Java: E/S (io), hilos, utilidades, excepciones y sockets

28.2.2024

Enrique Krause Buedo

1 Entrada y salida a ficheros

28.2.2024

Enrique Krause Buedo

Introducción a la escritura y lectura de ficheros

La vocación del lenguaje Java de ser independiente del sistema operativo supone una dificultad añadida en los procesos de lectura y escritura de ficheros, sobre todo en aplicaciones como los applets para los navegadores de Internet en los que el acceso a los ficheros locales está controlado por un complejo sistema de seguridad.

La escritura y lectura en ficheros se basa en flujos de datos, sobre estos flujos se puede leer y escribir datos, los cuales son el canal de comunicación entre el programa en la memoria del ordenador y el fichero en el soporte (disco) de almacenamiento. Los flujos se pueden clasificar en flujos de bytes (clases InputStream y Outputstream) y flujos de caracteres (clases Reader y Writer). Existen otras clases que sin conectar directamente con un origen de datos, por ejemplo un fichero, permiten transformar los flujos de datos creados por las clases generadores de flujos. A estas clases se les denomina filtros.

Introducción a la escritura y lectura de ficheros

El método general de lectura en ficheros consiste en enlazar un flujo y un filtro, creando un canal completo de transferencia de datos.

Mención especial en el acceso a ficheros se merece la clase File de utilidades de identificación de ficheros y directorios. Los objetos de la clase File deben conocer la naturaleza del sistema de ficheros sobre la cual se ejecuta la aplicación. El objeto de la clase File que podremos llamar fuente identifica el fichero. A esta fuente se le asocia un flujo, sobre el cual recae la responsabilidad de enviar los datos, generalmente en sucesiones de bytes no diferenciados, por lo que será necesario aplicar un filtro, para interpretar estos datos como variables primitivas, cadenas de caracteres y estructuras de objetos.

.

La gestión de ficheros y directorios; la clase File

La clase File tiene los siguientes constructores:

```
File(String directorio);
File(String fichero);
File(String directorio, String fichero);
File(File directorio, String fichero);
```

El Objeto File es un directorio puede utilizar **Métodos de la clase File p/directorios.**

boolean isDirectory() Investiga si es un directorio

boolean mkdir() Crea un directorio

boolean mkdirs() Crea los directorios necesarios de un path

boolean exist() Investiga si existe un directorio

boolean delete() Borra el directorio

String [ ] list() Retorna la lista de ficheros que hay en el directorio

String [ ] listRoots() Lista de volúmenes de almacenamiento (discos)

Introducción a la escritura y lectura de ficheros

El programa ListaFicheros muestra cómo obtener la lista de ficheros de un directorio.

ListaFicheros.java

```
import java.io.*;

public class ListaFicheros {

public static void main(String[] args) {

File canal =new File("F:\\00música\\11lecciones joe pass");

File [ ] lista = canal.listFiles();

for(int i = 0; i < lista.length;i++) {

System.out.println(lista[i]);

}

}

}
```

```
F:\00música\11lecciones joe pass\(Guitar Lesson) Joe Pass - The Blue Side Of Jazz (Hot Licks).avi
F:\00música\11lecciones joe pass\Guitar Lesson - Joe Pass - An Evening With [Reh].avi
F:\00música\11lecciones joe pass\Guitar Lesson - Joe Pass - Jazz Lines [REH].mpg
F:\00música\11lecciones joe pass\Guitar Lesson - Joe Pass - SOLO JAZZ GUITAR (REH Instruction Video).mpg
F:\00música\11lecciones joe pass\Joe Pass - SOLO JAZZ GUITAR -Guitar Lesson - (REH Instruction Video).mpg
F:\00música\11lecciones joe pass\Joe Pass - Solo Jazz Guitar.avi
```

Introducción a la escritura y lectura de ficheros

Si el objeto File es un fichero puede utilizar los métodos que se recogen: la tabla 10.2

Métodos de la clase File para ficheros.

- **boolean isFile()** Investiga si es un fichero.
- **boolean exists()** Investiga si existe el fichero. boolean canRead() Investiga si se puede leer el fichero.
- **boolean canWrite()** Investiga si se puede escribir sobre el fichero, boolean delete() Borra el fichero.
- **long length()** Investiga el tamaño del fichero en bytes.
- **long lastModified()** Investiga la fecha de la ultima modificación. boolean renameTo() Cambia el nombre del fichero.

El programa **AtributosFichero** muestra los atributos de un directorio: especificado por el usuario. El Programa solicita el nombre de un fichero Con este nombre crea un objeto de la clase File. Sobre este objeto se ejcutan los métodos getName y getPath para investigar el nombre y el directorio en el que está el fichero. Los métodos canRead y canWrite investigan si se puede leer y modificar. El método length investiga el tamaño del fichero en bytes.

Introducción a la escritura y lectura de ficheros

```
import java.io.*;
public class AtributosFichero {
public static void main(String[] args) {
String nombreF;
InputStreamReader flujo = new  InputStreamReader(System.in);
BufferedReader teclado = new BufferedReader (flujo) ;
try{
System.out.println("Introduce el nombre del fichero");
    nombreF = teclado.readLine ();
    File f = new File(nombreF);
    System.out.println( "Nombre: "+f.getName() );
    System.out.println( "Camino: "+f.getPath() );
if (f.exists()) System.out .println( "El fichero existe" );
if (f.canRead()) System.out.println("Se puede leer");
if(f.canWrite()) {
System.out.println( "Se puede modificar");
System.out.println("La longitud del fichero es:" + f.length()+" bytes" );
}else System.out.println( "El fichero no existe." );
}catch (Exception e) {
System.out.println(e.getMessage());
}    } }
```

```
Introduce el nombre del fichero
F:\00música\11lecciones joe pass\
(Guitar Lesson) Joe Pass - The Blue
Side Of Jazz (Hot Licks).avi
Nombre: (Guitar Lesson) Joe Pass -
The Blue Side Of Jazz (Hot Licks).avi
Camino: F:\00música\11lecciones joe
pass\(Guitar Lesson) Joe Pass - The
Blue Side Of Jazz (Hot Licks).avi
El fichero existe
Se puede leer
Se puede modificar
La longitud del fichero es:732449408
bytes
```

Las clases Reader y Writer

Las clases Reader y Writer son clases abstractas de las que se derivan las clases que permiten crear flujos de caracteres de 16 bits (Unicode) de lectura y escritura en ficheros. La lista muestra las clases de uso más frecuente par la lectura y escritura de ficheros de texto derivadas de las clases Reader y Writer.196 Capítulo 10

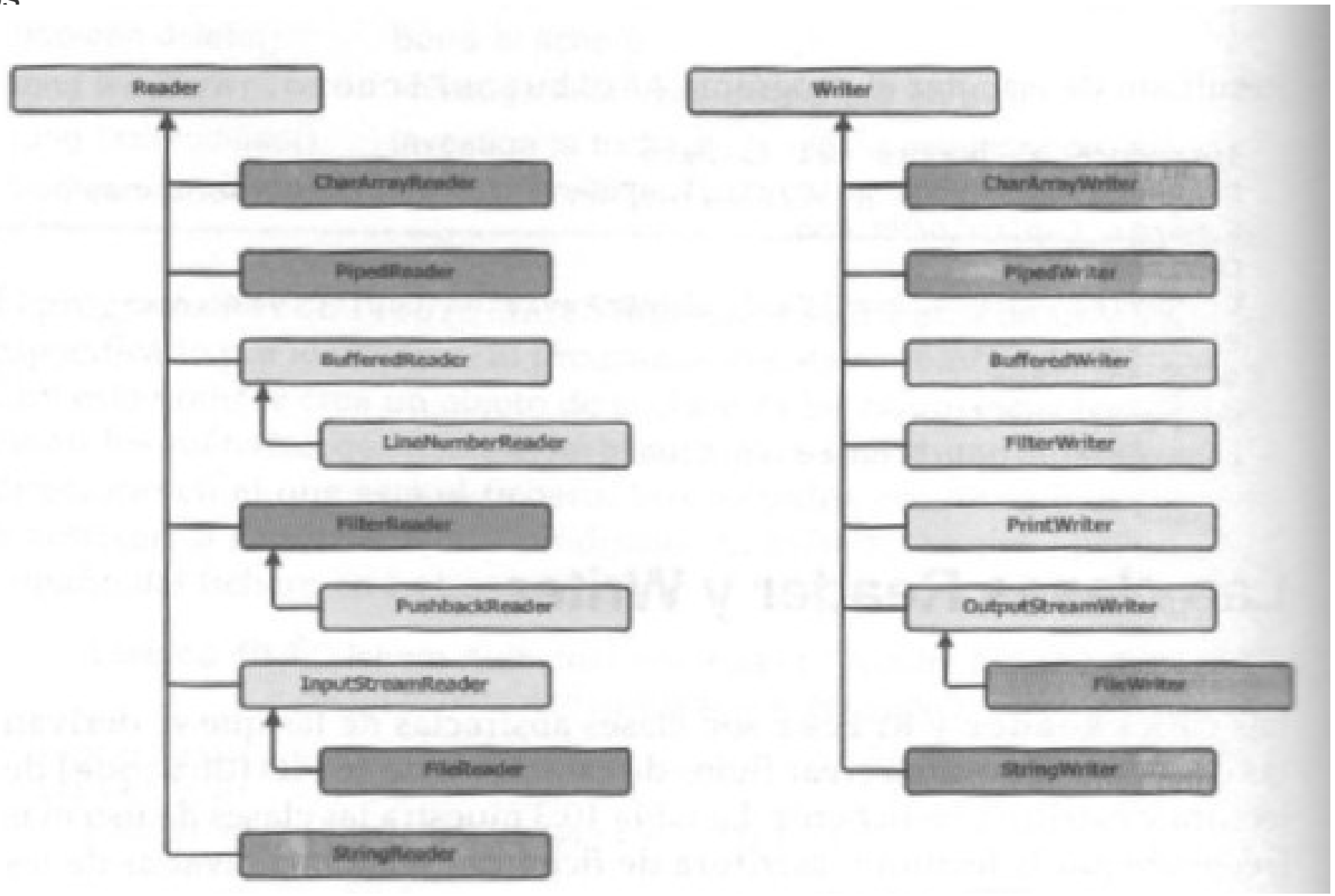
Las clases principales derivadas de las clases Reader y Writer,

- **FileReader** Permite crear flujos de bytes procedentes de un fichero dz caracteres. Esta clase se deriva de la clase InputStream
- **StreamReader** Ya vista como entrada de datos a través de la consola del sistema
- **FileWriter** Permite crear flujos de bytes para enviar datos a un fichero, se deriva de la clase OutputStreamWriter.
- **BufferedReader** Permite crear un flujo de entrada de datos con área de almacenamiento (buffer) con lo que permite la lectura de bloques de datos mayores que un byte.
- **BufferedWriter** Permite crear un flujo de envío de datos con área de almacenamiento (buffer).

Las clases Reader y Writer

Jerarquía de clases de Reader y Writer.

En la figura las clases con fondo gris definen flujos y las clases en fondo blan definen filtros



Las clases Reader y Writer

Si queremos escribir un fichero de texto línea a línea en Java se recomienda utilizar un objeto de flujo derivado de la clase FileWriter y utilizar un filtro de la clase BufferedWriter, tal como muestra el programa GeneraFichero que se recoge a continuación. En éste se crea el objeto de la clase FileReader; fLS y el objeto fS de la clase BufferedReaer; sobre este objeto se aplica el método newLine() que escribe líneas de texto en el fichero.

```
import java.io.*;
import java.util.Scanner;
public class GeneraFicheroTexto {
     public static void main(String[] args) {
          Scanner sc = new Scanner (System.in);
          String nombre;
          try {
               FileWriter flS=new FileWriter("Agenda.txt");
               BufferedWriter fS=new BufferedWriter(flS);
               do {
                    System.out.print ("Introduce un nombre: ");
                    nombre = sc.nextLine();
                    if (nombre.length () >0) {
                    System.out.print ("Teléfono:");
                    String teléfono = sc.nextLine();
                    fS.write(nombre +"," +teléfono) ;
                    fS.newLine() ;
               }
          }while (nombre. length () >0);
               fS.close();
          }catch(IOException e) {
               System.out.println("Error en el fichero");
               }    }    }
```

```
Introduce un nombre: Margarita
Teléfono:600523526
Introduce un nombre: Patricio
Teléfono:855362456
Introduce un nombre: Diana
Teléfono:523874125
Introduce un nombre:

Fichero Agenda.txt
(dentro del
workspace,proyecto)
Margarita,600523526
Patricio,855362456
Diana,523874125
```

Las clases Reader y Writer

El código LeeFicheroTexto siguente, muestra cómo leer el fichero Agenda.txt creado con el programa GeneraFicheroTexto. Este fichero se lee línea a línea y cada una de estas líneas se desglosa en dos datos: nombre y teléfono, localizando la Posición del la coma que separa estos datos con el método indexOf y se extrae del texto los dos datos con el método substring.

```
import java.io.*;
public class LeeFicheroTexto {
     public static void main(String[] args) {
          String texto="";
          try {
               FileReader flE=new FileReader("Agenda.txt");
               BufferedReader fE=new BufferedReader (flE) ;
               while(texto != null) {
                     texto = fE.readLine();
                     if(texto != null) {
                          int posi=texto.indexOf(",");
                          String nombre=texto.substring(0,posi);
                          String teléfono=texto.substring(posi+1);
                          System.out.print ("Nombre: "+nombre);
                          System.out.println(" Teléfono: "+ teléfono);
                     }
               }
               fE.close();
               }catch (IOException e) {
                    System.out.println("Error en el fichero");
               }
     }
}
```

```
Nombre: Margarita Teléfono: 600523526
Nombre: Patricio Teléfono: 855362456
Nombre: Diana Teléfono: 523874125
```

Las clases InputStream y OutputStream

Las clases InputStream y OutputStream son clases abstractas de las que se derivan las clases principales de lectura y escritura del lenguaje Java a través de flujos de bytes (conocidos como streams). En la tabla adjunta, se recogen

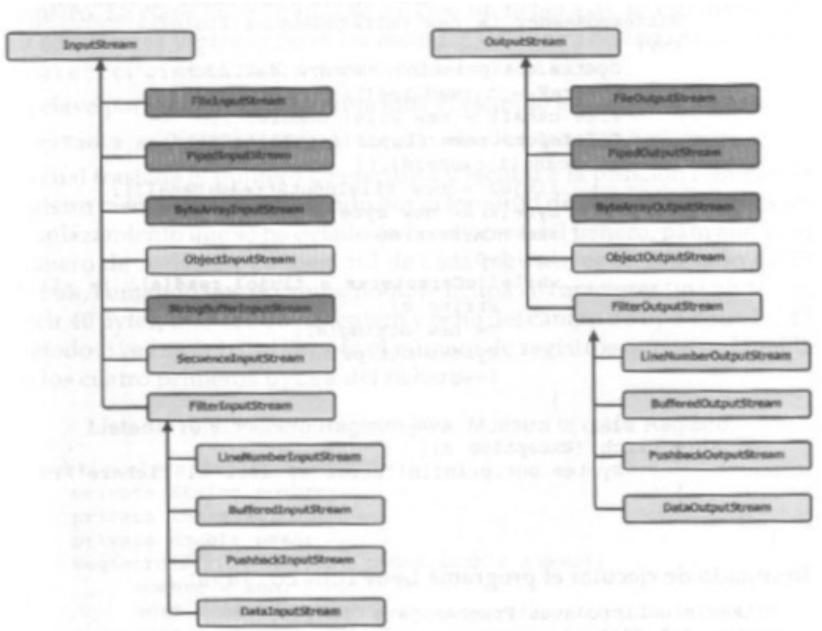
Las clases principales derivadas de InputStream y OutputStream,

- **FileInputStream** El constructor de la clase toma como argumento el nombre de un fichero o un objeto de la clase File y abre el fichero para lectura.
- **FileOutputStream** El constructor de la clase toma como argumento el nombre de un fichero o un objeto de la clase File y abre el fichero para escritura, bien sobreescribiendo su contenido o añadiendo.
- **DataInputStream** a clase DataInputStream deriva de la clase Filter- InputStream que a su vez deriva de InputStream. El constructor de la clase DataInputStream admite como argumento un objeto de la clase FileInputStream. Un objeto de la clase DataInput Stream se utiliza para filtrar un flujo de bytes y convertir éstos en primitivas válidas en el lenguaje Java.

Las clases InputStream y OutputStream

Las clases principales derivadas de InputStream y OutputStream,

- **ObjectOutputStream** El constructor de esta clase toma como argumento un objeto de la clase FileOutputStream. El objeto creado de ObjectOutput Stream crea un flujo de serialización de objetos, es decir, un flujo de bytes que escribe en un fichero la estructura de un conjunto de objetos.
- **ObjectInputStream** El constructor de esta clase toma como argumento un objeto de la clase FileInputStream. El objeto creado de Object InputStreanm crea un flujo de serialización de objetos, es decir, lee desde un fichero la estructura de un conjunto de objetos, previamente archivados con el flujo ObjectOutputStream.



Jerarquía de las clases InputStream y OutputStream

Ficheros de acceso directo o aleatorio

La clase RandomAccessFile posibilita la creación de ficheros de acceso directo, para lo cual cuenta con el método seek con el que el cursor de lectura y escritura en el fichero se puede situar en cualquier posición del fichero, fijando ésta contado el número de bytes desde el inicio del fichero. Los métodos readChar, readInt, readDouble, writeChar, writeInt y writeDouble posibilitan la lectura y escritura en el fichero. El método length retorna la longitud en bytes del fichero. En Java al no existir estructuras de registro resulta más laborioso el manejo de conjuntos de datos de diferentes tipos, si bien el adecuado diseño de clases posibilita la gestión de registros. En el ejemplo que se recoge en el codigo siguiente se utiliza la clase Registro recogida en el Registro para definir un registro. La clase TablaBaseDatos crea un fichero de acceso directo en su constructor y proporciona los métodos de escribeRegistro, leeRegistro, cierraTabla y dameNúmeroRegistros.

Ficheros de acceso directo o aleatorio

La clave para entender cómo funciona el ejemplo está en la sentencia:

```
miTabla. seek ((nRegistro-1) *longitudR+desplaza) ;
```

La cual traslada el puntero de escritura y lectura a la posición número de registro menos uno, multiplicado por la longitud de cada registro, más un desplazamiento que se ha dejado en la cabecera del fichero, para anotar el número de registros. La longitud de cada registro, es en este caso de 52 bytes, contando que el campo nombre ocupa 20 caracteres Unicode, es decir 40 bytes, más cuatro del entero y ocho del campo de tipo double. El método cierraTabla () anota el número de registros que tiene la tabla en los cuatro primeros bytes del fichero.

Ficheros de acceso directo o aleatorio

```
import java.io.*;
public class TablaBaseDatos {
     private RandomAccessFile miTabla;
     private int númeroRegistros;
     private int longitudR = 52;
     private int desplaza = 4;
     TablaBaseDatos (String tabla){
     try {      miTabla= new RandomAccessFile(tabla, "rw");
          if ( miTabla.length() == 0) númeroRegistros= 0;
     else númeroRegistros= miTabla.readInt();
     }catch( IOException e) { e.getMessage (); }     }
     public int dameNúmeroRegistros () {
          return númeroRegistros;          }
     public Registro leeRegistro(int nRegistro)
throws IOException{
          String nombre = "";
          int edad;
          double peso;
          miTabla.seek((nRegistro-1)*longitudR+desplaza) ;
          for(int i =0; i<20; i++)
               nombre += miTabla.readChar();
          edad = miTabla.readInt();
          peso= miTabla.readDouble();
          return (new Registro(nombre,edad,peso));         }
     public void escribeRegistro (String nombre,int edad, double peso,int nRegistro)
throws IOException{
          int longitud = nombre.length();
          miTabla.seek ((nRegistro-1) *longitudR+desplaza);
          for (int i=0; i< 20; i++) {
               if(i < longitud ) miTabla.writeChar(nombre.charAt(i));
               else miTabla.writeChar(' ');
          } miTabla.writeInt (edad);
          miTabla.writeDouble (peso);
          númeroRegistros++;          }
     public void cierraTabla() throws IOException{
          miTabla.seek(0);
          miTabla.writeInt(númeroRegistros);
          miTabla.close();
     }  }
```

```
public class Registro{
          private String nombre;
          private int edad;
          private double peso;
          Registro(String nom,int nEdad,double nPeso){
               nombre = nom;
               edad = nEdad;
               peso = nPeso;
          }
          public String muestraRegistro() {
                return nombre +" "+ edad+ " "+ peso;
          }
          public String dameNombre(){
          return nombre;
          }     }
```

Ficheros de acceso directo o aleatorio

```
import java.util.*;
public class VerTabla {
     public static void main(String[] args) throws Exception {
           Scanner sc = new Scanner (System.in);
           TablaBaseDatos t=new TablaBaseDatos("Datos.dir");
           System.out.print("Nombre: ");
           String nombre=sc.nextLine ();
           System.out.print("Edad: ");
           int edad =sc.nextInt ();
           System.out.print("Peso: ");
           double peso = sc.nextDouble();
           int nR = t.dameNúmeroRegistros();
           nR++;
           t.escribeRegistro(nombre,edad, peso, nR) ;
           for (int i=1; i<= nR; i++) {
                Registro r = t.leeRegistro(i);
                System.out .println(r.muestraRegistro());
           }
     t.cierraTabla();
     }
}
```

```
Primera ejecución
Nombre: Esteban
Edad: 41
Peso: 70
Esteban              41 70.0
Quinta ejecución
Nombre: Paulina
Edad: 52
Peso: 71
Esteban              41 70.0
María                26 60.0
Juan                 45 75.0
Adriana              36 65.0
Paulina              52 71.0
Archivo Datos.dir
    E s t e b a n
        )@Q€     M a r í a
                   @N      J u a
n                                  -
@RÀ     A d r i a n a
            $@P@     P a u l i n a
                       4@QÀ
```

Serialización

La serialización es el proceso por el cual un objeto o una colección de objetos se convierten en una secuencia de bytes, que pueden ser almacenados en un fichero y recuperados posteriormente Para su uso. Se dice que el objeto u objetos tienen persistencia. Cuando se serializa un objeto se almacena la estructura de la clase y todos los objetos referenciados por éste.

Java proporciona los interfaces Serializable, Externalizable, Replaceable y Resolvable del paquete java.io para resolver diferentes casos de serialización. La serialización más sencilla es la que proporciona la interface Serializable. Para utilizar esta capacidad la clase cuyos objetos deseamos serializar deben implementar la interface Serializable. Se utilizan las clases ObjectInputStream y ObjectOutputSream con los métodos writeObject y readObject.

Serialización

Al serializar un objeto se serializan todas las propiedades, tanto primitivas como objetos miembros, los cuales debe ser a su vez serializables, es decir, que sus clases implementen la interfaz Serializable. Los objetos de tipo static no son serializados, por lo que deben ser guardados y recuperados de forma independiente. Los métodos writeInt, writeDouble, readInt y readDouble, permiten serializar de forma explícita variables primitivas de sus respectivos tipos.

El código siguiente es la aplicación Serializable1 en la que se utiliza un objeto de nombre flujo_salida de la clase FileOutputStream y un objeto de nombre filtro_salida della clase ObjectOutputStream para crear un fichero de nombre temporal, en el cual almacenar el objeto p de la clase Arbol (ver código).

Serialización

El método writeObject envía este objeto al fichero. Una vez cerrado el fichero se vuelve a abrir con los objetos flujo_entrada y filtro_entrada de las clases FileInputStream y ObjectInputStream respectivamente. El método readObject recupera el objeto y lo referencia en el objeto q. Este objeto contiene los datos del objeto anterior p.

```
import java.io.*;
public class Arbol implements Serializable{
     private static final long serialVersionUID = 1L;
     private String nombreVulgar;
     private String nombreCientifico;
     private double alturaMedia;
     Arbol (String nombre) {
     this.nombreVulgar=nombre;
     }
     public void ponNombreCientifico (String nombre) {
          this.nombreCientifico=nombre;
     }
     public void ponAlturaMedia (double altura) {
          alturaMedia=altura;
     }
     public String muestraArbol () {
           return nombreVulgar+ " "+ nombreCientifico + " " + alturaMedia;
     }
}
```

Ficheros de acceso directo o aleatorio

```
import java.io.*;
import java.util.*;

public class Serializable1{
     public static void main(String argumentos []) {
          try {
               Scanner sc = new Scanner (System.in);
               System.out .print ("Nombre vulgar: ");
               String texto=sc.nextLine ();
               Arbol p = new Arbol(texto);
               System.out.print ("Nombre cientifico: ");
               texto=sc.nextLine ();
               p.ponNombreCientifico (texto);
               System.out.print ("Altura media: ");
               double altura = sc.nextDouble ();
               p.ponAlturaMedia (altura) ;
               FileOutputStream f=new FileOutputStream("tmp");
               ObjectOutputStream fis=new ObjectOutputStream(f);
               //escribimos en tmp el objeto p
               fis.writeObject(p);
               fis.close();
               //leemos desde tmp el objeto q
               FileInputStream fe = new FileInputStream("tmp");
               ObjectInputStream fie = new ObjectInputStream(fe);
               Arbol q = (Arbol)fie.readObject();
               System.out.println(q.muestraArbol());
               fie.close();
          }catch ( Exception e) {
               System.out .println(e.getMessage ());
          }
     }
}
```

Resultado:

```
Nombre vulgar: Ciruelo Jardín
Nombre cientifico: Prunus
Pissardii
Altura media: 3
Ciruelo Jardín Prunus
Pissardii 3.0
Archivo tmp:
¬í sr serializable.Arbol
 D alturaMediaL
nombreCientificot
Ljava/lang/String;L
nombreVulgarq ~ xp@      t
Prunus Pissardiit Ciruelo
JardÃn
```

Lectura de ficheros con métodos de la clase Scanner

En la versión Java 5 se incorporaba como novedad la clase Scanner que ya se ha visto para leer datos en la consola del sistema, sus métodos también se pueden usar para leer un fichero. El código recoge el programa LeeFicheros en el que se lee línea a línca el propio fichero fuente del programa y se muestra en la consola del sistema.

```
import java.util.*;
import java.io.*;
public class LeeFicheroS {
public static void main(String[] args) {
     try{
          Scanner sc = new Scanner(new File("C:\\Users\\RUTA_del_FICHERO\\LeeFicheroS.java"));
          while (sc.hasNextLine()) {
               String texto = sc.nextLine ();
               System.out .println(texto);
          }
     }catch (Exception e) {}
}
}
```

2 hilos de ejecución

28.2.2024

Enrique Krause Buedo

Concepto de proceso e hilo de ejecución

Los sistemas operativos denominados multitarea o multiproceso como los sistemas Unix, Linux, Windows NT, Windows 2000 o XP, permiten ejecutar varios programas simultáneamente. En la mayoría de casos el procesador, antes de la primera década del siglo, era único y sólo puede atender a un proceso cada si bien repartiendo el tiempo entre todas las tareas, de tal forma que la sensación de que está ejecutando varias tareas a la vez. Estas independientes se conocen con el nombre de procesos. El usuario sistema puede conocer qué procesos se están ejecutando. Estos procesos son muy independientes unos de otros, de tal forma que podrían pararse unos sin que se vean afectados los demás. El sistema operativo cuida esta independencia precisamente por la seguridad y estabilidad del propio sistema operativo.

Concepto de proceso e hilo de ejecución

El concepto de hilo de ejecución parte de un principio similar, si bien es el sistema operativo quien gestiona éstos, sino las propias aplicación en ejecución, es decir, un proceso podría tener varios hilos de ejecución incluso actuar sobre la misma colección de datos. Una aplicación realizada en el lenguaje Java puede generar varios hilos de ejecución, detenerlos cuando sea necesario, sincronizarlos para que no generen conflictos al actuar sobre un mismo objeto, etc.

La clase Thread

La clase Thread permite crear estos hilos de ejecución. El constructor esta clase permite especificar un nombre a cada nuevo hilo generado. grupo (ThreadGroup) y un destino. El grupo es una denominación que permite gestionar los hilos formando grupos de hilos. El destino es objeto que implementa la clase Runnable y cuando se ejecuta el método Start sobre el nuevo hilo, se ejecuta el método run que contiene la al que pertenece el objeto destino.

La tabla siguiente muestra los métodos más importantes de la clase Thread

Luego, el código que sigue muestra el ejemplo PrimerHilo, en el cual se crea un de ejecución sobre la propia clase. Este hilo de ejecución continúa en ejecución hasta que finaliza el tiempo de retardo que se impone con el método sleep, de tal forma que acaba antes de ejecutarse el método principal en el hilo inicial, que el hilo hijo creado.

La clase Thread

Métodos más importantes de la clase Thread

- **activeCount** Investiga el número de hilos en el grupo activo.
- **chekAccess** Investiga si el hilo activo un hilo.
- **currentThread** Devuelve una referencia al hilo que se está ejecutando.
- **destroy** Produce la terminación inmediata del un hilo.
- **getName** Investiga el nombre de un hilo.
- **GetPriority** Investiga la prioridad del un hilo.
- **GetThreadGroup** Devuelve una referencia al grupo al que pertenece un hilo.
- **Interrupt** Interrumpe la ejecución de un hilo.
- **isAlive** Investiga si un hilo esta vivo.
- **IsDaemon** Investiga si un hilo es un demonio, es decir, si es un proceso desvinculado de la consola del sistema.

La clase Thread

Métodos más importantes de la clase Thread

- **IsInterrupted** Investiga si un hilo ha sido interrumpido.
- **Join** Mezcla dos hilos en uno.
- **run** Comienza ha ejecutar el código destino de un hilo.
- **setDaemon** Marca un hilo como demonio.
- **SetName** Cambia el nombre de un hilo.
- **setPriority** Permite cambiar la prioridad de ejecución de un hilo.
- **Sleep** Cesa la ejecución de un hilo durante el tiempo especificado.
- **Start** Inicia la ejecución de un hilo. Es decir, se llama al método run del destino.
- **toString** Muestra los datos de un hilo.
- **Yield** Permite que el procesador ejecute otros hilos.

La clase Thread

La sentencia:

```
PrimerHilo destino = new PrimerHilo();
```

Crea un objeto de la propia clase de nombre destino. La sentencia:

```
Thread nuevoHilo = new Thread (destino, "Nuevo hilo");
```

Crea un nuevo hilo que se activa con la sentencia

```
nuevoHilo.start
```

```
public class PrimerHilo implements Runnable{
     static final short limiteS = 1000;
     private short limiteI =0;
     public static void main(String argv[]) {
          PrimerHilo destino = new PrimerHilo();
          Thread nuevoHilo = new Thread(destino, "Nuevo hilo");
          nuevoHilo.start();
          System.out.println("El hilo principal ha terminado");
     }
          public void run() {
               while( limiteI++ < limiteS) {
                    System.out.print ("El hilo actual es: ");
                    System.out.println( Thread.currentThread().toString());
                    try{
                         Thread.currentThread().sleep(100);
                    }catch (InterruptedException e) {
                         System.out.println("Hilo hijo interrumpido" );
                    }
                    System.exit(1);
               }   } }
```

```
El hilo principal ha terminado
El hilo actual es: Thread[Nuevo hilo,5,main]
```

La clase Thread

En el código siguiente se muestra cómo se puede parar el nuevo hilo antes de termine su ejecución. Para esto se crea un nuevo hilo auxiliar con la misma referencia del que se quiere detener, se asigna a null la referencia del que se quiere detener y se aplica el método interrupt al hilo auxiliar

```
public class SegundoHilo implements Runnable{
     static final short limiteS = 1000;
     private short limiteI =0;
     public static void main(String args[]) {
          SegundoHilo destino =new SegundoHilo();
          Thread nuevoHilo=new Thread (destino, "Nuevo hilo");
          nuevoHilo.start() ;
          Thread hiloAux = nuevoHilo;
          hiloAux.interrupt() ;
          System.out.print ("El hilo principal ");
          System.out.println("ha terminado");
     }
     public void run() {
          while( limiteI++ < limiteS) {
               System.out.print ("El hilo actual es: ");
               System.out.println(Thread.currentThread().toString());
               try{
                    Thread.currentThread().sleep(1000);
               }catch (InterruptedException e) {
                    System.out.print ("Interrumpido ");
                    System.out.println("el hilo hijo");
               }
               System.exit (1);
          }    }         }
```

```
El hilo principal ha terminado
El hilo actual es: Thread[Nuevo hilo,5,main]
Interrumpido el hilo hijo
```

La clase Thread

Un hilo que se acaba de crear será un hilo nuevo, y no hará nada hasta que se ejecute sobre este el método **start, el cual buscará el método run de la clase de destino, e iniciará su ejecución**. Una vez ejecutado el método start el hilo se convierte en ejecutable, entiéndase que no se denomina ”ejecutado", sino, "ejecutable", en modo condicional, ya que dependerá la gestión de la CPU, si en un preciso instante este hilo está o no en ejecución. Un hilo en estado ejecutable puede pasar a estado bloqueado, bien porque intenta actuar sobre un objeto sincronizado, que está siendo intervenido por otro hilo, bien porque está esperando una Operación de entrada y salida, o porque los métodos ***sleep*** y ***wait*** lo han bloqueado. Un hilo que ha terminado de ejecutar el método ***run*** es un hilo muerto, ya no tiene nada que hacer. Igualmente el método ***interrupt*** ejecutado sobre un hilo puede pasarlo a hilo muerto.

La tabla a continuación recoge los estados en los que se pueden encontrar un hilo.

La clase Thread Estados de un hilo.

Nuevo (new) El hilo está creado pero no se ha ejecutado el método start. En esta situación no admite la ejecución de más métodos que start.

Ejecutable(runnable) Un hilo está en estado ejecutable cuando se ha ejecutado el método start sobre él, siempre que no esté bloqueado ni muerto. Un hilo ejecutable puede estar ejecutándose o no dependiendo del reparto de tiempos de la CPU.

Bloqueado(blocked) Un hilo que está en estado ejecutable puede pasar a bloqueado si se ejecuta sobre el método sleep o el método wait También puede estar bloqueado por estar esperando por una operación de entrada salida, o bien porque intenta actuar sobre un objeto que está bloqueado por otro hilo.

Muerto (dead) Un método está en estado muerto cuando ha finalizado la ejecución del método run de la clase de destino. O bien se ha ejecutado sobre él el método interrupt.

Sincronización de hilos

Cuando una aplicación genera varios hilos que actúan sobre un mismo objeto se pueden producir colisiones que alteren el normal funcionamiento del programa. Una técnica que evita esta colisión es declarar el objeto como sincronizado, con lo cual todos los hilos de ejecución que incidan sobre el objeto sincronizado bloquean éste mientras actúan, y lo liberan cuando termina, para que otro hilo posteriormente lo gestione. Se puede sincronizar un objeto, tal como indican las siguientes sentencias:

```
Synchronized(campoTexto)([código que actúa sobre el objeto])
```

También se puede sincronizar un método:

```
synchronized void miMetodo()( [código del método] )
```

Para una buena sincronización de un objeto, o de una variable, es necesario que todos los métodos que puedan actuar sobre éstos deben estar sincronizados.

Sincronización de hilos

Si todos los métodos de una clase están sincronizados se garantiza que sobre un objeto de la clase no puedan actuar dos métodos simultáneamente, si bien pudiera ocurrir que sobre varios objetos distintos, estén actuando simultáneamente varios hilos de forma sincronizada.

En una sincronización mal hecha podría ocurrir que un hilo bloquee un recurso de forma indefinida, impidiendo que otros hilos realizaran su tarea, para evitar este riesgo, es conveniente sincronizar sólo los recursos imprescindibles.

Es posible declarar un hilo como demonio (daemon), es decir, un hilo desvinculado de la línea de comandos. Un caso típico de hilo que se ejecuta segundo plano (background) es por ejemplo un escuchador de un puerto de comunicaciones, que está pendiente de las peticiones de comunicación que accedan a ese puerto. Para convertir un hilo en demonio se utiliza el método setDaemon.

Métodos wait y notify de la clase Object

Al margen de la sincronización automática, vista en los epígrafes anteriores, se puede lograr una sincronización específica gestionada por el programador que en algunos casos será más eficiente. La gestión de la sincronización se realiza con los métodos wait y notify de la clase Object. El método wait bloquea un objeto hasta que el hilo que ha tomado posesión de él reciba la ejecución del método ***notify***, por parte de un hilo con permiso para su desbloqueo. El método ***notifyAll*** notifica a todos los hilos del grupo la orden de desbloqueo.

En el código siguiente se recoge el programa Sincroniza que crea un objeto la clase Compartido que se recoge en el segundo código; después se crean dos hilos de ejecución a través de la clase MiHilo, los cuales comparten un mismo objeto dato. En el primer hilo, al ejecutar éste, se solicita la ejecución del método dameDato de la clase Compartido. En este método e bloquea el objeto dato, con el método ***wait***, hasta que el otro hilo, con el método cambiaDato, desbloquea el objeto.

Métodos wait y notify de la clase Object

```
public class Sincroniza {
     public static void main(String args[]) {
          Compartido dato= new Compartido("Primer dato");
          System.out.println("El dato inicial es \"Primer dato\"");
          MiHilo hilo1= new MiHilo("uno",dato);
          MiHilo hilo2= new MiHilo("dos",dato);
          hilo1.start();
          hilo2.start();
          System.out.println("Fin de programa");
     }
}
```

```
// métodos wait y notify
public class Compartido {
     String identificador;
     Compartido(String id){
          identificador=id;
     }
     public synchronized void cambiaDato(String datoNuevo) {
          identificador=datoNuevo;
          notify();
     }
     public synchronized String dameDato() {
      try{
          wait ();
     }catch (Exception e) {};
     return identificador;
     }
}
```

Métodos wait y notify de la clase Object

```
public class MiHilo extends Thread{
     Compartido dato;
     String id;
     MiHilo(String id,Compartido dato){
          this.dato=dato;
          this. id=id;
     }
     public void run() {
          if (id.equals ("uno")) {
               System.out.println("El hilo "+id+ " va
ha pedir el dato y lo bloquea");
               System.out.println("El hilo "+id+ " ha
conseguido el dato: "+dato.dameDato());
          }else {
               System.out.println("El hilo "+id+ " va a
cambiar el dato y desbloquea este");
               dato.cambiaDato ("Nuevo dato");
          }
     }
}
```

```
El dato inicial es "Primer dato"
Fin de programa
El hilo uno va ha pedir el dato y lo bloquea
El hilo dos va a cambiar el dato y desbloquea este
El hilo uno ha conseguido el dato: Nuevo dato
```

Prioridades de ejecución de hilos

Los hilos tienen por defecto la prioridad de ejecución normal de valor 5. Con el método setPriority se puede cambiar esta prioridad desde el valor mínimo 1 hasta el máximo de 10. El método **getPriority** investiga la prioridad de ejecución de un hilo. Si hay varios hilos de igual prioridad, el sistema operativo repartirá en tiempo entre ellos ejecutándolos de torma alternativa. El código siguiente muestra cómo cambiar la prioridad de ecución de un hilo. En el programa se crean dos hilos de la clase MiHilo2 que se recoge en el listado codigo que continua. Al hilo uno se le da la máxima prioridad y al hilo dos la prioridad normal de 5. A pesar de que le hilo dos empieza antes que el hilo uno, el uno termina antes que el hilo dos.

Prioridades de ejecución de hilos

```
public class VerPrioridad {
     public static void main(String[] args) {
               MiHilo2 hiloRapido = new MiHilo2 ("uno");
               MiHilo2 hiloLento = new MiHilo2 ("dos");
               //Establece la prioridad al máximo; valor 10
               hiloRapido.setPriority(Thread.MAX_PRIORITY);
               //Establece la prioridad al valor normal 5
               hiloLento.setPriority(Thread.NORM_PRIORITY);
               hiloLento.start();
               System.out.println("El hilo rápido tiene la prioridad de " +
hiloRapido.getPriority());
               System.out.println("El hilo lento tiene la prioridad de " +
hiloLento.getPriority());
               hiloRapido.start();
     }
}

public class MiHilo2 extends Thread{
      String id;
      MiHilo2 (String id){
                this.id=id;
      }
     public void run(){
      for(long i=0;i<200_000;i++){
           try {
               sleep(100);
          } catch (InterruptedException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
          }
          System.out.println("El hilo "+id+ " ya por el número "+i);
         System.out.println("El hilo "+id+ " ha terminado");
      }
     }
}
```

```
El hilo rápido tiene la prioridad de 10
El hilo lento tiene la prioridad de 5
El hilo uno ya por el número 0
El hilo uno ha terminado
El hilo dos ya por el número 0
El hilo dos ha terminado
El hilo uno ya por el número 1
El hilo uno ha terminado
El hilo dos ya por el número 1
El hilo dos ha terminado
El hilo uno ya por el número 2
El hilo uno ha terminado
El hilo dos ya por el número 2
El hilo dos ha terminado
El hilo uno ya por el número 3
El hilo uno ha terminado
```

Ejemplo

El siguiente código cuenta el tiempo que tarda un hilo secundario en terminar el proceso después del primario

```
public class HiloCronometro extends Thread{
     HiloCronometro(String nombre){
          super(nombre);
          start();
     }
     public void run() {
           long inicio1=System.currentTimeMillis();
           for (int i=0; i<10; i++) {
                long inicio2 = System.currentTimeMillis();
                do {
                }while((System.currentTimeMillis()-inicio2)<=1000);
                System.out.println((System.currentTimeMillis()-inicio1)/1000+" seg." );
                }
           System.out.println("Finaliza el hilo secundario");
     }
}


public class VerHiloCronometro {
     public static void main(String args[]) {
          HiloCronometro hiloCronometro=new HiloCronometro ("Cronometro") ;
          System.out.println(" finaliza el hilo principal.");
     }
}
```

```
 finaliza el hilo principal.
1 seg.
2 seg.
3 seg.
4 seg.
5 seg.
6 seg.
7 seg.
8 seg.
9 seg.
10 seg.
Finaliza el hilo secundario
```

3 excepciones

28.2.2024

Enrique Krause Buedo

Concepto de excepción

El lenguaje Java tiene desde su origen una vocación de independencia del sistema operativo, lo que le confiere unas características que permiten su ejecución en redes de ordenadores con diferentes sistemas Operativos. acceso remoto a objetos en otras máquinas de la red, etc. Este alejamiento del sistema operativo le supone perder parte del control sobre los periféricos, sobre los cuales actúa en base a multitud de capas de software. Por otro lado, son los periféricos la fuente principal de errores o de situacio- nes inesperadas en la ejecución de un Programa, bien sea porque se cuenta con un fichero que no existe, una impresora que no tiene papel, o una dirección de un ordenador (URL) inaccesible.

Todas estas circunstancias pueden provocar desastres en la ejecución del Programa que suponen la finalización de la aplicación de forma descontrolada, dejando ficheros abiertos, pérdida de datos por falta de su archivo, etc.

Concepto de excepción

Para evitar estos problemas y dotar al lenguaje Java de una estructura robusta frente a circunstancias sobre las que no podría tener control, los diseñadores del Java dotaron a éste de un Mecanismo sofisticado de control de errores, creando el concepto de excepción, que es un Objeto que se genera automáticamente cuando se produce un acontecimiento circunstancial dañino Para el normal funcionamiento del programa, y que puede ser previsto y controlado. Este objeto generado, que se denomina excepción, sobre el acontecimiento producido y transmite esta información al método que le ha llamado, o al usuario a través del correspondiente mensaje. Una excepción provoca que se paralice la ejecución de una o varias sentencias implicadas en el incidente. Normalmente, y sobre todo en aplicaciones complejas, esto no supone la finalización anormal de la aplicación, sino que se desvía el flujo del programa para evitar ese accidente.

Captura de las excepciones

Las aplicaciones deben tener, y en muchos casos está obligado por el compilador, un mecanismo de captura de excepciones mediante el cual, en sentencias, o bloques de sentencias, en las que es previsible que se produzca una excepción, ésta sea capturada, o enviada al método que le ha llamado. En general las excepciones deben ser tratadas adecuadamente con el objetivo de evitar efectos no deseados sobre la aplicación. Hay dos técnicas de gestión de excepciones que corresponden a la captura y a la expulsión de la excepción. En unos casos simplemente se expulsa y en otros se captura. Si el que expulsa la excepción es el método main se produce la finalización anormal de la aplicación. Hasta aquí es lo que venía ocurriendo en los ejemplos, a partir de aquí se utilizará la captura como técnica preferente, sin olvidar que la expulsión de excepciones también es una técnica recomendada, siempre que se establezcan métodos capaces de capturar estas excepciones en métodos específicos para esta tarea,

Captura de las excepciones

En los programas de ejemplo, vistos hasta aquí, se ha utilizado el modificador throws Exception, o throws IOException, para expulsar las posibles excepciones que se podrían producir en los procesos de entrada y salida de datos a la consola del sistema. Esta sentencia rechaza la excepción y la envía al método que la ha llamado, éste puede hacer lo mismo enviándole la excepción al método ancestro de éste, o bien capturarla con la estructura try-catch que se recoge en el epígrafe siguiente.

Cómo se procesan excepciones con try catch

La estructura formada con try catch permite crear bloques alternativos que se ejecutan cuando se produce una excepción. El código muestra la estructura de try catch, en éste hay un bloque definido por **try** (intenta) que es el bloque de sentencias en las cuales se puede producir una excepción y son las que se ejecutan siempre en circunstancias normales de ejecución.

Cómo se procesan excepciones con try catch

Seguido de este bloque está uno o varios bloques de tipo **catch** (captura)que se ejecutan cuando se ha producido una excepción, estos bloques se ensayan en serie, de tal forma que si la excepción producida no es del tipo del primer bloque catch se investiga el siguiente y así hasta que se encuentra un bloque de sentencias del tipo de la excepción que se ha producido. Generalmente se pone un bloque último para una excepción genérica de la clase Exception que es la clase principal de todas la excepciones. Después de los bloques de tipo catch puede aparecer un bloque de tipo **finally** que se ejecuta siempre, es decir, se haya producido una excepción o no. Estos bloques se pueden utilizar, por ejemplo, para cerrar un fichero o guardar un dato. El bloque de tipo finally son opcionales y en pocas ocasiones tienen un cometido claro, a pesar de que algunos autores de libros de Java los consideren muy importante para un buen estilo de programación.

Cómo se procesan excepciones con try catch

Estructura de sentencia try...catch.

```
try{

[Bloque de sentencias que se ejecutan en circunstancias normales]

}catch (ClasedeExcepcion1 e){

[Bloque que se ejecutan si se ha Excepcion1]

}catch ( ClasedeExcepcion2 e){

[Bloque que se ejecutan si se ha producido una excepción de la clase2]

----

}catch ( Excepcion e){

[Bloque que se ejecutan si se ha producido una excepción no capturada]

} finally {

[Bloque de sentencias que se ejecutan siempre]

}
```

Cómo se procesan excepciones con try catch

clases generadores excepciones de uso frecuente

ArithmeticException Excepción en una operación aritmética, p.ej. división por cero

ArrayStoreException Excepción en una operación en el uso de un array

NegativeArraySizeException Excepción en una operación en el uso de un array

NullPointerException Excepción producida por una referencia a un objeto nulo

SecurityException Excepción en el sistema de seguridad

EOFException Excepción producida por alcanzar el final de un fichero

IOException Excepción en un proceso de entrada y salida

FileNotFoundException Excepción por no encontrar un fichero

NumberFormatException Excepción en la conversión de una cadena de caracteres a un valor numérico.

Cómo se procesan excepciones con try catch

El programa siguiente, muestra cómo utilizar la estructura try catch para capturar y procesar una excepción genérica producida en el bloque try; bien sea por un problema de entrada de datos desde la consola, o bien por la división, si el denominador vale cero. Si se produce una excepción se genera un objeto de la clase Exception que en el ejemplo se ha denominado e, este objeto contiene información sobre el error producido. En el bloque catch se muestra parte de esta información a través del método getMessage. La sentencia siguiente muestra los nombres de los métodos que han llamado al método que produjo el error hasta el método que captura la excepción. Este método es útil cuando existen múltiples llamadas en cadena a métodos y el programador puede perder las circunstancias en las que se produce el error.

El método getMessage perteneciente a la clase Exception muestra un mensaje asociado al tipo de excepción que se ha producido. El método printStrackTrace imprime en la consola del sistema los nombres de los métodos llamados hasta el método que ha producido la excepción. El programa repite la solicitud de los dos números mientras que (while) la variable error tenga el valor true, este valor lo toma al pasar por el bloque catch.

Cómo se procesan excepciones con try catch

```
import java.io.*;
public class Excepcion1{
     public static void main(String args[]) {
          BufferedReader teclado = new BufferedReader(new InputStreamReader (System.in));
          boolean error=false;
          do {
               try{
                    error=false;
                    System.out.print ("\nIntroduce un número: ");
                    String texto= teclado.readLine();
                    int i = Integer.parseInt(texto);
                    System.out.print ("Teclea otro número: ");
                    texto= teclado.readLine();
                    int j = Integer.parseInt(texto);
                    System.out.printf ("La división %d/%d =%d\n",i, j, i/j);
               }catch (Exception e) {
                    //Muestra error producido
                    System.out.println(e.getMessage ());
                    /*Imprime las llamadas producidas hasta el método que genero el error*/
                    e.printStackTrace ();
                    error=true;
               }
          }while (error);
     }
}
```

```
Introduce un número: 45
Teclea otro número: 0
/ by zero
java.lang.ArithmeticException: / by zero

Introduce un número:  at A01PruebasYo/exception1.Excepcion1.main(Excepcion1.java:17)
```

Cómo se procesan excepciones con try catch

Un adecuado sistema de control de excepciones permite la realización de programas más estables y robustos, además de permitir un mantenimiento sencillo y sin problemas en su explotación.
El programa siguiente muestra cómo se puede capturar de forma selectiva una excepción. Si la excepción producida no es de la clase del primer catch se investiga el siguiente, y así hasta que algún bloque catch lo capture. El último bloque captura cualquier clase, ya que todas las clases de excepciones derivan de la clase Exception.

```
import java.io.*;
public class Excepcion1{
     public static void main(String args[]) {
          BufferedReader teclado = new BufferedReader(new InputStreamReader (System.in));
          boolean error=false;
          do { try{
                    error=false;
                    System.out.print ("\nIntroduce un número: ");
                    String texto= teclado.readLine();
                    int i = Integer.parseInt(texto);
                    System.out.print ("Teclea otro número: ");
                    texto= teclado.readLine();
                    int j = Integer.parseInt(texto);
                    System.out.printf ("La división %d/%d =%d\n",i, j, i/j);
               }catch (Exception e) {
                    //Muestra error producido
                    System.out.println(e.getMessage ());
                    /*Imprime las llamadas producidas hasta el método que genero el error*/
                    e.printStackTrace ();
                    error=true;
               }
          }while (error);     }    }
```

```
Teclea un número:152
Teclea otro número: 0

División por cero

Teclea un número:52
Teclea otro número: a
Error: en la
conversión

Teclea un número:45
Teclea otro número: 20
La división 45/20 =2
```

Generación de excepciones; sentencia throw

La sentencia throw permite al programador generar excepciones específicas para la lógica de sus aplicaciones. La clase Persona muestra cómo el método ponEdad genera una excepción cuando la edad es menor de cero y mayor de cien. La sentencia genera una excepción con el mensaje "Edad no válida" que es enviada al método que ha llamado a este método, a través de los modificadores throws Exception. Cuando se genera la excepción no se ejecuta la sentencia de asignación que hay después de la sentencia de generar la excepción, ya que el método termina con la generación de la excepción. En el método que ha realizado la llamada a ponEdad se captura la excepción a través del bloque catch de la clase Exception.

```
public class Persona{
     private int edad;
     public void ponEdad(int edadNueva) throws Exception{
          if (edadNueva < 0 || edadNueva > 100 ) throw (new Exception("Edad no valida"));
          edad=edadNueva;
     }
     public String toString() {
          return "Edad: "+ edad;
     }
}
```

Generación de excepciones; sentencia throw

```
package exception1;
import java.util.*;
public class MiniAgenda {
     public static void main(String[] args) {
          boolean error=false;
          Scanner se =new Scanner(System.in);
          Persona p = new Persona ();
          do {
               error=false;
          System.out.print ("Introduce la edad: ");
          try{
               int edad=se.nextInt();
               p.ponEdad(edad) ;
               System.out.println(p.toString());
          }catch (Exception e) {
               System.out.println(e.getMessage ());
               error=true;
          }
          }while (error) ;
     }
}
```

```
Introduce la edad: -5
Edad no valida
Introduce la edad: 110
Edad no valida
Introduce la edad: 80
Edad: 80
```

Excepciones personalizadas

El programador de Java puede crear su propia clase de excepciones, el programa VerMiError muestra cómo utilizar una excepción de validar datos de la clase MiError. La clase Validar utiliza esta excepción, si el dato a validar no cumple las condiciones especificadas en el argumento del método esValido de la clase Validar.

Si en la llamada a esValido no se especifica un mensaje en el segundo argumento, la excepción toma el mensaje por defecto que está en la clase MiError es decir, "Me ha fallado".

Para probar este ejemplo, se necesita tener tres ficheros independientes VerMiError , MiError y Validar. Al ejecutar la aplicación VerMiError, si el usuario escribe un número negativo se produce la excepción y se emite el mensaje "Me ha Fallado".

```
public class MiError extends RuntimeException{
     public MiError() {
          super ("¡Ya me ha fallado!");
     }
     public MiError (String mensaje) {
               super (mensaje) ;
     }
}
```

Las excepciones personalizadas permiten precisar las circunstancias en las que se producen las excepciones, además de permitir modificar el idioma de los mensajes de error.

Generación de excepciones; sentencia throw

```
public class Validar {
     public static void esValido (boolean valido) throws MiError{
          if(!valido) throw new MiError();
     }

     public static void esValido(boolean valido, String mensaje) throws MiError{
          if (!valido) throw new MiError(mensaje);
     }
}

import java.util.*;
public class VerMiError {
     public static void main(String args[]) {
          boolean error=false;
          int nu=0;
          Scanner sc = new Scanner(System.in);
          do {
               error=false;
          try {
               System.out.print ("Introduce un número");
               System.out.print(" entero positivo: ");
               nu = sc.nextInt();
               Validar.esValido(nu > 0);
          }catch (MiError e) {
               System.out.println(e.getMessage());
               error=true;
          }catch (Exception e) {
               System.out.println(e.getMessage());
               error=true;
          }
          }while (error) ;
          System.out.println("Admitido el valor "+nu);
     }  }
```

```
Introduce un número
entero positivo: -50
¡Ya me ha fallado!
Introduce un número
entero positivo: 40
Admitido el valor 40
```

Ejemplos

```
import java.io.*;
public class Repaso {
     public static void main(String args[]) {
          String nombre= "";
          int número;
          boolean tieneÑ=false;
          BufferedReader teclado = new BufferedReader (new InputStreamReader(System.in));
          /*Solicita un número y no termina hasta que el número sea mayor de 10 y menor de 30*/
          try{  do{
                    System.out.print ("Teclea un número mayor");
                    System.out.print(" de 10 y menor de 30: ");
                    nombre = teclado.readLine();
                    número = Integer.parseInt (nombre);
               }while (número <= 10 || número >=30);
               if (número % 5 == 0) {
                    System.out.print ("El número "+ número);
                    System.out.println(" es divisible por 5" );
               }else {
                    System.out.print ("El número "+ número );
                    System.out.println(" no es divisible por 5");   }
               System.out.print ("Escribe tu nombre: ");
               nombre= teclado.readLine();
          }catch (Exception e) {
               System.out.println(e.getMessage ()); }
          System.out.print ("Tu nombre tiene");
          System.out.println(nombre.length()+" caracteres");
          System.out .print ("Empieza por la letra ");
          System.out.println(nombre.charAt (0));
          System.out.print ("Termina por la letra ");
          System.out.println(nombre.charAt(nombre.length()-1));
          for (número=0;número<nombre.length();número++) {
               if (nombre.charAt(número) =='ñ'|| nombre.charAt(número) == 'n') {
                    tieneÑ= true;   }          }
          System.out.print ("Tu nombre en mayúsculas es: ");
          System.out.println(nombre.toUpperCase ());
          System.out.print ("Tu nombre en minúsculas es: ");
          System.out.println(nombre.toLowerCase ());
          if (tieneÑ)System.out.println("Tu nombre tiene A");
          else System.out.println("Tu nombre no tiene Ñ");
     }  }
```

```
Teclea un número mayor de 10 y menor de 30: 5
Teclea un número mayor de 10 y menor de 30: 50
Teclea un número mayor de 10 y menor de 30: 20
El número 20 es divisible por 5
Escribe tu nombre: juan
Tu nombre tiene4 caracteres
Empieza por la letra j
Termina por la letra n
Tu nombre en mayúsculas es: JUAN
Tu nombre en minúsculas es: juan
Tu nombre tiene A
```

3 Utilidades de fechas, matemáticas y otras

28.2.2024

Enrique Krause Buedo

La clase Date y GregorianCalendar

La clase Date permite crear objetos que informan de la fecha y hora sistema. Las clases **Calendar** y **GregorianCalendar** permiten crear objetos que pueden gestionar fechas y tiempos. El método *setTime* permite asignar a un objeto de la clase GregorianCalendar un objeto de clase **Date**, con los datos de fecha y hora del sistema. El método *set* permite establecer otra fecha al objeto de la clase GregorianCalendar, este cambio obviamente no afecta al sistema.

La clase **TimeZone** se puede utilizar para identificar la franja horaria local, el método get Id de esta clase identifica la franja horaria con la que trabaja el sistema, obteniendo ésta de un objeto del método getTimezone de la clase CregorianCalendar.

La aplicación VerFecha muestra la fecha del sistema, crea un objeto de la clase Date y otro de la clase GregorianCalendar. El método setT asigna al objeto calendarioG los datos de fecha y hora contenidos en objeto hoy.

28.2.2024

Enrique Krause Buedo

La clase Date y GregorianCalendar

Observe cómo el índice del array diasSemana está disminuido en una unidad, ya que la propiedad DAY_OF_WEEK de la inteface Calendar cuenta los días desde el uno y no desde el cero como las demás propiedades.

```
public class VerFecha {
     public static void main(String args[]) {
           String diasS[] = {"domingo", "lunes", "martes", "miércoles", "jueves",
"viernes", "sábado"};
           String mesA[] = {"enero","febrero","marzo", "abril", "mayo", "junio", "julio",
"agosto", "septiembre", "octubre", "noviembre", "diciembre"};
           Date hoy = new Date();
           GregorianCalendar calenG = new GregorianCalendar ();
           calenG.setTime (hoy) ;
           System.out.print ("Hoy es ");
           System.out.print (diasS [calenG.get(Calendar.DAY_OF_WEEK)-1]);
           System.out.print("  "+calenG.get(Calendar. DAY_OF_MONTH));
           System.out.print ("-"+mesA[calenG.get(Calendar.MONTH)]);
           System.out.print ("-"+calenG.get (Calendar. YEAR)) ;
           System.out.print(" a las "+(calenG.get(Calendar.HOUR)));
           System.out.print (":"+calenG.get(Calendar.MINUTE));
           System.out.println(":"+calenG.get (Calendar.SECOND));
     }
}
```

```
Hoy es domingo  26-febrero-2023 a las 4:57:54
```

La clase Date y GregorianCalendar

El programa VerFecha2 muestra cómo se puede formatear un dato de fecha según las características de regionalización instaladas en el ordenador. El objeto formateadorFecha de la clase DateFormat obtiene las propiedades de formato de datos numéricos y de fecha y aplica este estilo al dato de fecha obtenido con el objeto hoy de la clase Date.

```
import java.util.*;
import java.text.*;
public class VerFecha2 {
     public static void main(String args[]) {
          Date hoy = new Date();
          DateFormat formateadorFecha = DateFormat.getDateInstance();
          String fechaLocal=formateadorFecha.format(hoy);
                    System.out.println(fechaLocal) ;
     }
}
```

```
En un ordenador con OS en Inglés:
Feb 26, 2023
```

La clase Date y GregorianCalendar

El programa VerFecha3, muestra cómo se puede formatear un dato de fecha según las características de regionalización que desee el programador, independientemente de las que estén instaladas en el ordenador. También muestra cómo, los diferentes estilos dentro de estas propiedades de regionalización, o también conocidas como propiedades de localización. Los valores constantes DateFormat.SHORT, DateFormat.MEDIUM, DateFormat.LONG y DateFormat FULL determinan los estilos del formato de salida dentro de una misma colección de regionalización. El segundo parámetro de getDateInstance establece qué colección de propiedades debe ser usada, así el valor Locale.US determina el grupo de propiedades de EEUU

La sentencia:

```
Locale fEspañol=new Locale("es", "ES", "Traditional_WIN");
```

Crea el objeto fEspañol con los datos del formato de fecha y números d español.

La sentencia:

```
DateFormat formato = new SimpleDateFormat("EEE d MMM yy", fEspañol);
```

crea un formato siguiendo el estilo español, si bien, con estructura libr definidas por la formula: "EEE d MMM yy".

La clase Date y GregorianCalendar

```
import java.util.*;  import java.text.*;
public class VerFecha3 {
public static void main(String args[]) {
       Date hoy = new Date();
       System.out.println("FECHA CON FORMATO AMERICANO");
       String fechaL=DateFormat.getDateInstance(
                  DateFormat.SHORT, Locale.US).format(hoy);
       System.out.println("Fecha corta: "+fechaL);
       fechaL= DateFormat.getDateInstance (
                  DateFormat.MEDIUM, Locale.US).format(hoy);
       System.out.println("Fecha media: "+fechaL);
       fechaL= DateFormat.getDateInstance (
                  DateFormat.LONG, Locale.US).format(hoy);
       System.out.println("Fecha larga: "+fechaL);
       fechaL=DateFormat.getDateInstance (
                  DateFormat.FULL, Locale.US).format(hoy);
       System.out.println("Fecha completa: "+fechaL);
       System.out.println ("FECHA CON- FORMATO ESPAÑOL");
       Locale fEspañol=new Locale(
                  "es", "ES", "Traditional_WIN");
       fechaL = DateFormat.getDateInstance(
                  DateFormat.SHORT, fEspañol).format(hoy);
       System.out .println("Fecha corta: "+fechaL);
       fechaL= DateFormat.getDateInstance (
                  DateFormat.MEDIUM, fEspañol).format(hoy);
       System.out.println("Fecha media: "+fechaL);
       fechaL= DateFormat.getDateInstance(
                  DateFormat.LONG, fEspañol).format(hoy) ;
       System.out.println("Fecha larga: "+fechaL);
       fechaL = DateFormat.getDateInstance (
                  DateFormat.FULL, fEspañol).format(hoy) ;
       System.out.println("Fecha completa: "+fechaL);
      DateFormat formato = new SimpleDateFormat ( "EEE d MMM yy", fEspañol);
      fechaL = formato.format(hoy) ;
      System.out.println("Fecha formato libre "+fechaL);
 }    }
```

```
FECHA CON FORMATO AMERICANO
Fecha corta: 2/26/23
Fecha media: Feb 26, 2023
Fecha larga: February 26, 2023
Fecha completa: Sunday, February 26, 2023
FECHA CON- FORMATO ESPAÑOL
Fecha corta: 26/2/23
Fecha media: 26 feb 2023
Fecha larga: 26 de febrero de 2023
Fecha completa: domingo, 26 de febrero de 2023
Fecha formato libre dom 26 feb 23
```

La clase Date y GregorianCalendar

En el siguiente código se recoge el programa NúmeroDias que muestra cómo se puede obtener una fecha a través de su formato como valor numérico de tipo long. Con esta utilidad se puede calcular cualquier fecha sumando a este formato el número de milisegundos de una fecha para obtener otra. Para obtener el valor en milisegundos de un fecha que no sea la actual se puede obtener con un objeto de la clase Calendar y el método getTimeInMillis.

```
import java.util.*;
import java.text.*;
public class numeroDias {
     public static void main(String args[]) {
                Scanner sc = new Scanner (System.in);
                System.out.println( "¿Cuantos días tardarás en terminar de preparar el
curso?");
                long nDias=Long.parseLong(sc.nextLine ());
                long nMSegundos=nDias*24*60*60*1000;
                Date dia= new Date(System.currentTimeMillis()+nMSegundos);
                DateFormat formateadorFecha = DateFormat.getDateInstance();
                System.out.println("El final del curso será el día: ");
                System.out .println(formateadorFecha.format(dia));
     }
}
```

```
¿Cuantos días tardarás en terminar de preparar el curso?
19
El final del curso será el día:
Mar 17, 2023
```

La clases DecimalFormat

La clase DecimalFormat permite dar formato, es decir, controlar el número de decimales a mostrar, siguiendo diferentes estilos, según los idiomas o regiones culturales, así de esta forma es posible poner como separador de decimales la coma en español o el punto en formato inglés. En el código que sigue, se recoge la aplicación FormatoDecimal en la que se lee un número introducido por el usuario a través del teclado en formato español, es decir con el separador de decimales usando el carácter coma para esto se usa la sentencia:

```
sc.nextDouble ();
```

que lee un dato de tipo double en la entrada a través del teclado, siendo sc un objeto de la clase Scanner. La sentencia:

```
sc.useLocale (fEspañol) ;
```

determina que se lea el dato de entrada siguiendo el formato numérico del español. Posteriormente se muestra este dato con el formato "#########. ##" que se pasa al constructor de la clase DecimalFormat con la sentencia:

```
DecimalFormat fE = new DecimalFormat ("#########.##");
```

La clases DecimalFormat

La sentencia:

`fE.format(euros)`

obtiene la cadena de caracteres del valor numérico en el formato fE obtenido anteriormente. El programa posteriormente muestra el mismo dato en el formato americano.

```
Cantidad en Pts en formato español.
52.000
52000 pesetas =312.53 €
Cantidad en Pts en formato americano
52,000
52000 pesetas =312.53 €
```

```
import java.util.*;
import java.text.*;
public class FormatoDecimal {
     static public void main(String args[]) {
           Scanner sc=new Scanner (System.in);
           Locale fEspañol = new Locale("es", "ES", "Traditional_WIN");
           sc.useLocale(fEspañol);
           DecimalFormat fE = new DecimalFormat ("########.##");
           System.out.println("Cantidad en Pts en formato español.");
           double ptas=sc.nextDouble();
           double euros=ptas/166.386;
           System.out.println(fE.format(ptas) +" pesetas ="
           + fE.format (euros)+" €");
           Locale fAmericano = new Locale( "en", "US", "Traditional_WIN");
           sc.useLocale (fAmericano);
           DecimalFormatSymbols dFSAmericano= new DecimalFormatSymbols (fAmericano) ;
           DecimalFormat fA = new DecimalFormat ("########.##", dFSAmericano) ;
           System.out.println("Cantidad en Pts en formato americano");
           ptas=sc.nextDouble ();
           euros=ptas/166.386;
           System.out.println(fA.format (ptas) + " pesetas =" + fA.format (euros)+" €");
     }
}
```

La clase Random

La clase Random permite crear objetos que generan números aleatorios. La aplicación EnterosAleatorios muestra cómo crear diez números enteros aleatorios. El método nextDouble permite obtener números reales aleatorios. El método getGaussian permite obtener números que siguen una distribución en campana de Gauss.

https://es.wikipedia.org/wiki/Funci%C3%B3n_gaussiana

https://www.teorema.top/campana-de-gauss-todo-lo-que-necesitas-saber/

```
import java.lang.*;
import java.util.*;
public class EnterosAleatorios {
     public static void main(String args[]) {
          Random aleatorios = new Random();
          System.out.println("Diez números aleatorios");
          for(int i=0;i<10;++i)
               System.out.println(aleatorios.nextInt ());
     }
}
```

```
Diez números aleatorios
-1427938072
-1516639031
-1698919994
-66137488
1217570164
1646667523
-1266732434
-94599185
1921724366
1087834919
```

La clase Math

La clase java. lang.Math proporciona métodos estáticos para realizar operaciones matemáticas trigonométricas, raíces cuadradas, potencias, etc así como las constantes PI y E. La tabla recoge los

Métodos principales de esta clase.

abs() Valor absoluto de un número

ceil() Entero más cercano por encima.

floor() Entero más cercano por debajo.

round() Entero más cercano.

rint(double) Retorna un entero mas próximo dado un número double.

IEEremainder (double,double) Calcula el resto de la división.

La clase Math

Métodos principales de esta clase.

exp() Calcula el exponencial de un número.

log() Calcula el logaritmo natural en base e de un número.

max(,) Retorna el máximo entre dos valores.

min(,) Retorna el mínimo entre dos valores.

random() Genera un número aleatorio entre 0 y 1.

power(,) Retorna el primer argumento elevado al segundo.

sqrt() Calcula la raíz cuadrada de un número.

La clase Math

En la versión java 5 se incluyó la importación de clase de forma estática, lo que permite acceder a sus métodos sin tener que incluir el nombre de la clase, así la sentencia:

```
import static java.lang.Math.*;
```

permite acceder a los métodos y constantes de la clase Math, así la sentencia:

```
System.out.println ("Valor absoluto" + Math.abs(cantidad));
```

Se usaría de la siguiente forma:

```
System.out.println("Valor absoluto " + abs(cantidad));
```

El siguiente código muestra cómo se aproximan un valor entero partiendo de un valor decimal.

La clase Math

```
import java.util.Scanner;
public class AproximaEnteros {
     public static void main(String[] args) {
               Scanner sc=new Scanner (System.in);
               System.out .println("Introduce una cantidad con decimales.");
               double cantidad = sc.nextDouble ();
               System.out.println("Valor inicial "+cantidad);
               System.out.println("Valor absoluto " + Math.abs (cantidad));
               System.out.println ("Aproximación superior entera " + Math.ceil(cantidad));
               System.out.println("Aproximación inferior entera " + Math.floor(cantidad));
               System.out.println ("Aproximación más cercana " + Math.rint(cantidad));
               System.out.println("Aproximación por \"casting\" " + ((int) cantidad));
     }
}
```

Para acceder a los métodos, y constantes de la clase Math desde java 6, no es necesario importar el paquete Math, sólo es necesario anteponer, al nombre del método, el nombre de la clase, por ejempl Math.PI, para acceder a la constante PI.

```
Introduce una cantidad con decimales.
12.56897
Valor inicial 12.56897
Valor absoluto 12.56897
Aproximación superior entera 13.0
Aproximación inferior entera 12.0
Aproximación más cercana 13.0
Aproximación por "casting" 12
```

La clase Cipher

La clase Cipher proporciona la funcionalidad para cifrar (encriptar) y cifrar (desencriptar) y pertenece a la extensión Java Cryptographic Extension (JCE). Para cifrar un texto hay que obtener una instancia de clase Cipher para lo que es necesario obtener un generador de claves la clase estática **KeyGenerator**. El método Proporciona la clave generadora indicándole el algoritmo de cifrado en este caso el algoritmo (Data Encryption Standad). En el código que sigue se recoge el programa CifradorDescifrador que cifra y descifra un array de bytes. En este programa la sentencia:

```
cifrador = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

Obtiene una instancia a la clase Cipher indicando el algoritmo de cifrado DES, la modalidad ECB y el esquema PKCS5Padding. A continuación se establece el "cifrador" en la modalidad de "cifrar", con la sentencia:

```
cifrador.init (Cipher.ENCRYPT_MODE, clave);
```

La clase Cipher

Después de rellenar el array de bytes de nombre textoACif£rar con los caracteres se desea cifrar, se procede al cifrado con la sentencia:

```
byte[] textoCifrado = cifrador.doFinal (textoACifrar);
```

Para descifrar se procede a poner el cifrador en la modalidad de descifrar con la sentencia:

```
cifrador.init (Cipher.DECRYPT_MODE, clave);
```

y después se procede al descifrado con la sentencia:

```
byte[] textoDesCifrado = cifrador.doFinal (textoCifrado);
```

La clase Cipher

```
import javax.crypto.*;
public class CifradorDescifrador {
     public static void main(String[] args) throws Exception{
          Cipher cifrador;
          KeyGenerator generadorClave = KeyGenerator.getInstance("DES");
          SecretKey clave = generadorClave.generateKey ();
          cifrador = Cipher.getInstance ("DES/ECB/PKCS5Padding");
          cifrador.init (Cipher.ENCRYPT_MODE, clave);
          byte [] textoACifrar = "Armando esteban Quito".getBytes();
          System.out.println("\nTexto antes de cifrar:");
          for(int i=0;i<textoACifrar.length; i++) {
               char c=(char)textoACifrar[i];
               System.out.print(c);
          }
          byte[] textoCifrado = cifrador.doFinal(textoACifrar);
          System.out.println("\nTexto cifrado:");
          for(int i=0;i <textoCifrado.length; i++) {
               char c=(char)textoCifrado[i];
               System.out.print(c);
          }
          cifrador.init(Cipher.DECRYPT_MODE, clave);
          byte[] textoDesCifrado = cifrador.doFinal (textoCifrado);
          System.out.println("\nDespués de descifrar:");
          for(int i=0;i<textoDesCifrado.length; i++) {
               char c=(char)textoDesCifrado[i];
               System.out.print (c);
          }
     }
}
```

```
Texto antes de cifrar:
Armando esteban Quito
Texto cifrado:
?????&?!?b?$??????{!
Después de descifrar:
Armando esteban Quito
```

La clase StringTokenizer

La clase StringTokenizer permite crear un analizador de cadenas de caracteres con objeto de obtener palabras y datos contenidos en dicha cadena. La aplicación VerPalabras a continuación, solicita al usuario una frase con varias palabras y la descompone en palabras. El método nextToken obtiene la próxima palabra de la cadena. El método hasMoreTokens investiga si quedan más palabras por extraer. El método countTokens cuenta el número de palabras que tiene la cadena. El constructor de la cadena de la clase StringTokenizer toma como argumento la cadena resultado de leer el flujo de datos del objeto teclado.

```
import java.util.*;
public class VerPalabras {
     public static void main(String[] args) {
     Scanner sc = new Scanner (System.in);
     try{
          System.out.println("Escriba una frase");
          String frase = sc.nextLine();
          StringTokenizer cadena = new StringTokenizer (frase);
          int nPalabras = cadena.countTokens();
          System.out.println(nPalabras+" palabras");
          while (cadena.hasMoreTokens())
               System.out.print (cadena.nextToken ()+"+*");
     }catch (Exception e) {
          System.out.println(e.getMessage ());
     }  }  }
```

```
Escriba una frase
el veloz murciélago que
andaba de noche en una
baticueva
10 palabras
el+*veloz+*murciélago+*que+*a
ndaba+*de+*noche+*en+*una+*ba
ticueva+*
```

4 Comunicación a través de sockets

28.2.2024

Enrique Krause Buedo

Concepto de socket

Una alternativa para realizar la comunicación entre dos ordenadores, es a través de vínculos de comunicación conocidos por la denominación genérica de sockets. Tanto los clientes como los servidores atienden peticiones de otros, en un puerto determinado del ordenador, si se recibe la petición de establecer una conexión, y esta conexión es aceptada, se establece un intercambio de datos a través del protocolo TCP (protocolo de control de transporte), se dice, entonces que se han creado dos sockets, es decir, dos puntos de comunicación en la red. Cada socket se identifica por la dirección IP del ordenador en el que se atiende a demandas de comunicación de otros y a un número de puerto. Este tipo de comunicación, conocida como comunicación a través de conexión, es más fiable que otra en la que no quede establecida esta conexión, ya que TCP proporciona un conjunto de mensajes de reconocimiento de ambos sockets, así como servicios de detección y recuperación de errores.

Comunicaciones en Internet

La red de redes que es Internet utiliza mayoritariamente sockets para la comunicación entre ordenadores. Todos los ordenadores que forman Internet están registrados en el Centro de Información de redes de Inte (InterNIC). Las comunicaciones en Internet se realizan a través de paquetes de datos del protocolo de Internet (IP). Para identificarse, los ordenadores utilizan su dirección IP, que consiste en una serie de cuatro números enteros del 0 al 255. Adicionalmente y para facilitar la identificación usuarios, se utilizan nombres que identifican las direcciones IP a través de tablas de asignación de nombres, que constituyen el Sistema de Nombres de Dominio (DNS). Como se ha mencionado anteriormente, la comunicación entre dos computadoras se puede realizar en cualquier número de puerto, si bien con objeto de estandarizar, en Internet hay un conjunto de puertos por defecto que utilizan todos, y por lo tanto, sólo necesitamos conocer la dirección IP de un servidor Web, o su nombre de dominio establecer una conexión con ese servidor.