Programación en JAVA

Indice

Humberto Martínez Barberá
Dep. Informática, Inteligencia Artificial, y Electrónica
Universidad de Murcia

http://www.um.es/~humberto
humberto@fcu.um.es

Tema I: Fundamentos del Lenguaje JAVA

El lenguaje JAVA es un lenguaje orientado al objeto, que toma sus caracteristicas principalmente de C++ y C. Asociado al lenguaje JAVA existe una máquina virtual JAVA, que ejecuta (interpreta) los byte-codes que se producen al compilar un programa fuente escrito en JAVA. De esta forma, tanto los programas JAVA fuentes como los compilados se pueden considerar independientes de la máquina. Así, cada fabricante de sistemas operativos ha de desarrollar una que sea 100% compatible con la implementación de Sun (estándar).

1. Descripción del entorno de desarrollo

Para poder desarrollar programas en lenguaje JAVA, necesitamos, al menos, el entorno de desarrollo mínimo, que es el JDK. Del mismo existen varias versiones, aunque las más extendidas son dos: JDK 1.0.2 y JDK 1.1.5. Básicamente son similares, aunque la última incorpora la posibilidad de uso de JavaBeans, nuevo modelo de eventos del AWT, serialización, etc. En principio, salvo que se diga explícitamente de otra forma, nos referiremos a la versión JDK 1.1.5. Los elementos básicos del mismo son:

- javac
- java
- appletviewer
- javadoc
- documentación de las APIs

El JDK incluye herramientas muy básicas. Para desarrollar programas JAVA, interesa tener un entorno que incorpore, además, un debugger gráfico y un RAD (Rapid Application Development). Ejemplos de estos entornos son el Symantec Visual Café, el Borland JAVA Builder, y el SGI Cosmo Code.

Cuando se desarrolla un programa en lenguaje JAVA, se sigue el siguiente proceso:

- Edición
- Compilación
- Carga de las clases
- Verificación de los byte-codes
- Ejecución (interpretación)

Básicamente,se pueden tener dos tipos de programas en JAVA, dependiendo de dónde se produzca la ejecución de los mismos:

- Aplicaciones: se ejecutan como otra aplicación en el sistema operativo (java)
- Applets: se ejecutan dentro de un navegador de web (appletviewer)

Como primer ejemplo de programas en JAVA, siguen un applet y una aplicación:

Applet

```
/*** Este es nuestro primer Applet de Java.
     * Al ejecutarlo nos muestra
     * mensaje de bienvenida al curso.
     * @author Humberto Martínez Barberá
     * @version 1.0
*/

import java.awt.graphics;
import java.applet.applet;
```

```
public class BienvenidaApplet extends Applet {

public void paint (Graphics g) {
        g.drawString("Bienvenidos al curso de
Java",20,20);
}
}
```

```
<HTML>
<HEAD>
<TITLE>Applet de Bienvenida</TITLE>
</HEAD>

<BODY>
<APPLET CODE="BienvenidaApplet.class"
WIDTH=300 HEIGHT=200></APPLET>
</BODY>
</HTML>
```

Aplicación

```
/*** Esta es nuestra primera aplicación en
Java.
     * Al ejecutarla nos muestra
     * mensaje de bienvenida al curso.
     * @author Humberto Martínez Barberá
     * @version 1.0
*/

import java.lang.*;

class BienvenidaAplicacion {

public static void main (String args[]) {
        System.out.println ("Bienvenidos al
curso de Java");
          }
}
```

2. Elementos básicos del lenguaje

2.1. Los comentarios en JAVA

Tenemos tres formas distintas de realizar comentarios en JAVA:

- //, comentarios de una sola línea (estilo C++).
- /* */, comentarios de múltiples líneas (estilo C).
- /** */, comentarios de múltiples línes (estilo JavaDoc).

Los comentarios JavaDoc sirven para que se pueda generar de forma automática la documentación de un programa, en el lenguaje HTML. El texto generado es similar a la documentación de las APIs incluidas en el JDK. De esta forma se definen una serie de campos (que comienzan con el símbolo @), que contendrán la información que aparecerá en la documentación. Se pueden distinguir campos para:

- Clases e interfaces
- Atributos
- Métodos y constructores

2.1.1. Campos para clases e interfaces

- @author *nombre*
- @version *código-versión*
- @see *clase*

Un ejemplo de un comentario para una clase:

```
/*** Una clase representando una ventana en la pantalla
* Por ejemplo:
* <pre>
* Window win = new Window(parent);
* win.show();
* </pre>
*
```

```
* @author Humberto Martínez Barberá
* @version %I%, %G%
* @see java.awt.BaseWindow
* @see java.awt.Button
*/

class Window extends BaseWindow {...}
```

2.1.2. Campos para atributos

- @see *nombre*

Un ejemplo de un comentario de un atributo:

```
/*** La coordenada X de una ventana
*
* @see window#1
*/

int x = 1263732;
```

2.1.3. Campos para constructores y métodos

- @see *clase*
- @param *nombre-parámetro descripción*
- @return *descripción*
- @exception *nombre-excepción descripción*

Un ejemplo de un comentario para un método:

```
/*** Devuelve el carácter en el índice especificado. Un índice
* va desde <tt>0</tt> 0 <tt>length() - 1</tt>.
*
* @param indice El índice del carácter que se desea.
* @return el carácter pedido.
* @exception StringIndexOutOfRangeException
* si el índice no esta en el rango<tt>0</tt>
* a <tt>length()-1</tt>.
*/

public char charAt (int indice) {...}
```

2.2. Variables y tipos básicos de datos

La declaración de variables en JAVA se hace de forma similar al C. En adelante, un *identificador* será el nombre de una variable. Así declararemos:

```
tipo identificador [, identificador];
```

donde *tipo* va a ser uno de los 8 tipos básicos de datos (primitivos) de JAVA. Estos son:

Nombre tipo	Clase	Tamaño
byte	entero	8-bit
short	entero	16-bit
int	entero	32-bit

long	entero	64-bit
float	real	32-bit
double	real	64-bit
char	carácter	16-bit (Código Unicode)
boolean	booleano	1 bit

2.3. Operadores

En JAVA se tienen, básicamente, los mismos operadores que en C. Estos se pueden clasificar en:

- Operadores aritméticos
- Operadores relacionales
- Operadores lógicos
- Operadores de bits
- Operadores de asignación

Estos operadores se muestran en las tablas siguientes, así como una breve descripción de su uso.

Operadores aritméticos

Operador	**Uso**	**Descripción**
+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	El resto de dividir op1 por op2
+	+ op	Indica un valor positivo
-	- op	La negación aritmética de op
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++op	Incrementa op en 1; evalúa el valor después de incrementar
--	op--	Decrementa op en 1; evalúa el valor antes de decrementar
--	--op	Decrementa op en 1; evalúa el valor depués de decrementar

Operadores relacionales

Operador	Uso	Devuelve verdadero si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 no son iguales

Operadores lógicos

Operador	Uso	Devuelve verdadero si
&&	op1 && op2	op1 y op2 son ambos verdaderos
||	op1 || op2	op1 o op2 son true
!	! op	op es falso

Operadores de bits

Operador	Uso	Descripción
>>	op1 >> op2	Desplaza bits de op1 a la derecha según op2
<<	op1 << op2	Desplaza bits de op1 a la izquierda según op2
>>>	op1 >>> op2	Desplaza bits de op1 a la derecha según op2 (sin signo)
&	op1 & op2	AND a nivel de bits
|	op1 | op2	OR a nivel de bits
^	op1 ^ op2	XOR a nivel de bits
~	~ op	Complemento a nivel de bits

Operadores de asignación

Operador	Uso	Descripción
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

JAVA es un lenguaje fuertemente tipado (no se pueden operar datos de distinto tipo). En el caso de operaciones donde así se requiera, existe la posibilidad de realizar una conversión explícita de tipo (cast de C). De esta forma se convierte el contenido de una variable (de un determinado tipo) al tipo que se exprese entre paréntesis. Un ejemplo podría ser:

```
int a;
long b;

a = (int) b;
```

2.4. Expresiones

Las expresiones en JAVA se forman de forma similar a las del lenguaje C. Así se pueden combinar variables y constantes por medio de los operadores descritos anteriormente. El lenguaje establece, además, prioridades a los operadores, de forma que sea claro el orden en que se aplican. Estos operadores se muestran, de mayor a menor prioridad, en la siguiente tabla:

Clase operador	Operadores
postfijos	[ ] . (params) expr++ expr--
unarios	++expr --expr +expr -expr ~ !

creación y cast	new (type)expr
multiplicadores	* / %
aditivos	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
AND de bits	&
XOR de bits	^
OR de bits	|
AND lógico	&&
OR lógico	||
condicional	? :
asignación	= += -= *= /= %= ^= &= |= <<= >>= >>>=

En el caso de haber más de un operador con la misma prioridad, esta se resuelve de izquierda a derecha.

2.5. Instrucciones condicionales

Se utilizan para romper las ejecución se cuencial de un programa. Son similares a las del lengueje C.

2.5.1. if-then-else

Con esta instrucción se ejecuta uno u otro bloque (*then* o *else*) según el valor de verdad de una expresión lógica (*if*). La segunda parte se puede omitir en caso necesario.

```
if (nota >= 9) {
     calificacion = "Sobresaliente";
} else if (puntuacion >= 7) {
     calificacion = "Notable";
} else if (puntuacion >= 5) {
     calificacion = "Aprobado";
} else {
     calificacion = "Suspenso";
}
```

2.5.2. switch-case

Similar a un encadenamiento de sentencias if-then-else. Con esta instrucción se compara el valor de una variable con cada uno de los distintos casos (*case*), y se ejecuta el bloque correspondiente. En caso de que no coincida ninguno, se ejecuta el bloque por defecto (*default*).

```
int dia;

switch (dia)
{
case 1:
     System.out.println("Lunes");
     break;
case 2:
     System.out.println ("Martes");
     break;
case 3:
     System.out.println ("Miércoles");
     break;
case 4:
     System.out.println ("Jueves");
     break;
case 5:
     System.out.println ("Viernes");
     break;
case 6:
     System.out.println ("Sábado");
     break;
case 7:
     System.out.println ("Domingo");
     break;
default:
     System.out.println ("Día no válido");
     break;
}
```

2.6. Instrucciones de control del flujo

2.6.1. for

Con esta instrucción podemos realizar bucles pre-prueba, a la vez que se incorpora en la semántica de la directiva el incremento de un contador.

```
int contador;

for (contador = 0; contador < 50; contador++)
{
      . . .
}
```

2.6.2. do-while

Con esta instrucción podemos realizar bucles post-prueba. Se ejecuta siempre, al menos, una vez.

```
int contador = 0;

do
{
     . . .
     contador++;
} while (contador < 50);
```

2.6.3. while

Con esta instrucción podemos realizar bucles pre-prueba.

```
int contador = 0;

while (contador < 50)
{
```

```
        . . .
        contador++;
}
```

2.6.4. break y continue

Sirven para terminar prematuramente un bucle (*break*), o para terminar la ejecución del paso actual de un bucle (*continue*). En el caso de tener bucles anidados, se pueden poner etiquetas para saber a cual nos referimos. Por ejemplo:

```
char a;
int i, j;

etiqueta1:
     for (i = 0; i < 10; i++)
     {
     etiqueta2:
          for (j = 0;  j < 10;   j++)
          {
               a = (char) System.in.read ();
               if ( a == "b")
               {
                    break etiqueta1;
               }
               if (a == "c")
               {
                    continue etiqueta2;
               }
          }
     }
```

2.6. Arrays

La declaración y uso de los arrays en JAVA es similar a la del lenguaje C, salvo que difieren en cuando se define el tamaño de un array. Se pueden declarar arrays multidimensionales, al igual que se pueden definir sobre cualquier tipo de datos de JAVA.

2.6.1. Declaración e inicialización

La declaración de los arrays se realiza de la siguiente forma:

```
tipo identificador[ ], ...;
```

donde *tipo* es cualquier tipo de datos. Se pueden inicializar escribiendo los elementos separados por comas, y agrupados por llaves en cada dimensión, de la siguiente forma:

```
tipo identificador[ ] = { valor, valor, ...., valor };
tipo identificador[ ]...[ ] = { { valor, valor, ...., valor }, ..., { ... }};
```

En el caso de la inicialización, esta corresponde a una declaración + creación + asignación inicial de valores al array.

2.6.2. Creación

Cuando se crea un array, se reserva espacio en memoria para almacenar todos los elementos. Así, este es el momento donde se indica el número de elementos que tendrá en cada dimensión. Se crean los arrays de la siguiente forma:

```
identificador = new tipo [tamaño];
identificador = new tipo [tamaño]... [tamaño];
```

Es importante indicar que el primer elemento de un array tiene un índice cero, y el último tiene un índice (tamaño - 1).

2.6.3. Uso

Para acceder al contenido de una celda de un array, simplemente se le dice la posición en cada una de las dimensiones.

Esto se realiza de la siguiente forma:

```
identificador[posición]...[posición]
```

Si queremos saber el tamaño de un array, e.d. el número de celdas que tiene, se puede usar la siguiente función:

```
identificador.length
```

En el caso de que intentemos acceder a una celda fuera de los límites del array (índice negativo, o índice mayor o igual que tamaño), se producirá una excepción del tipo *ArrayIndexOutBoundsException*.

Por ejemplo, el siguiente programa sumará una matriz dada a una introducida desde teclado:

```
/*** Esta es una aplicación de prueba de matrices en Java.
     * Al ejecutarla tendremos que introducir los elementos de
     * la matriz B.
     * @author Humberto Martínez Barberá
     * @version 1.0
*/

import java.lang.*;

class Matrices
{
      public static void main (String args[]) throws IOException
     {
         int b[ ][ ], c[ ][ ], a[ ][ ] = { { 1, 2, 3}, { 4, 5, 6}, {7, 8, 9} };
         int i, j;

         b = new int[3][3];
         for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
            {
                System.out.print ("celda " + i + "," + j + "? ");
                b[i][j] = (int) System.in.read ();
            }

         c = new int[3][3];
         for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                c[i][j] = a[i][j] + b[i][j];

         for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
               System.out.print ("celda " + i + "," + j + " = " +  c[i][j]);
     }
}
```

Tema II: Programación OO con JAVA

La programación orientada al objeto (OOP) no es sólo una característica semántica de un lenguaje de programación. También está relacionada con una metodología de programación. La OOP, en general, se caracteriza por tener una serie de propiedades. De ellas estudiaremos las relacionadas con el lenguaje JAVA. Básicamente, la OOP se basa en el concepto de clase (abstracto) y de objeto (particularización), el cual puede tener una serie de atributos (propiedades) y métodos (funcionalidades).

1. Fundamentos de OOP

Una clase (*class*) es una plantilla para múltiples objetos con características similares. Las clases recogen todas las características y funcionalidades de un conjunto particular de objetos.

Una instancia (*instance*) de una clase es un objeto real. Si una clase es la representación genérica de un objeto, una instancia es su representación concreta.

Cuando se realiza un programa, en realidad se diseñan y construyen un conjunto de clases. Cuando se ejecuta el programa, se crean y se destruyen instancias de las mismas, según se van necesitando. El cometido de un programador es crear el conjunto adecuado de clases que realicen la tarea que se quiere resolver.

Cada clase que se desarrolla en JAVA va a estar formada por dos tipos de elementos:

- Atributos
- Métodos

Los atributos (*attributes*) son las propiedades individuales que diferencian un objeto de otro, y determinan la apariencia, estado, u otras características de un objeto. Los atributos se definen por medio de variables, que toman valores determinados para cada tipo de objeto. Así, se llaman también variables de instancia. Aparte están también lo que se llaman variables de clase, que se aplican a la clase en sí, y a todas las instancias de la misma.

Los métodos (*methods*) son funciones que se definen dentro de las clases. Estos pueden operar sólo sobre instancias de la clase, y se denominan métodos de instancia, o bien sobre la clase misma, y se denominan métodos de clase. Una clase u objeto puede llamar métodos de otra clase u objeto para comunicar cambios en el entorno o solicitar que un objeto cambie su estado.

Uno de los conceptos más importantes de la OOP es la *herencia*. Por medio de esta, las clases se organizan de forma jerárquica, y así cada clase tiene una superclase, y puede tener cero, una, o varias subclases. En la raiz del árbol de herencia se encuentra una clase, llamada *Object*, de la cual heredan todas las clases del sistema. Aparte de definir una estructura jerárquica, la herencia también caracteriza cada una de las clases, pues estas heredan tanto los atributos como los métodos. Una subclase es libre de modificar la funcionalidad de un método de la superclase redefiniéndolo: declararlo de nuevo. JAVA incorpora un modelo de herencia llamado *herencia simple*, mediante el cual una clase sólo puede heredar de una única clase. El modelo de C++ se llama de *herencia múltiple*, mediante el cual una clase puede heredar de varias clases simultáneamente.

2. OOP con JAVA

El formato general de una clase en JAVA es la siguiente:

```
tipo class nombre1 extends nombre2 implements nombre3, . . .
{
    atributos (clase + instancia)

    constructores
```

```
        métodos (clase + instancia)
}
```

donde *tipo* puede ser una combinación de las siguiente opciones:

- *public*: cualquier clase puede crear objetos ella.
- *protected*: sólo pueden crear objetos de esta clase los descencientes por herencia.
- *private*: sólo pueden ser accedidas desde clases definidas en el mismo fichero.
- *final*: no pueden haber subclases de esta clase.
- *abstract*: se definen una serie de métodos, que tendrán que implementar las subclases.

Si no se especifica nada, la clase es pública. El parámetro *nombre2* es el nombre de la clase de la cual se hereda. Por defecto se hereda de Object. La lista *nombre3* es la serie de interfaces que son implementados. En un fichero .java puede haber únicamente una clase pública/protegida, y todas las privadas que se deseen.

Los atributos en JAVA se declaran de la siguiente forma:

```
tipo clase nombre;
```

donde *clase* es la clase del atributo, *nombre* el nombre que tomará dicha variable, y *tipo* puede ser una combinación de las siguientes opciones:

- *public*: el atributo es visible desde cualquier clase.
- *private*: el atributo sólo es visible desde las clases herederas.
- *protected*: el atributo no es visible.
- *final*: el atributo no se puede modificar (constante).
- *static*: el atributo es una variable de clase.

Los métodos, similares a las funciones de C, se definen en JAVA de la siguiente forma:

```
tipo retorno nombre (lista_parámetros)
{
    variables locales
    instrucciones
}
```

donde *nombre* es el nombre de la función, y *tipo* puede ser una combinación de las siguientes opciones:

- *public*: el método puede ser llamado por cualquier clase.
- *protected*: el método sólo puede ser llamado por clases herederas.
- *private*: el método sólo puede se llamado desde la misma clase.
- *static*: el método es un método de clase.
- *final*: el método no se puede redefinir.
- *abstract*: el método se ha de redefinir, pues no tiene implementación.

Además *retorno* es la clase que devuelve la función, o *void* en caso de no devolver nada. La *lista_parámetros* es una lista de pares clase y variable, separados por comas, que son los parámetros de entrada a la función. Esta lista puede estar vacía.

Los constructores son los métodos que se invocan cuando se crea un objeto (*new*). Se definen de forma similar a los métodos. En los constructores es el lugar adecuado para dar valores iniciales a los atributos. Así tenemos que los

constructores en java se definen:

```
tipo  nombre_clase (lista_parámetros)
{
    variables locales

    instrucciones
}
```

donde *nombre_clase* es el nombre de la clase de la cual es constructor, *lista_parámetros* es similar al caso anterior, y *tipo* puede ser uno de los siguientes:

- *public*: se puede usar el constructor desde cualquier clase.
- *protected*: sólo pueden usar el constructor clases herederas.
- *private*: no se puede usar el constructor.

En cualquier situación donde se pueda producir ambigüedad entre el nombre de una variable parámetro y un atributo, se puede utilizar la notación *this*. De esta forma, para referirse a un atributo se le antepone el *this* de la siguiente forma:

```
this.atributo
```

Para utilizar una clase hay que seguir los siguientes pasos:

- crear el objeto
- llamar a los métodos necesarios de ese objeto
- destruir el objeto

Para crear un objeto podemos hacer:

```
Object prueba;
prueba = new Object ();
```

donde primero declaramos la variable prueba que referenciará el objeto, y posteriormente llamamos a un constructor de la clase.

Para llamar a un método, sobre el objeto previamente creado, se especifica el nombre del método:

```
prueba.miMetodoDePrueba ();
```

por último, para destruir el objeto se puede utilizar el método genérico *dispose ()*. Esto no es habitual, ya que existe un proceso, llamado Garbage Collector, que se encarga de eliminar los objetos que ya no se usan.

3. OOP avanzada con JAVA

Ya hemos visto la forma básica de trabajar con clases y métodos. Además de estas, JAVA tiene otra serie de características como son los *packages* y los *interfaces*.

Los *packages* (paquetes) son conjuntos de clases relacionadas por algún motivo, de forma que el manejo de las mismas, cuando el número de clases crece, sea un poco coherente. Así, los paquetes son en realidad directorios que contienen los ficheros de las clases, de forma jerárquica. Los paquetes se denominan de forma similar a los nombres en Internet (notación de punto), donde cada nombre es una parte del directorio. Hay un paquete imprescindible que contiene todas las clases básicas de JAVA, que se denomina java, del que cuelgan los demás paquetes como:

- java.util
- java.language
- java.awt

En general, para cada clase definida hay que especificar en qué paquete va a estar. Esto se hace siempre al principio de cada fichero en JAVA de la siguiente forma:

```
package nombre;
```

Si no se especifica, se supone que está localizado en la raiz. Posteriormente a la declaración de paquete hay que especificar que clases vamos a usar, con su paquete correspondiente de la forma:

```
import nombre_de_clase;
```

Si vamos a usar varias clases del mismo paquete, por ejemplo URL y Socket del paquete java.net, lo podemos especificar de las siguientes formas:

```
import java.net.URL;
import java.net.Socket;
```

```
import java.net.*;
```

Los *interfaces* son formas de especificar plantillas de funcionalidades. En realidad son definiciones de clases que no tienen implementación, y que otras clases usarán e implementarán. Los interfaces se declaran de forma similar a las clases, solo que los atributos son todos *public static final* y los métodos *public abstract*. Un ejemplo de declaración de interface sería:

```
public interface mi Interface {
        public static final int coche = 1;
        public static final int casa = 2;

        public abstract int miMetodo1 ();
        public abstract float miMetodo2 (int par);
}
```

Para hacer uso de un interface se utiliza la cláusula *extends* de la declaración de clases. De esa forma la clase debe de implementar todos los métodos que use de la interfaz. Esta es la forma de simular herencia múltiple mediante JAVA. La razón por la que no está incorporada en el lenguaje es por hacerlo lo más simple y robusto posible.

Tema III: Excepciones en JAVA

Las excepciones en Java están destinadas, al igual que en el resto de los lenguajes que las soportan, para la detección y corrección de errores. Si se produce un error, la aplicación no debería abortar. En su lugar debería lanzar (throw) una excepción que podríamos capturar (catch) y resolver la situación de error. Java sigue el mismo modelo de excepciones que se utiliza en C++. Utilizadas en forma adecuada, las excepciones aumentan, en gran medida, la robustez de las aplicaciones así como su mantenimiento.

1. Qué son y por qué usarlas.

Una excepción es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de instrucciones. Se utilizan, generalmente, para:

- Separar el código de manejo de errores del código regular.
- Propagar errores en la pila de llamadas.
- Agrupar tipos de errores.

1.1. Separar el código de manejo de errores del código regular.

Un ejemplo de pseudocódigo para leer un fichero en memoria podría ser:

```
readFile
{
     open the file;
     determine its size;
     allocate that much memory;
     read the file into memory;
     close the file;
}
```

Que ocurre si ...??:

- El fichero no puede ser abierto
- No se puede determinar la longitud del fichero
- No se puede obtener suficente memoria
- Si falla una lectura
- Si no se puede cerrar el fichero

Hay que tratar los posibles errores de alguna forma: el tradicional, sin usar excepciones (código no JAVA) y el moderno, usando excepciones (código JAVA).

Código no JAVA

```
errorCodeType readFile {
                   initialize errorCode = 0;
                   open the file;
                   if (theFileIsOpen) {
                       determine the length of the file;
                       if (gotTheFileLength) {
                           allocate that much memory;
                           if (gotEnoughMemory) {
                               read the file into memory;
                               if (readFailed) {
```

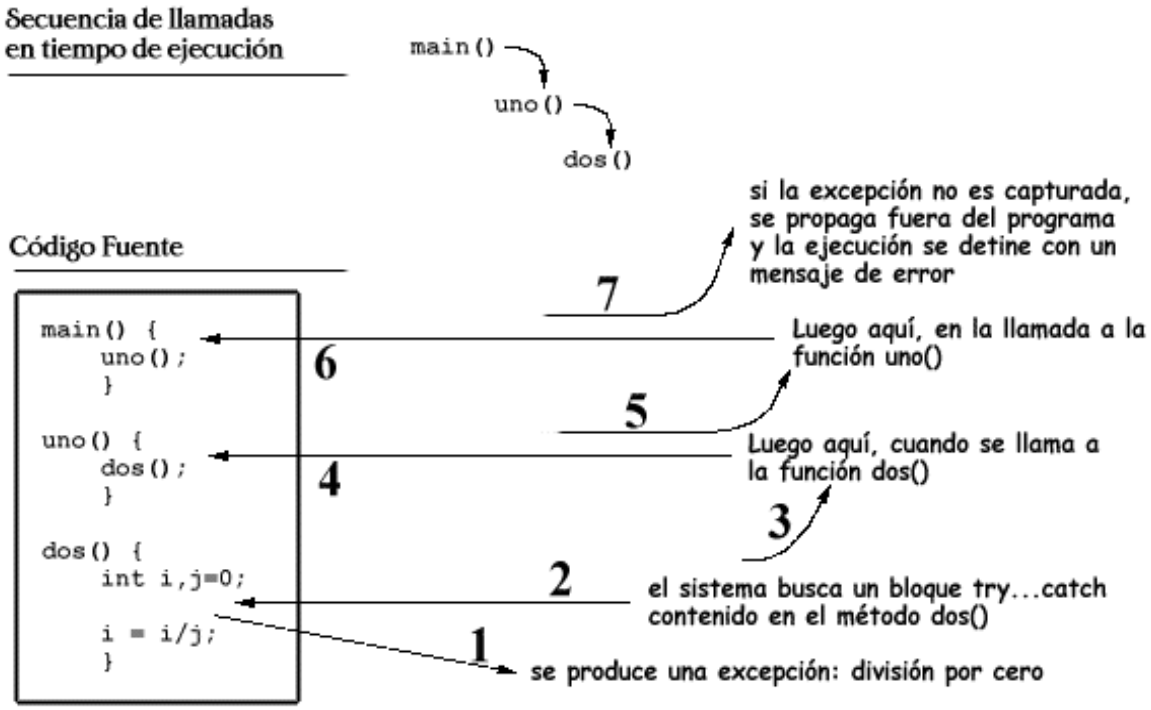
```
                                errorCode = -1;
                            }
                        } else {
                            errorCode = -2;
                        }
                    } else {
                        errorCode = -3;
                    }
                    close the file;
                    if (theFileDidntClose && errorCode == 0) {
                        errorCode = -4;
                    } else {
                        errorCode = errorCode and -4;
                    }
                } else {
                    errorCode = -5;
                }
                return errorCode;
            }
```

Código JAVA

```
readFile {
                try
                {
                    open the file;
                    determine its size;
                    allocate that much memory;
                    read the file into memory;
                    close the file;
                } catch (fileOpenFailed) {
                    doSomething;
                } catch (sizeDeterminationFailed) {
                    doSomething;
                } catch (memoryAllocationFailed) {
                    doSomething;
                } catch (readFailed) {
                    doSomething;
                } catch (fileCloseFailed) {
                    doSomething;
                }
            }
```

1.2. Propagar errores en la pila de llamadas.

En el caso de tener métodos que se llaman unos a otros, en el caso de que se produzca una excepción, esta se propaga hacia arriba hasta que se procese (catch) o bien se interrumpa el programa. Sin hacer uso de excepciones se tienen que tratar todas las posibles causas de error en cada punto conflictivo.

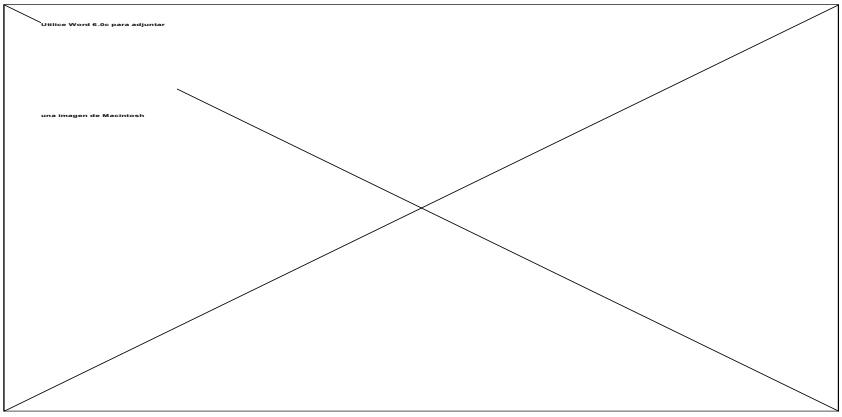


Código JAVA

```
method1 {
     try {
           call method2;
     } catch (exception) {
           doErrorProcessing;
     }
}

method2 throws exception {
     call method3;
}

method3 throws exception {
     call readFile;
}
```

1.3. Agrupar tipos de errores.

Para ayudar a la hora de tratar los errores, estos se pueden agrupar por tipos. Así, si tenemos una jerarquía de errores como sigue:



podemos tratar por separado cada tipo de error. Por ejemplo podemos querer tratar el caso de intentar acceder a una celda de un array que no exista, sin importarnos si el índice es muy grande o muy pequeño.

2. Ejemplos básicos de excepciones

Vamos a ver cual es la sintaxis de las excepciones en JAVA. Para ello tomaremos como base el siguiente ejemplo:

Código sin tratamiento de excepciones

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

class ListOfNumbers {
    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }

    public void writeList() {
        PrintStream pStr = null;
        int i;


        System.out.println("Entering try statement");
        pStr = new PrintStream(
                  new BufferedOutputStream(
                     new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));

        pStr.close();
    }
}
```

En este código, el método writeList puede ser problemático, ya que tenemos dos posibles fuentes de error:

- la apertura del fichero de salida: pStr = new PrintStream (...);
- el acceso al array: victor.elementAt (i);

Para capturar las excepciones se utiliza un bloque *try-catch-finally*. En el cuerpo de *try* se coloca el código que se quiere ejecutar, en cada bloque *catch* se especifica una clase de excepciones y el código de servicio de la misma, y en el bloque *finally* se sitúa el código que se quiere ejecutar al finalizar el conjunto *try-catch*.

```
try {
        // código
} catch ( ... ) {
        // código
} catch ( ... ) {
        // código
} finally {
        // código
}
```

De esta forma, el código original modificado para tratar excepciones nos quedaría:

Código con tratamiento de excepciones

```
public void writeList() {
              PrintStream pStr = null;

              try {
                int i;

                System.out.println("Entering try statement");
                pStr = new PrintStream(
                          new BufferedOutputStream(
                           new FileOutputStream("OutFile.txt")));

                for (i = 0; i < size; i++)
                  pStr.println("Value at: " + i + " = " + victor.elementAt(i));
              } catch (ArrayIndexOutOfBoundsException e) {
                  System.err.println("Excepción: " + e.getMessage());
              } catch (IOException e) {
                  System.err.println("Excepción: " + e.getMessage());
              } finally {
                  if (pStr != null) {
                      System.out.println("Closing PrintStream");
                      pStr.close();
                  } else {
                      System.out.println("PrintStream not open");
                  }
              }
            }
```

Si se ejecuta este código, se pueden tener tres ejecuciones distintas: que se produzca una *IOException*, un *ArrayIndexOutOfBoundsException*, o bien que no se produzca ningún error. La salida en pantalla sería la siguiente:

IOException

```
Entering try statement
Caught IOException:
OutFile.txt
PrintStream not open
```

ArrayIndexOutOfBoundsException

```
Entering try statement
Caught ArrayIndexOutOfBoundsException: 10 >= 10
Closing PrintStream
```

Sin errores

```
Entering try
statement
Closing PrintStream
```

Para especificar que excepciones pasa un método, se utiliza la cláusula *throws*. De esta forma se pasa a otro manejador de excepciones situado superiormente en la pila de llamadas. Así se podría especificar un código que pasa excepciones

de un determinado tipo de la siguiente forma:

Código pasando excepciones

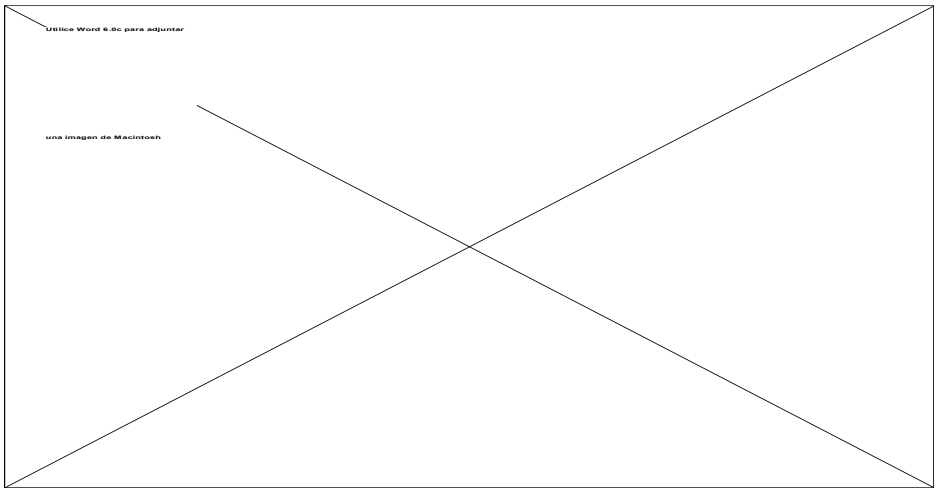
```
public method1 () {
     try {
           method2 ();
     } catch (IOException e) {
           doErrorProcessing ();
     }
}

public method2 () throws IOException {
     method3 ();
}

public method3 () throws  IOException {
     readFile ();
}
```

Un método puede generar sus propias excepciones, haciendo uso de la cláusula *throw*. De esta forma, por ejemplo, si creamos una pila, en la operación de extracción podemos chequear si está vacía, y en su caso notificar el error. Esto se puede ver con el siguiente código:

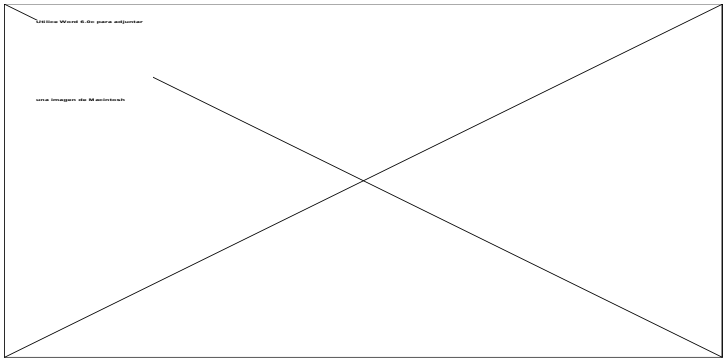
Código generando excepciones

```
public Object pop() throws EmptyStackException {
         Object obj;

          if (size == 0)
                    throw new EmptyStackException();

           obj = objectAt(size - 1);
           setObjectAt(size - 1, null);
           size--;
          return obj;
}
```

La librería básica de clases en JAVA contiene una serie de tipos de excepciones predefinidas, que heredan de la clase Throwable.



Además de las excepciones predefinidas, podemos crear una jerarquía completamente nueva, o bien ampliar la existente.



Tema IV: MultiThreading en Java

Un *thread*, también llamado contexto de ejecución o proceso ligero, es un flujo secuencial de instrucciones dentro de un proceso. Así, un proceso podrá estar formado por distintos threads que se ejecutan en paralelo. En general, en JAVA tendremos un único proceso formado por diferentes threads. La capacidad de multithreading está cada vez más extendida entre los sistemas operativos. La implementación de JAVA varía mucho de un sistema a otro, por lo que la especificación de los threads suele ser la parte más complicada en un programa JAVA.

1. Declaración de threads

El siguiente ejemplo muestra una ejecución típica de un programa en JAVA que está formado por dos threads ("Fiji" y "Jamaica"). Se puede observar como evolucionan ambos threads, observando la salida en pantalla.

Programa con threads

```
class TwoThreadsTest {
        public static void main (String[] args) {
            new SimpleThread ("Jamaica").start ();
            new SimpleThread ("Fiji").start ();
        }
}


class SimpleThread extends Thread {
        public SimpleThread (String str) {
            super(str);
        }

        public void run() {
            for (int i = 0; i < 10; i++)
                 system.out.println (i + name);
            system.out.println ("DONE! " + name);
        }
 }
```

Salida Pantalla

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

Se pueden tener threads de distintas maneras. La más simple es heredando de la clase Thread, y redefiniendo el método *run ()*, que es el que se llama siempre cuando se ejecuta un thread. Entre otros, se tienen los siguientes constructores de la clase Thread:

- Thread ()
- Thread (String)
- Thread (Runnable)
- Thread (Runnable, String)

La forma general de declarar un thread es la siguiente:

```
class nuestroThread extends Thread {
       public void run () {
             // Hacer lo que sea ...
        }
}
```

Para arrancar el thread desde otra clase, simplemente se ejecuta:

```
new nuestroThread ().start ();
```

Otro método para declarar threads (generalmente el más común) consiste en crear una clase que implemente la interfaz Runnable y crear un metodo *run ()* de forma similar al caso anterior. Esta es la forma de implementarlo como herencia múltiple.

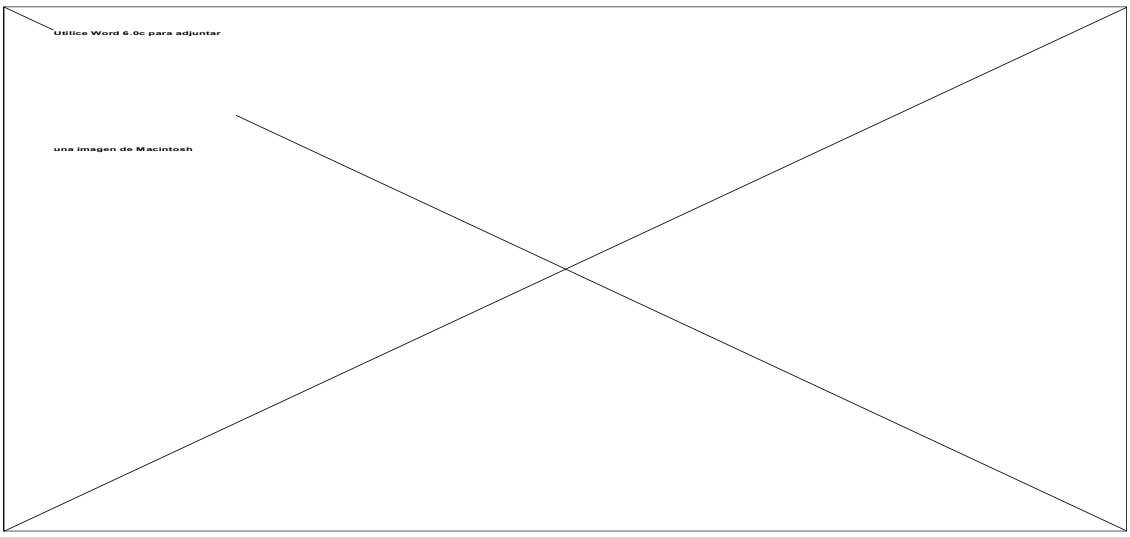
```
class nuestroThread extends MiClase implements Runnable {
      public void run() {
                // Hacer lo que sea ...
           }
 }
```

Para arrancar el thread desde otra clase, simplemente se ejecuta:

```
Thread miThread;

miThread = new Thread (nuestroThread);
miThread.start ();
```

2. Ejecución de threads

Para que el sistema pueda ejecutar todos los threads en paralelo, se sigue un algoritmo de planificación similar a los que emplean los sistemas operativos. De esta forma, los threads se ejecutan durante pequeños periodos de tiempos de forma secuencial, y así parecer que se ejecutan todos en paralelo. Así, un thread, en un instante dado, puede estar en uno de los siguientes estados:



El algoritmo de planificación es de prioridad fija (determinístico). En cualquier momento, si hay múltiples threads ejecutándose, el runtime de JAVA escoge el thread (en estado *runnable*) de mayor prioridad. Si tienen la misma prioridad, se utiliza un esquema de round-robin. Así, un thread se ejecutará hasta que:

- Otro thread de mayor prioridad pase al estado *runnable*.
- Este thread llega al final del *run ()*, o cede la ejecución con *yield ()*.
- En sistemas con time-slicing, cuando acaba su quantum asignado.

Cuando se crea un thread, este hereda la prioridad del thread que lo creó. Se puede cambiar su prioridad usando *setPriority (int)*. El rango de valores que puede tener la prioridad va desde Thread.MIN_PRIORITY hasta Thread.MAX_PRIORITY.

Un ejemplo de cómo afectan las prioridades y el sistema de asignación de quanta, tenemos el siguiente programa que crea una serie de threads que simplemente se dedican a contar:

```
class SelfishRunner extends Thread
{
       public int tick = 1;
       public int num;

       SelfishRunner (int num) {
            this.num = num;
       }

       public void run () {
            while (tick < 400000) {
                  tick++;
                  if ((tick % 50000) == 0) {
                     System.out.println("Thread #" + num + ", tick=" + tick);
                   }
            }
       }
}

class RaceTest {
        final static int NUMRUNNERS = 2;
        public static void main(String[] args) {
              SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];

              for (int i = 0; i < NUMRUNNERS; i++) {
                      runners[i] = new SelfishRunner(i);
                      runners[i].setPriority(2);
              }
              for (int i = 0; i < NUMRUNNERS; i++) {
                      runners[i].start();
              }
        }
}
```

Este programa se ejecutará con dos configuraciones distintas:

- Ejemplo 1: los dos threads con la misma prioridad y time-slicing
- Ejemplo 2: el thread número 0 con prioridad 2 y el número 1 con prioridad 3

Ejemplo 1

```
Thread #0, tick = 50000
Thread #1, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #0, tick = 150000
Thread #1, tick = 150000
Thread #0, tick = 200000
Thread #1, tick = 200000
Thread #0, tick = 250000
Thread #1, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #0, tick = 350000
```

Ejemplo 2

```
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #0, tick = 50000
Thread #1, tick = 350000
Thread #1, tick = 400000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
```

```
Thread #1, tick = 350000          Thread #0, tick = 300000
Thread #0, tick = 400000          Thread #0, tick = 350000
Thread #1, tick = 400000          Thread #0, tick = 400000
```

Si, independientemente del sistema que tengamos, queremos hacer que nuestros threads implemente como mínimo una multitarea colaborativa, se puede utilizar *yield ()*, que detiene el thread en ese justo instante. Hay que notar que esto sólo funciona para procesos de igual prioridad. Por ejemplo tenemos el siguiente código:

```
class PoliteRunner extends Thread {
       public int tick = 1;
       public int num;

       PoliteRunner (int num) {
               this.num = num;
       }

       public void run () {
               while (tick < 400000) {
                  tick++;
                  if ((tick % 50000) == 0) {
                     System.out.println (" Thread #" + num + ", tick=" + tick);
                     yield();
                  }
               }
       }
}

class RaceTest2 {
        final static int NUMRUNNERS=2;

        public static void main (String[] args) {
                 PoliteRunner[] runners=new PoliteRunner[NUMRUNNERS];

                 for (int i=0; i < NUMRUNNERS; i++) {
                          runners[i]=new PoliteRunner(i);
                          runners[i].setPriority(2);
                 }
                 for (int i=0; i < NUMRUNNERS; i++) {
                          runners[i].start();
                 }
        }
}
```

Hay un tipo de threads especiales que corren en segundo plano y dan servicios al resto de threads en ejecución que se denominan *daemon threads* (threads demonio). Cuando sólo quedan daemon threads en ejecución, el interprete termina pues no quedan threads a los que darles servicio. Siempre hay threads de este tipo en ejecución, generalmente del sistema. Entre estos están:

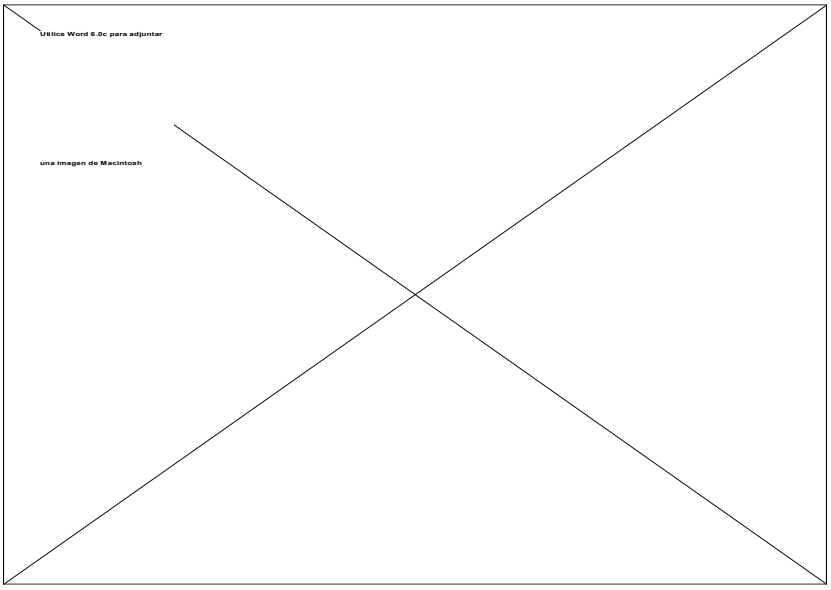
- Garbage collector
- Image Fetcher

Para trabajar con este tipo de threads tenemos los siguientes métodos:

- *setDaemon (boolean)*: convierte el thread a demonio
- *isDaemon ()*: consulta si el thread es un demonio

3. Grupos de threads

Cada thread en JAVA es miembro de un *thread group*. Estos grupos crean un esquema de agrupamientp de threads en un solo objeto, facilitando la manipulacion de estos de forma flexible y unificada.



Así, si queremos crear un grupo de threads, simplemente le damos un nombre y creamos una instancia de la clase ThreadGroup.

```
ThreadGroup miGrupo;
miGrupo = new ThreadGroup ("Nombre del grupo");
```

A partir de aquí, sobre ese grupo podremos realizar una serie de acciones, que se propagarán por todos los threads que formen el grupo:

- *setDaemon (boolean)*: establece el grupo como daemon
- *setMaxPriority (int)*: cambia la prioridad del grupo
- *stop ()*: termina la ejecución del grupo
- *suspend ()*: detiene la ejecución del grupo
- *resume ()*: continúa la ejecución del grupo
- *destroy ()*: elimina el grupo

Para asignar un thread a un grupo previamente creado, se añade la información del grupo en el constructor del thread. Por ejemplo:

```
Thread miThread;
miThread = new Thread (miGrupo, "Nombre del thread");
```

Así, para asignar threads a grupos tenemos los siguientes constructores:

- Thread (ThreadGroup, Runnable)
- Thread (ThreadGroup, String)
- Thread (ThreadGroup, Runnable, String)

Un ejemplo de programa para ver la lista de los threads que están en el mismo grupo sería el siguiente:

```
class EnumerateTest {
          void listCurrentThreads() {
              ThreadGroup currentGroup = Thread.currentThread ().getThreadGroup ();
              int numThreads;
              Thread[] listOfThreads;

              numThreads = currentGroup.activeCount ();
              listOfThreads = new Thread[numThreads];
              currentGroup.enumerate (listOfThreads);
              for (int i = 0; i < numThreads; i++) {
                    System.out.println ("Thread #" + i + "=" +
                        listOfThreads[i].getName ());
                }
        }
}
```

4. Comunicación entre threads

Cuando dos o más threads en ejecución necesitan intercambiar información lo pueden hacer de distintas formas. El método más simple es por medio de una variable compartida. Así, podemos tener los siguientes métodos:

- Comunicación no monitorizada
- Comunicación monitorizada

Los problemas básicos de comunicación entre threads se suelen estudiar por medio del problema del productor y del consumidor. En este caso un thread irá produciendo información y otro irá procesando esa información. Así tendremos los siguientes elementos:

- Productor: el thread ProduceInteger
- Consumidor: el thread ConsumeInteger
- Variable compartida: de la clase HoldInteger

La diferencia entre los distintos sistemas de comunicación están en la implementación de la variable compartida. La parte común del programa es la siguiente:

```
public class SharedCell {
   public static void main( String args[] )
   {
      HoldInteger h = new HoldInteger ();
      ProduceInteger p = new ProduceInteger (h);
      ConsumeInteger c = new ConsumeInteger (h);

      p.start();
      c.start();
   }
}

class ProduceInteger extends Thread {
   private HoldInteger pHold;

   public ProduceInteger( HoldInteger h )
   {
      pHold = h;
   }

   public void run()
   {
```

```
            for ( int count = 0; count < 10; count++ ) {
                 pHold.setSharedInt (count);
                 System.out.println("Producer set sharedInt to " + count);
                 // sleep for a random interval
                 try {
                       sleep( (int) ( Math.random() * 3000 ) );
                 } catch( InterruptedException e ) {
                       System.err.println ("Exception " + e.toString());
                 }
            }
         }
      }

      class ConsumeInteger extends Thread {
         private HoldInteger cHold;

         public ConsumeInteger (HoldInteger h ) {
              cHold = h;
         }

          public void run() {
              int val;

              val=cHold.getSharedInt();
              System.out.println( "Consumer retrieved " + val );
              while (val != 9) {
                    // sleep for a random interval
                    try {
                          sleep( (int) ( Math.random() * 3000 ) );
                    } catch( InterruptedException e ) {
                          System.err.println( "Exception " + e.toString() );
                    }
                    val = cHold.getSharedInt ();
                    System.out.println( "Consumer retrieved " + val );
              }
          }
      }
```

De esta forma la clase HoldInteger tiene un atributo que es la información que se comunican. En el caso de comunicación no monitorizada tenemos la siguiente ejecución:

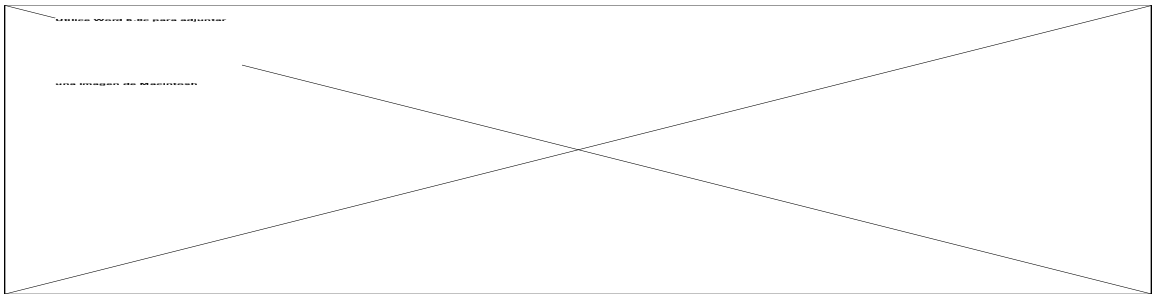
ComunicaciÛn no moritorizada

```
class HoldInteger {
    private int sharedInt;

    public void setSharedInt (int val) {
          sharedInt = val;
    }

    public int getSharedInt () {
          return sharedInt;
    }
}
```

Salida pantalla

```
Producer set sharedInt to 0
Consumer retrieved 0
Producer set sharedInt to 1
Consumer retrieved 1
Consumer retrieved 1
Producer set sharedInt to 2
Producer set sharedInt to 3
Producer set sharedInt to 4
Consumer retrieved 4
Producer set sharedInt to 5
Consumer retrieved 5
Producer set sharedInt to 6
Consumer retrieved 6
Producer set sharedInt to 7
Producer set sharedInt to 8
Producer set sharedInt to 9
Consumer retrieved 9
```

Para el caso de comunicación monitorizada tenemos el siguiente modelo:



El monitor se encarga de sincronizar los dos procesos, de forma que no se pierda información durante la ejecución. La ejecución del caso de comunicación monitorizada se muestra a continuación:

ComunicaciÛn monitorizada

```
class HoldInteger {
   private int sharedInt;
   private boolean writeable = true;

   public synchronized void setSharedInt (int val)
   {
      while (!writeable) {
         try {
           wait ();
         }
         catch  (InterruptedException e) {
           System.err.println("Exception: " +
               e.toString() );
         }
      }
      sharedInt = val;
      writeable = false;
      notify();
   }

   public synchronized int getSharedInt ()
   {
      while (writeable) {
         try {
            wait();
         }
         catch (InterruptedException e) {
            System.err.println( "Exception: " +
                e.toString() );
         }
      }
      writeable = true;
      notify ();
      return sharedInt;
   }
}
```

Salida pantalla

```
Producer set sharedInt to 0
Consumer retrieved 0
Producer set sharedInt to 1
Consumer retrieved 1
Producer set sharedInt to 2
Consumer retrieved 2
Producer set sharedInt to 3
Consumer retrieved 3
Producer set sharedInt to 4
Consumer retrieved 4
Producer set sharedInt to 5
Consumer retrieved 5
Producer set sharedInt to 6
Consumer retrieved 6
Producer set sharedInt to 7
Consumer retrieved 7
Producer set sharedInt to 8
Consumer retrieved 8
Producer set sharedInt to 9
Consumer retrieved 9
```

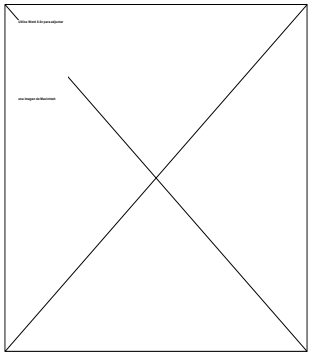
Se pueden dar casos más complejos de comunicación, que se resuelven con técnicas de compartción de recursos: semáforos, bloqueos, etc. El lenguaje JAVA posee los mecanismos básicos para implementar estas técnicas, la sincronización. Así podemos declarar *synchronized* tanto métodos como variables. El sistema asegura que la operación (variable) o las operaciones (método) se ejecutan de forma atómica.

Tema V: Comunicaciones en JAVA

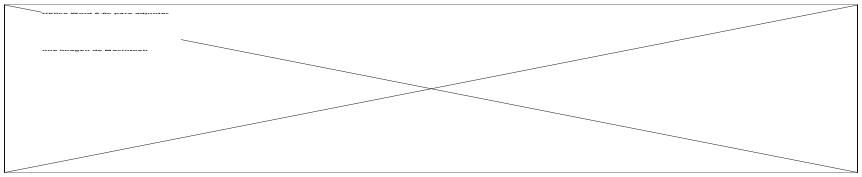
Las clases básicas del sistema de JAVA nos permiten realizar todo tipo de comunicaciones TCP/IP. Para hacer independiente el uso de las mismas de la implementación que haga el sistema operativo, se utiliza el modelo ampliamente difundido de sockets. Además, JAVA incorpora mecanismos para trabajar con protocolos recientes tales como URLs de HTTP, etc.

1. Comunicaciones TCP/IP

A nivel general, en TCP/IP, podemos encontrar cinco niveles de comunicaciones: *físico*, *enlace*, *red*, *transporte* y *aplicación* (los demás no están soportados por el modelo TCP/IP). Al escribir programas Java no nos preocuparemos de los niveles inferiores, y generalmente la programación se realizará a nivel de *aplicación*.



Las clases del paquete java.net nos proporcionan capacidades de programación en red de forma independiente al sistema operativo basandose principalmente en los *sockets* como primitiva de comunicación. Cuando dos aplicaciones se comunican, primero establecen una conexión, y a través de la misma envían y reciben información. Los sockets son puntos finales de enlace de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets.



Sockets Stream (*TCP*, Transport Control Protocol)

Son un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados. El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones. Por ejemplo, se tienen los siguientes protocolos TCP:

- Telnet
- HTTP
- SMTP

- Rlogin
- FTP

Sockets Datagrama (*UDP*, User Datagram Protocol)

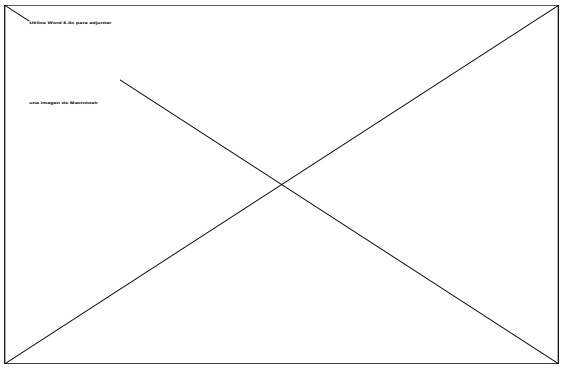
Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió. El protocolo de comunicaciones con datagramas es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación. Por ejemplo se tienen los siguientes protocolos UDP:

- Ping
- NTP
- SNMP
- Multicast
- Streams Audio
- Streams Vídeo

Sockets Raw

Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos. Por ejemplo se tiene:

- ICMP



2. Trabajando con URLs en JAVA

Una *URL* (Uniform Resource Locator) es una referencia o dirección útil para localizar un recurso en Internet. Por ejemplo tenemos: *http://java.sun.com*. Cada URL está formada por dos componentes importantes : el identificador de protocolo y el nombre del recurso. En este ejemplo el identificador de protocolo es *http* (Hyper Text Transfer Protocol), que es el usado por los navegadores. Otros protocolos contemplados son *ftp* (File Transfer Protocol), *gopher* (Gopher), *file* (ficheros locales) y *news* (News Protocol). El nombre del recurso corresponde a la dirección completa del mismo, y en este caso es java.sun.com. El formato del nombre del recurso depende del protocolo utilizado pero en la mayoría existen los siguientes componentes: nombre del servidor, nombre del fichero, número de puerto, y referencia

(localización dentro de un fichero), si bien son más utilizados para el protocolo HTTP. Así, en este caso tenemos nombre de host: java.sun.com, nombre del fichero: /, número de puerto: 80 y referencia: vacía.

Para trabajar en JAVA con URLs, simplemente hacemos uso de la clase URL, teniendo en cuenta de capturar o lanzar la excepción *MalformedURLException*. Sobre URLs tenemos los siguientes métodos:

- *getProtocol ()*: devuelve el protocolo de la URL
- *getHost ()*: devuelve el host de la URL
- *getPort ()*: devuelve el puerto de la URL
- *getFile ()*: devuelve el fichero de la URL
- *getRef ()*: devuelve la referencia de la URL

Un ejemple de creación, uso y conexión a una URL es el siguiente:

```
import java.net.*;
import java.io.*;

class LeeURL {
       public static void main(String [] args) {
           try {
               URL um = new URL("http://www.um.es/");
               DataInputStream dis = new DataInputStream (um.openStream());
               String inputLine;

               while ((inputLine = dis.readLine ()) != null) {
                   System.out.println (inputLine);
               }
               dis.close();
           } catch (MalformedURLException me) {
               System.out.println ("MalformedURLException: " + me);
           } catch (IOException ioe) {
               System.out.println ("IOException: " + ioe);
           }
       }
  }
```

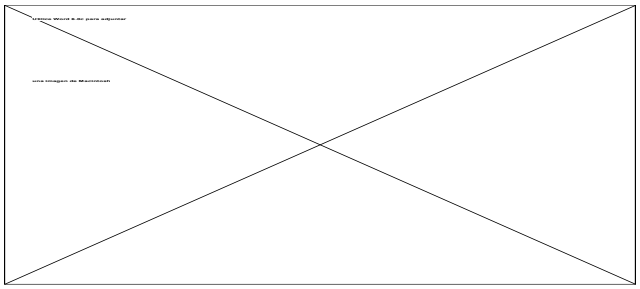
Si queremos comunicarnos con la URL, se ha de usar la clase *URLConnection* que permite establecer una conexión con una URL especificada, ya sea para leer o escribir sobre ella. Por ejemplo se tiene:

```
import java.io.*;
import java.net.*;

public class InvierteCadenaURL  {
    public static void main (String[] args) {
        try {
            if (args.length != 1) {
                System.err.println ("Uso:  InvierteCadena cadena_a_invertir");
                System.exit (1);
            }
            String stringToReverse = URLEncoder.encode(args[0]);

            URL url = new URL ("http://java.sun.com/cgi-bin/backwards");
            URLConnection connection = url.openConnection();

            PrintStream outStream;
            outStream = new PrintStream (connection.getOutputStream());
            outStream.println ("string=" + stringToReverse);
            outStream.close ();

            DataInputStream inStream;
            inStream = new DataInputStream (connection.getInputStream());
```

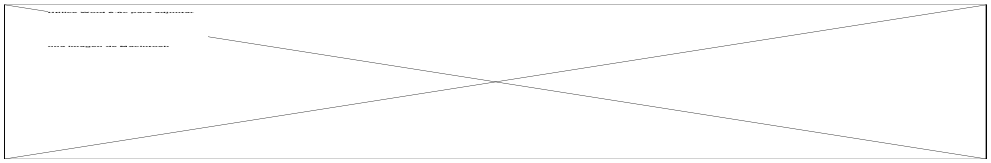
```
                String inputLine;

                while ((inputLine = inStream.readLine()) != null) {
                    System.out.println(inputLine);
                }
                inStream.close ();
            } catch (MalformedURLException me) {
                System.err.println ("MalformedURLException: " + me);
            } catch (IOException ioe) {
                System.err.println ("IOException: " + ioe);
            }
        }
    }
```

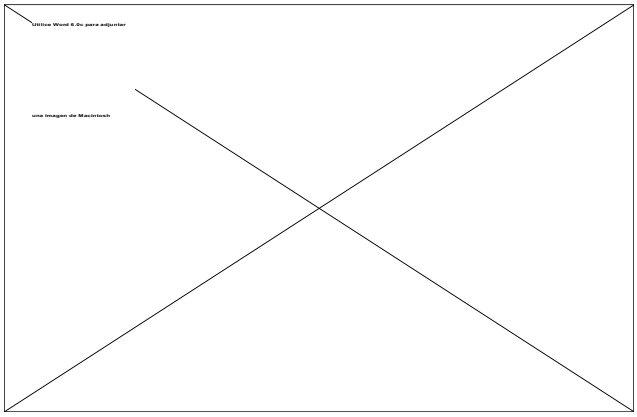
3. Trabajando con Sockets en JAVA

Como se vió anteriormente, un *socket* es un punto de comunicación bidireccional entre programas funcionando en lugares distintos en la red. En general, en las arquitecturas *cliente-servidor* existe un servidor que proporciona un servicio y unos clientes que lo utilizan. Analogamente existirán sockets servidores y sockets de cliente representados por las clases de JAVA *SocketServer* y *Socket*, situadas en el paquete java.net.



La comunicación bidireccional se implementa en Java utilizando los flujos de entrada y salida (InputStream y OutputStream) por lo que se consigue manejar de manera uniforme el acceso a ficheros y a conexiones remotas. Así, para cada socket podremos acceder a los flujos de entrada con *getInputStream ()* y a los de salida con *getOutputStream ()*.





Para abrir un socket cliente sobre un determinado servidor, utilizaremos el siguiente código:

```
Socket miCliente;
  try {
       miCliente = new Socket ("servidor", numeroPuerto);
 } catch (IOException e) {
         System.out.println (e);
     }
```

Para abrir un socket servidor, al que puedan acceder los cliente, utilizaremos el siguiente código:

```
Socket miServicio;
try {
    miServicio = new ServerSocket (numeroPuerto);
} catch (IOException e) {
    System.out.println (e);
}


Socket socketServicio = null;
try {
        socketServicio = miServicio.accept ();
} catch (IOException e) {
        System.out.println (e);
 }
```

Para crear un stream de entrada sobre un socket previamente abierto, utilizaremos el siguiente código:

```
DataInputStream entrada;
try {
     entrada = new DataInputStream (miCliente.getInputStream ());
 } catch (IOException e) {
         System.out.println (e);
 }
```

Para crear un stream de salida sobre un socket previamente abierto, utilizaremos el siguiente código:

```
DataOutputStream salida;
try {
      salida = new DataOutputStream (miCliente.getOutputStream ());
} catch (IOException e) {
      System.out.println (e);
}
```

4. Implementación de servidores: servicio de ECO

El *servicio de eco* consiste en un servidor que acepta conexiones, y todo lo que recibe por su entrada lo replica por su salida. Así, un cliente le envía datos y recibe los mismos. Es el ejemplo más sencillo de programa cliente-servidor. Así, el programa cliente sería el siguiente:

Servicio de ECO: cliente

```
import java.net.*;
import java.io.*;

class ClienteECO {
    public static void main( String args[] ) {
         Socket miCliente;
         DataInputStream entrada;
         DataOutputStream salida;

         try {
                    miCliente = new Socket ("localhost", 9999);
                    entrada = new DataInputStream(miCliente.getInputStream());
                    salida = new DataOutputStream( miCliente.getOutputStream() );

                    salida.writeBytes(Hola \n");                 // Envío datos
                    System.out.println(entrada.readLine());      // Recibo el eco

                    entrada.close();
                    salida.close();
           } catch( IOException e ) {
                     System.out.println( e );
           }
       }
}
```

El servidor de ECO se puede implementar de varias formas. La primera y más simple es como un servidor en un sólo thread. De esta forma, cuando llega una petición de conexión el servidor la atiende, y no acepta ninguna nueva hasta que esta se termina. El programa servidor sería:

Servicio de ECO: servidor mono-thread

```
import java.net.*;
import java.io.*;

class ServidorEco {
    public static void main( String args[] ) {
        ServerSocket s = null;
        Socket cliente = null;

        // Establecemos el servicio  en el puerto 9999
        // No podemos elegir un puerto por debajo del 1023 si no somos
        // usuarios con los máximos privilegios (root)
        try {
            s = new ServerSocket( 9999 );
        } catch( IOException e ) {
            System.out.println( e );
            }

        // Creamos el objeto desde el cual atenderemos y aceptaremos
        // las conexiones de los clientes y abrimos los canales de
        // comunicación de entrada y salida
        while (true) {
           try {
                cliente = s.accept();
               manejaPeticion(cliente);
           } catch( IOException e ) {
```

```
                System.out.println( e );
            }
         }
    }

    public static void manejaPeticion (Socket s) throws IOException
    {
       DataInputStream sIn;
       PrintStream sOut;

       sIn = new DataInputStream (s.getInputStream() );
       sOut = new PrintStream (s.getOutputStream());
       String  texto = sIn.readLine ();                    // Recibo datos
       sOut.println (texto);                               // Replico datos

       sIn.close();
       sOut.close();
       s.close();
  }
}
```

Si el servidor se implementa como un sólo thread, si varios clientes envían una petición de forma simultánea, sólo se atenderá la primera que se reciba. Una forma de evitar que un cliente acapare el servidor es implementarlo con múltiples threads, uno para cada petición aceptada, mas uno para el proceso de aceptar conexiones. El programa servidor sería:

Servicio de ECO: servidor multi-thread

```
import java.net.*;
import java.io.*;

class ServidorEcoThread {
    public static void main( String args[] ) {
        ServerSocket s = null;
        Socket cliente = null;

        // Establecemos el servicio  en el puerto 9999
        // No podemos elegir un puerto por debajo del 1023 si no somos
        // usuarios con los máximos privilegios (root)
        try {
            s = new ServerSocket( 9999 );
        } catch( IOException e ) {
            System.out.println( e );
        }

        // Creamos el objeto desde el cual atenderemos y aceptaremos
        // las conexiones de los clientes y abrimos los canales de
        // comunicación de entrada y salida
        while (true) {
           try {
               cliente = s.accept();
               new GestorPeticion(cliente).start();
           } catch( IOException e ) {
                System.out.println( e );
           }
        }
    }
}

class GestorPeticion extends Thread {
      Socket s;

      public GestorPeticion (Socket s ) {
         this.s = s;
      }
```

```
        public void run() {
           DataInputStream sIn;
           PrintStream sOut;

           try {
               sIn = new DataInputStream(s.getInputStream() );
               sOut = new PrintStream(s.getOutputStream());
               String  texto = sIn.readLine();                  // Recibo datos
               sOut.println( texto );                            // Replico datos

               sIn.close();
               sOut.close();
               s.close();
           } catch( IOException e ) {
                    System.out.println( e );
           }
       }
   }
```

Tema VI: Ejemplo de un chat en JAVA

Vamos a desarrollar un ejemplo relativamente complejo donde se traten todos los aspectos vistos a lo largo del curso. Para ello vamos a construir una aplicación cliente-servidor que implemente un servicio de chat: una serie de usuarios (clientes) se conectan al chat (servidor) de forma que cada información transmitida por un usuario le llega a todos los demás.

1. Servidor de chat

El servidor se implementará haciendo uso del multi-threading, para no limitar el número máximo de usuarios que se conecten al servicio. El servidor será el siguiente:

Servidor de chat: clase ChatServer

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

public class  ChatServer  {
  public ChatServer (int port) throws IOException {
    ServerSocket server = new ServerSocket (port);
    while (true) {
      Socket client = server.accept ();
      System.out.println ("Accepted from " + client.getInetAddress ());
      ChatHandler c = new ChatHandler (client);
      c.start ();
    }
  }

  public static void main (String args[]) throws IOException {
    if (args.length != 1)
      throw new RuntimeException ("Sintaxis: ChatServer puerto”);
    new ChatServer (Integer.parseInt (args[0]));
  }
}
```

Servidor de chat: clase ChatHandler

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

public class  ChatHandler  extends Thread {
  protected Socket s;
  protected DataInputStream i;
  protected DataOutputStream o;

  public ChatHandler (Socket s) throws IOException {
    this.s = s;
    i = new DataInputStream (new BufferedInputStream (s.getInputStream ()));
    o = new DataOutputStream (new BufferedOutputStream (s.getOutputStream ()));
  }

  protected static Vector handlers = new Vector ();

  p ublic void run () {
    String name = s.getInetAddress ().toString ();
    try {
      broadcast (name + " has joined.");
```

```
      handlers.addElement (this);
      while (true) {
        String msg = i.readUTF ();
        broadcast (name + " - " + msg);
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    } finally {
      handlers.removeElement (this);
      broadcast (name + " has left.");
      try {
        s.close ();
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }

  protected static void broadcast (String message) {
    synchronized (handlers) {
      Enumeration e = handlers.elements ();
      while (e.hasMoreElements ()) {
        ChatHandler c = (ChatHandler) e.nextElement ();
        try {
          synchronized (c.o) {
            c.o.writeUTF (message);
          }
          c.o.flush ();
        } catch (IOException ex) {
          c.stop ();
        }
      }
    }
  }
}
```

2. Cliente de chat

El programa cliente se ha implementado tanto en forma de aplicación como en forma de applet. Como se puede observar, en general, la mayor parte del código es similar.

Cliente de chat: aplicaciÛn

```
/**
* ChatClient.java  1.00 98/10/05 Humberto Martínez Barberá
*
* Basado en ChatApplet de Merlin Hughes
*/

import java.net.*;
import java.io.*;
import java.awt.*;

public class   ChatClient  extends Frame implements Runnable {
  protected DataInputStream i;
  protected DataOutputStream o;

  protected TextArea output;
  protected TextField input;

  protected Thread listener;

  public ChatClient (String title, InputStream i, OutputStream o) {
```

```
    super (title);
    this.i = new DataInputStream (new BufferedInputStream (i));
    this.o = new DataOutputStream (new BufferedOutputStream (o));
    setLayout (new BorderLayout ());
    add ("Center", output = new TextArea ());
    output.setEditable (false);
    add ("South", input = new TextField ());
    pack ();
    show ();
    input.requestFocus ();
    listener = new Thread (this);
    listener.start ();
  }

  public void run () {
    try {
      while (true) {
        String line = i.readUTF ();
        output.appendText (line + "\n");
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    } finally {
      listener = null;
      input.hide ();
      validate ();
      try {
        o.close ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
  }

  public boolean handleEvent (Event e) {
    if ((e.target == input) && (e.id == Event.ACTION_EVENT)) {
      try {
        o.writeUTF ((String) e.arg);
            o.flush ();
      } catch (IOException ex) {
        ex.printStackTrace();
        listener.stop ();
      }
      input.setText ("");
      return true;
    } else if ((e.target == this) && (e.id == Event.WINDOW_DESTROY)) {
      if (listener != null)
        listener.stop ();
      hide ();
      return true;
    }
    return super.handleEvent (e);
  }

  public static void main (String args[]) throws IOException {
    if (args.length != 2)
      throw new RuntimeException ("Sintaxis: ChatClient host puerto");

    Socket s = new Socket (args[0], Integer.parseInt (args[1]));
    new ChatClient ("Chat " + args[0] + ":" + args[1],
                    s.getInputStream (), s.getOutputStream ());
  }
}
```

Cliente de chat: applet

```
/**
* ChatApplet.java  1.00 96/11/01 Merlin Hughes
*
* Copyright (c) 1996 Prominence Dot Com, Inc. All Rights Reserved.
*
* Permission to use, copy, modify, and distribute this software
* for non-commercial purposes and without fee is hereby granted
* provided that this copyright notice appears in all copies.
*
* http://prominence.com/                  merlin@prominence.com
*/

import java.net.*;
import java.io.*;
import java.awt.*;
import java.applet.*;

// Applet parameters:
//   host = host name
//   port = host port

public class ChatApplet extends Applet implements Runnable {
  protected DataInputStream i;
  protected DataOutputStream o;

  protected TextArea output;
  protected TextField input;

  protected Thread listener;

  public void init () {
    setLayout (new BorderLayout ());
    add ("Center", output = new TextArea ());
    output.setEditable (false);
    add ("South", input = new TextField ());
    input.setEditable (false);
  }

  public void start () {
    listener = new Thread (this);
    listener.start ();
  }

  public void stop () {
    if (listener != null)
      listener.stop ();
    listener = null;
  }

  public void run () {
    try {
      String host = getParameter ("host");
      if (host == null) host = getCodeBase ().getHost ();
      String port = getParameter ("port");
      if (port == null) port = "9830";
      output.appendText ("Connecting to " + host + ":" + port + "...");
      Socket s = new Socket (host, Integer.parseInt (port));
      i = new DataInputStream (new BufferedInputStream(s.getInputStream()));
      o = new DataOutputStream (new BufferedOutputStream(s.getOutputStream()));
      output.appendText (" connected.\n");
      input.setEditable (true);
      input.requestFocus ();
      execute ();
    } catch (IOException ex) {
      ByteArrayOutputStream out = new ByteArrayOutputStream ();
      ex.printStackTrace (new PrintStream (out));
      output.appendText ("\n" + out);
```

```
    }
  }

  public void execute () {
    try {
      while (true) {
        String line = i.readUTF ();
        output.appendText (line + "\n");
      }
    } catch (IOException ex) {
      ByteArrayOutputStream out = new ByteArrayOutputStream ();
      ex.printStackTrace (new PrintStream (out));
      output.appendText (out.toString ());
    } finally {
      listener = null;
      input.hide ();
      validate ();
      try {
        o.close ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
  }

  public boolean handleEvent (Event e) {
    if ((e.target == input) && (e.id == Event.ACTION_EVENT)) {
      try {
        o.writeUTF ((String) e.arg);
        o.flush ();
      } catch (IOException ex) {
        ex.printStackTrace();
        listener.stop ();
      }
      input.setText ("");
      return true;
    } else if ((e.target == this) && (e.id == Event.WINDOW_DESTROY)) {
      if (listener != null)
        listener.stop ();
      hide ();
      return true;
    }
    return super.handleEvent (e);
  }
}
```