    @Override
    public List<User> getAllUsers() {
        return this.users;
    }

    @Override
    public User findUser(String userName) {
        return users.
            stream().
            filter(t -> t.getUserName().equals(userName)).
            findFirst().get();
    }
}
```

```
import java.util.Iterator;
import java.util.List;

public class TestUser {
    public static void main (String [] users){
        UserDAO dao = new UserDAO();
        System.out.println("Find user uno: " + dao.findUser("uno"));
        List<User> list = dao.getAllUsers();
        Iterator i = list.iterator();
        while (i.hasNext()){
            System.out.println(i.next());
        }
    }
}
```

Ejercicio

Dadas las clases anteriores, generar el código necesario para cargar los usuarios de la lista de un archivo de texto plano.

El archivo de usuarios debe tener por renglón el nombre de usuario, un token de separación (pipe) y la contraseña. El nombre del archivo de usuarios debe estar almacenado en otro archivo de configuración.

```
public UserDAO0 {  
    this.users = new ArrayList<User>();  
    try {  
        BufferedReader br;  
        FileReader fr;  
        fr = new FileReader("info.txt");  
        br = new BufferedReader(fr);  
        fileName = br.readLine();  
        br = null;  
        fr.close();  
        fr = new FileReader(fileName);  
        br = new BufferedReader(fr);  
        String line = br.readLine();  
        while (line != null) {  
            StringTokenizer st = new StringTokenizer(line, " | ");  
            String usr = st.nextToken();  
            String pwd = st.nextToken();  
            User tmpUser = new User(usr.trim(), pwd.trim());  
            boolean add = users.add(tmpUser);  
            line = br.readLine();  
        }  
    } catch (IOException ioe){  
        System.out.println("\n\nError al abrir o leer el archivo:");  
        ioe.printStackTrace();  
    } catch (Exception e){  
        System.out.println("\n\nError:");  
        e.printStackTrace();  
    }  
}
```

Serialización

La serialización es un mecanismo para guardar los objetos como una secuencia de bytes y poderlos reconstruir en un futuro cuando se necesiten, conservando su estado.

Cuando un objeto se serializa solo los campos del objeto (atributos) son preservados, para conservar su estado intacto y poder utilizarlo en otro tiempo. La serialización guarda el objeto en un medio de almacenamiento secundario (archivo) para su posterior recuperación.

Para que un objeto sea serializable, la clase debe implementar la interfaz Serializable, la cual se encuentra en el paquete java.io.

Ejemplo 15

```
import java.io.Serializable;

public class Student implements Serializable {
    private String name;
    private int id;

    public Student(){
        name = "Ninguno";
        id = -10000;
    }

    public Student(String name, int id){
        this.name = name;
        this.id = id;
    }

    public String toString(){
        return name + " : " + id;
    }
}
```

Ejemplo 15

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

public class SerializeStudent {

    SerializeStudent() {
        Student student = new Student("Joe", 12345);
        System.out.println(student);
        try {
            FileOutputStream file = new FileOutputStream ("student.ser");
            ObjectOutputStream stream = new ObjectOutputStream (file);
            stream.writeObject (student);
            stream.close();
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }

    public static void main (String args[]) {
        new SerializeStudent();
    }
}
```

La deserialización es un mecanismo que permite recuperar objetos serializados a partir de un archivo de texto plano.

Un objeto se puede serializar en partes (por atributos) o de manera íntegra. La deserialización se debe realizar de la misma manera en la que se serializó el objeto.

Ejemplo 16

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class DeSerializeStudent {

    DeSerializeStudent () {
        Student student = null;

        try {
            FileInputStream file = new FileInputStream ("student.ser");
            ObjectInputStream stream = new ObjectInputStream(file);
            student = (Student) stream.readObject();
            stream.close();
        } catch (Exception e) {
            e.printStackTrace ();
        }
        System.out.println("Deserializar objeto de student.ser");
        System.out.println(student);
    }

    public static void main (String args[]) {
        new DeSerializeStudent();
    }
}
```

Si un atributo del objeto que se desea serializar hace referencia a otro objeto, este objeto debe ser también serializable.

Los objetos que no son serializables se deben declarar con la palabra reservada transient para evitar que se intenten serializar y se genere un error.

Ejemplo 17

```
import java.io.Serializable;

public class Student implements Serializable {
    private Schedule schedule = new Schedule();
    private String name;
    private int id;

    public Student(){
        name = "Ninguno";
        id = -10000;
    }

    public Student(String name, int id){
        this.name = name;
        this.id = id;
    }

    public String toString(){
        return name + " : " + id;
    }
}

public class Schedule {
```

Supóngase que en un instante de tiempo se serializa un objeto x de tipo Equis, posteriormente la abstracción del objeto cambia y, por ende, la clase Equis. ¿Qué pasa si ahora se quiere deserializar el objeto con esta nueva clase?

En realidad no se debería deserializar puesto que el objeto fue serializado con otros parámetros.

En Java cuando se serializa un objeto se asocia a cada clase serializable un número de versión, el cual permite identificar con qué versión de la clase se serializa una instancia:

```
[access-modifier] static final long serialVersionUID = 42L;
```

Cuando se intenta deserializar un objeto se verifica que la versión a la que se quiere reconstruir el objeto corresponda con la versión con la que fue serializada, para que los atributos guardados correspondan.

Si no se especifica un número de versión de manera explícita, java calcula lo genera para poder deserializar en el futuro.

Ejemplo 18

```
import java.io.Serializable;

public class Student implements Serializable {
    private static final long serialVersionUID = 123456789L;
    private String name;
    private int id;

    public Student(){
        name = "Ninguno";
        id = -10000;
    }

    public Student(String name, int id){
        this.name = name;
        this.id = id;
    }

    public String toString(){
        return name + " : " + id;
    }
}
```

Serialización en herencia

Cuando una clase base define que es serializable, toda la jerarquía de clases se vuelve serializable (hereda el comportamiento).

Por otro lado, si una clase derivada es serializable pero la clase base no lo es, el objeto se puede serializar pero los atributos que hereda no serán serializados y, por lo tanto, cuando se recupere el objeto, los atributos tendrán sus valores por defecto.

Ejemplo 19

```
import java.io.Serializable;

public class Person implements Serializable{
    private static final long serialVersionUID = 123456789L;
    protected char genero;
    protected int edad;

    public Person(){}
    public Person(char genero, int edad){
        this.genero = genero;
        if (edad >= 0)
            this.edad = edad;
    }
    public String toString(){
        return genero + " : " + edad + " : ";
    }
}
```

Ejemplo 19

```
public class Student extends Person {  
    private static final long serialVersionUID = 123L;  
    private String name;  
    private int id;  
  
    public Student(){  
        name = "Ninguno";  
        id = -10000;  
    }  
    public Student(String name, int id){  
        this.name = name;  
        this.id = id;  
    }  
    public Student(String name, int id, char genero, int edad){  
        super(genero, edad);  
        this.name = name;  
        this.id = id;  
    }  
  
    public String toString(){  
        return super.toString() + name + ":" + id;  
    }  
}
```

Ejemplo 19

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

public class SerializeStudent {

    SerializeStudent() {
        Student student = new Student("Joe", 12345, 'M', 18);
        System.out.println(student);
        try {
            FileOutputStream file = new FileOutputStream ("student.ser");
            ObjectOutputStream stream = new ObjectOutputStream (file);
            stream.writeObject (student);
            stream.close();
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }

    public static void main (String args[]) {
        new SerializeStudent();
    }
}
```

Ejemplo 19

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class DeSerializeStudent {

    DeSerializeStudent () {
        Student student = null;

        try {
            FileInputStream file = new FileInputStream ("student.ser");
            ObjectInputStream stream = new ObjectInputStream(file);
            student = (Student) stream.readObject();
            stream.close();
        } catch (Exception e) {
            e.printStackTrace ();
        }
        System.out.println("Deserializar objeto de student.ser");
        System.out.println(student);
    }

    public static void main (String args[]) {
        new DeSerializeStudent();
    }
}
```

6 Flujo de entrada y salida

Objetivo: Construir programas con el principio de flujo de entrada y salida para procesar información a partir de un problema resuelto.

- 6.1 Fundamentos de entrada y salida.
- 6.2 Jerarquía de clases de los flujos de datos.
- 6.3 Manipulación de archivos y carpetas.
- 6.4 Flujos de entrada de datos.
 - 6.4.1 Lectura de archivo.
 - 6.4.2 Lectura de teclado.
- 6.5 Flujos de salida de datos (escritura de archivo).
- 6.6 Procesamiento del flujo.