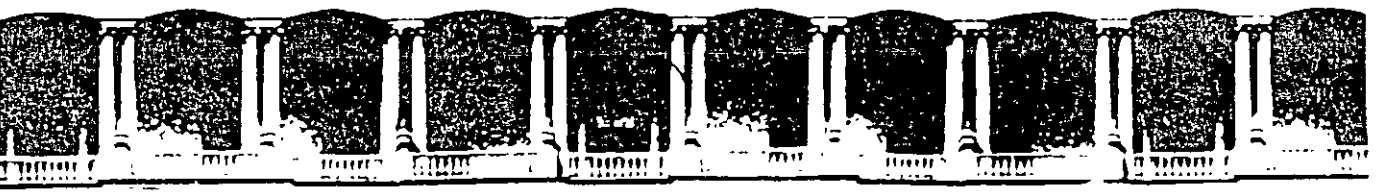


FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

P R O G R A M A C I O N

O R I E N T A D A A

O B J E T O S C O N J A V A

N O V I E M B R E D E L 2 0 0 2

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 5512-8955 5512-5121 5521-7335 5521-1987 Fax: 5510-0573 5521-4021 AL 25

PROGRAMACIÓN ORIENTADA A OBJETOS

EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Los primeros lenguajes de programación tienen su origen con las primeras computadoras digitales. En ese entonces los lenguajes eran un reflejo directo del hardware que controlaban. Poco a poco esto fue cambiando, de tal forma que la mayoría de los lenguajes contemporáneos son independientes de cualquier plataforma computacional.

LENGUAJES DE BAJO NIVEL

A finales de los años cuarenta, John von Neumman del Instituto de Estudios Avanzados de Princeton, New Jersey, con la colaboración de H. H. Goldstine y A. W. Burks, propusieron dos conceptos importantes:

1. El sistema de numeración usado por las computadoras debería ser el binario en lugar del decimal, dado que es más sencillo para los componentes electrónicos el tener que representar únicamente dos estados.

2. La memoria de la computadora, además de almacenar los datos de un programa, debería almacenar al programa mismo.

La organización de la máquina de von Neumman es relativamente simple. De manera general está compuesta de una unidad de control, una unidad aritmética, una unidad de entrada/salida, y un bloque de memoria. En la memoria se almacenan datos enteros, a los cuales se les puede aplicar operaciones aritméticas cuyos resultados son depositados en un registro llamado acumulador. Es posible modificar las localidades de memoria mediante una operación conocida como asignación. Las instrucciones del programa se toman de la memoria y se ejecutan de manera secuencial, a no ser que se encuentre una instrucción de salto, en cuyo caso la siguiente instrucción a ejecutar debe ser tomada de la localidad de memoria indicada por dicha instrucción.

A los lenguajes que una computadora puede entender de manera directa se les conoce como *lenguajes de máquina.*
De manera simplista, a cada instrucción de lenguaje de máquina se le asigna un código de operación, el cual es un simple número. Estos códigos de operación son almacenados en la memoria en su representación binaria. Por ejemplo, para incrementar en uno el valor del acumulador en el procesador Intel i486 se tiene el siguiente código de operación en binario:

01100110 01000000

Aunque para una máquina esta serie de unos y ceros es muy lógica, para los seres humanos resulta bastante incomprensible. Es por esto que surgió una variante de lenguaje de máquina conocido como lenguaje ensamblador, en donde los códigos de operación junto con otros elementos de un programa son reemplazados por nombres simbólicos. Dado el ejemplo anterior,

su equivalente en ensamblador sería:

```
inc eax
```

Aunque el ensamblador aún resulta un tanto críptico, es definitivamente más claro que su representación binaria. Tanto al lenguaje ensamblador como al lenguaje máquina se les refiere como *lenguajes de bajo nivel.*

LENGUAJES DE ALTO NIVEL

Cuando las computadoras digitales se convirtieron en un producto comercial, se vio la necesidad imponente de poder programarlas utilizando lenguajes más adecuados y sencillos que el ensamblador. Además, el lenguaje ensamblador tiene otra gran limitación : cada modelo de computadora tiene un lenguaje de máquina distinto a los demás, por lo que los programas escritos para una máquina específica solamente funcionan ahí.

Era evidente que para superar los problemas que existían en el desarrollo de los sistemas computacionales, la programación de éstos se debía realizar a un nivel superior que el impuesto por las máquinas. Es así que surge el concepto de lenguaje de alto nivel. Este tipo de lenguajes ofrecen al menos un par de ventajas significativas sobre los lenguajes de bajo nivel:

♦ **Transportabilidad**. Un lenguaje de alto nivel no es dependiente de una computadora específica; esto implica que los programas pueden correr en distintas computadoras siendo el único requisito el que exista el traductor del lenguaje para la máquina en cuestión.

♦ **Fácil comprensión**. Los programas escritos en lenguajes de alto nivel son más fáciles de escribir y entender. Esto generalmente permite una reducción en el tiempo de no tan solo la codificación, sino también de las fases de depuración y mantenimiento.

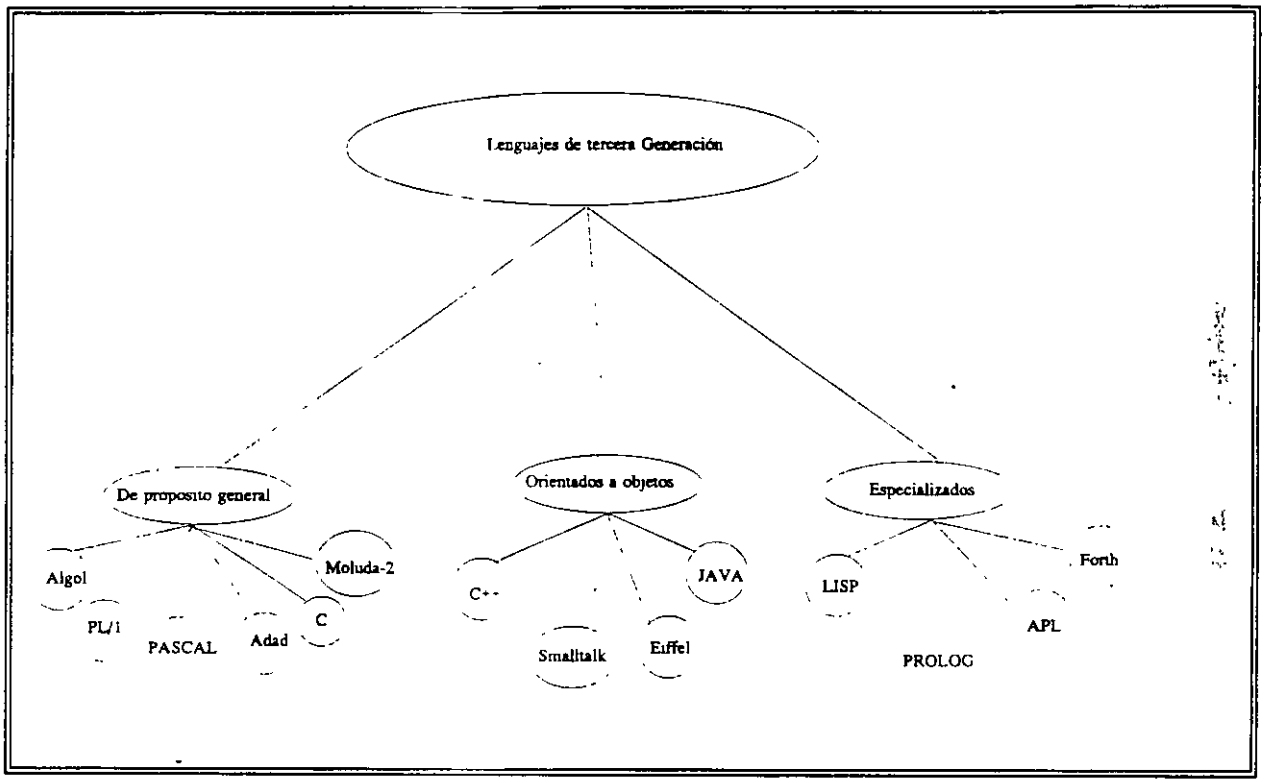
Las distintas características específicas que se han incorporado en los lenguajes de alto nivel a través de los años son muy variadas. Booch resume la evolución de los lenguajes de alto nivel en tres claras tendencias.

Los primeros lenguajes de alto nivel fueron diseñados para resolver problemas de índole numérico, por lo tanto tenían una tendencia hacia la representación de expresiones matemáticas. Posteriormente el énfasis cambió hacia cómo se debían hacer las cosas, es decir, los algoritmos comenzaron a jugar el papel preponderante. Finalmente, comenzando desde los años setenta y hasta la fecha, ha habido una tendencia hacia lo que se conoce como abstracción de datos, que consiste en diseñar los sistemas de tal forma que gran parte de un programa puede ser especificado con independencia de la representación interna de los datos.

Los lenguajes de tercera generación (también denominados *lenguajes de programación moderna o estructurada*) están caracterizados por sus potentes posibilidades procedimentales y

de estructuración de datos. Los lenguajes de esta clase se pueden dividir en tres amplias categorías, lenguajes de alto nivel de proposito general, lenguajes de alto nivel orientados a objetos y lenguajes especializados. Los lenguajes especializados han sido diseñados para satisfacer los requisitos especiales y a menudo, tienen una formulación y una sintaxis únicas.

A lo largo de la historia del desarrollo de software, siempre hemos intentado generar programas de computadora con cada vez mayores niveles de abstracción. Los lenguajes de primera generación trabajaban a nivel de instrucciones máquina, el menor nivel de abstracción



posible. Los lenguajes de segunda y tercera generación han subido el nivel de representación de los programas de computadora, pero aún hay que especificar distintos procedimientos algoritmicos perfectamente detallados. Durante la pasada década, los lenguajes de cuarta generación (4GL) han elevado aun mas el nivel de abstracción

PROGRAMACIÓN ORIENTADA A OBJETOS

Booch define a la programación orientada a objetos (POO) de la siguiente manera:

"La programación orientada a objetos es un método de implementación en donde los programas se organizan como una colección cooperativa de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases pertenecen a una jerarquía de clases unidas a través de una relación de herencia."

Un lenguaje orientado a objetos debe soportar y encausar la programación orientada a objetos. Se debe hacer la distinción entre los lenguajes que promueven la programación orientada a objetos y aquellos que simplemente la permiten. En teoría, se puede programar orientado a objetos en lenguajes que no fueron originalmente diseñados con ese propósito, pero el hacerlo constituye una labor extraordinaria.

Los conceptos generales más utilizados en el modelo orientado a objetos son: abstracción, encapsulación y modularidad. Y en cuanto a programación son: objeto, clase, método, envío y recepción de mensajes, herencia y polimorfismo.

Parece ser universalmente aceptado que para que un lenguaje sea considerado orientado a objetos debe contar con por lo menos las siguientes tres propiedades: encapsulamiento, polimorfismo y herencia. Cada una de estas propiedades se describe brevemente a continuación.

Abstracción

Abstracción es la descripción o especificación simplificada de un sistema que hace énfasis en algunos detalles o propiedades y suprime otros. Una buena abstracción es aquella que hace énfasis en los detalles significativos y suprime los irrelevantes.

La abstracción debe enfocarse más en qué es un objeto y qué hace antes de pensar en la implementación. Por ejemplo, un automóvil puede abstraerse como un objeto que sirve para desplazarse a mayor velocidad sin importar cómo lo haga.

Una característica de la abstracción es que un objeto puede abstraerse de diversas formas, dependiendo del observador. Así el automóvil que se mencionaba puede ser visto como un objeto de colección por un coleccionista, una herramienta de trabajo por un corredor profesional, una mercancía por un vendedor, etcétera.

Encapsulamiento

El encapsulamiento es la propiedad que los objetos tomaron prestada de sus primos los tipos de datos abstractos.

Un objeto tiene dos caras. La primera cara, la interfase, es la que el objeto da al mundo exterior; es la que muestra lo que puede hacer, mas no dice cómo lo hace. La otra cara, la implementación, es la que se encarga de hacer el trabajo y de mantener el estado del objeto. Solamente esta cara puede modificar el estado del objeto.

Al encapsular u ocultar información se separan los aspectos externos de un objeto (los accesibles para todos) de los detalles de implementación (los accesibles para nadie). Con esto se trata de lograr que al tener algún cambio en la implementación de un objeto no se tengan que modificar los programas que utilizan tal objeto.

Siguiendo con el ejemplo del automóvil, se sabe que existen diversos mecanismos para que funcione éste en particular se tiene el sistema de frenado que todo mundo sabe que sirve para detener el auto al pisar el pedal del freno, pero sólo el mecánico sabe los detalles de implementación. Por otro lado, si en algún momento se cambia el tipo de frenos para el conductor es transparente.

Modularidad

La modularidad consiste en dividir un programa en partes llamadas módulos, lo cuales puedan trabajarse por separado. En términos de programación, los módulos pueden compilarse por
separado y la división no depende de cierto número de líneas sino es una división en términos de
integrar en un módulo un conjunto de procedimientos relacionados entre sí junto con los datos que son manipulados por tales procedimientos. El objetivo de la modularidad es reducir el costo de elaboración de programas al poder dividir el trabajo entre varios programadores.

Por ejemplo, un automóvil está constituido por un conjunto de módulos tales como un sistema eléctrico, uno mecánico y uno de frenado. Cada módulo se trabaja por separado y el especialista sólo conoce la forma en que se relaciona su módulo con los otros pero no tiene porque saber los detalles de funcionamiento de otros módulos o sistemas.

Estos conceptos no son exclusivos de la POO pues se han desarrollado desde la programación estructurada sólo que en ésta se pueden omitir, desde luego bajo responsabilidad del programador, pues el hacerlo lleva a tener grandes programas en un solo archivo y sin estructura alguna, lo cual causa grandes pérdidas de tiempo al desear modificar tal programa. La POO no puede lograrse sin hacer uso de los mecanismos mencionados.

OBJETOS Y CLASES

A pesar de que el punto central en esta nueva metodología de programación es el concepto de objeto, resulta difícil tratar de definirlo. En un diccionario se puede encontrar la siguiente definición:

Un objeto es cualquier cosa que se ofrece a la vista y afecta los sentidos. Es una entidad tangible que exhibe algún comportamiento bien definido.

En términos de programación, un objeto no necesariamente es algo tangible (por ejemplo: un proceso). Lo que sí puede decirse de todo objeto es que tiene *estado, comportamiento e identidad.*

El *estado* de un objeto abarca todas las propiedades o características distintivas del mismo y los valores de cada una de estas propiedades. En términos de programación, puede decirse que las propiedades son las variables que sirven para describir tal objeto.

El *comportamiento* es la forma como actúa o reacciona un objeto en términos de cambios de estado, envío y recepción de mensajes. Está formado por la definición de las operaciones (funciones y procedimientos) que puede realizar este objeto. Los tipos más comunes de operaciones, o en POO métodos, son: modificar, seleccionar, iterar, construir y destruir.

El conjunto de operaciones que un objeto puede realizar sobre otro, se conoce como protocolo.

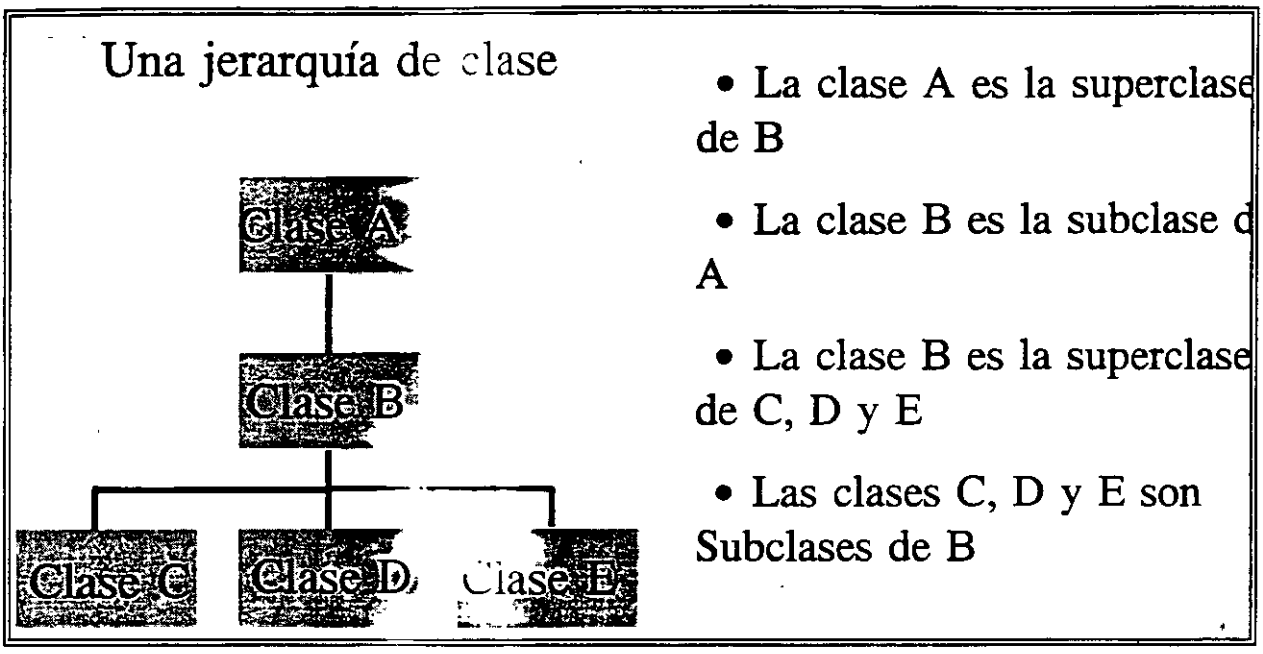
Identidad es la propiedad de un objeto que lo distingue de todos los demás. En un programa, normalmente se trata de un identificador.

En resumen, un objeto es un conjunto de localidades en memoria con un conjunto de subprogramas (en POO se conocen como métodos) que definen su comportamiento y un identificador asociado.

Lo más común es que en programa tenga más de un objeto con propiedades y comportamiento similares, así que en lugar de repetir la definición de un objeto se agrupan las características comunes de los objetos en una clase.

La clase representa la esencia del objeto y el objeto es una entidad que existe en el tiempo y el espacio. El objeto se define también como instancia de la clase a que pertenece.

La clase tiene dos vistas: La exterior, en la cual se hace énfasis en la abstracción y se oculta la estructura y secretos de comportamiento; y la vista interior o implementación. Aquí se nota que es indispensable hacer uso del concepto de encapsulación.



Herencia

Cada objeto es una in... ...a clase. Una clase es parecida a lo que se ha manejado como tipo de dato abstracto ... descripción abstracta de los datos y comportamiento que comparten objetos similares. ... es la manera de establecer relaciones entre las distintas clases que conforman a un sis... orientado a objetos. Con la herencia se establecen jerarquías del tipo "es un", en donde un... ...clase hereda la estructura y comportamiento de una o más super clases más generaliza... ...mente una subclase especializa a su super clase al aumentar o redefinir la funci... al ... esta última.

Existen lenguajes q... ... permiten que las subclases tengan un solo antecesor directo, mientras que otros p... ... tengan más de uno. A los primeros se le conoce como lenguajes con herencia sim... ...ue a estos últimos se les llama lenguajes con herencia múltiple.

La herencia es la c... ... importante de la POO, pues mediante este mecanismo es posible lograr la principa... ... POO que es la reutilización de código. El atractivo de la herencia consiste en qu... ... un nuevo componente para un programa, en lugar de diseñarlo y construirlo desd... ...sca una clase que proporcione una funcionalidad lo más parecido a la deseada; t... ...la, se crea una nueva clase que herede de ésta, y se agregan y/o modifican únic... ...nuellos detalles que sean precisos para obtener el componente que originalm... ...ecesitando. Una subclase define el comportamiento de

un conjunto de objetos que heredan algunas de las características de la clase padre, pero adquieren características especiales no compartidas por el padre, en este sentido se dice que la subclase es una especialización de la clase padre.

Lo anterior suena bien, pero no es tan sencillo de lograr. Para ello se necesita una jerarquía bien diseñada, y para lograr un buen diseño normalmente se requiere de mucha experiencia; pero una vez logrado, los beneficios son realmente compensadores.

Si se desea crear un sistema que maneje datos de estudiantes y de profesores, es fácil notar que independientemente de que sean alumnos o profesores todos son personas y por tanto tienen nombre y dirección.

Polimorfismo

Otro de los mecanismos aportados por la POO es el de polimorfismo, el cual es la capacidad de tener métodos con el mismo nombre pero que su implementación sea diferente. En la vida diaria se presenta el siguiente ejemplo de polimorfismo: al tratar de frenar un vehículo siempre se debe oprimir el pedal del lado izquierdo y el vehículo se detendrá sin importar si los frenos son de tambor o de aire.

Una forma de polimorfismo en POO, se da al usar un operador para aplicarlo a elementos de diferente tipo. Por ejemplo, al pretender sumar enteros, reales o complejos, se emplea el mismo símbolo +, esto se conoce como sobrecarga de operadores. En este caso el compilador se encarga de determinar cual es el método que se está invocando de acuerdo a los objetos involucrados en la operación.

En un programa con objetos las acciones ocurren cuando se les mandan mensajes a los objetos. El polimorfismo es una característica que permite a distintos objetos responder al mismo mensaje de manera única. El polimorfismo permite utilizar clases completamente nuevas en aplicaciones existentes, siendo el único requisito de las nuevas clases el que implementen los mensajes requeridos por la aplicación.

Un nombre de variable puede contener a diferentes momentos referencias a objetos de distintas clases que tienen un mismo antecesor. Cuando esto ocurre, dicho nombre de variable puede responder a un conjunto de operaciones comunes de diferentes maneras.

ORÍGENES DE JAVA

El nombre de Java se refiere al lenguaje de programación inventado por Sun Microsystems que se utiliza para crear contenido ejecutable que puede distribuirse. Comenzó en Sun Microsystems de California al mismo tiempo que se gestaba World Wide Web en Suiza.en 1991. La meta de ese primer grupo de desarrollo, incluyendo al creador de Java, James Gosling, era generar productos electrónicos para el consumidor que pudieran ser sencillos y estar libres

de errores. Lo que necesitaban era una manera de crear un código independiente de la plataforma y, que por lo tanto, permitiera que el software se ejecutara en cualquier CPU.

Como punto de partida para crear un lenguaje de computación que implantara esta independencia de la plataforma, el equipo de desarrollo se concentró en C++. Sin embargo, el equipo no pudo logar que C++ hiciera todo lo que querían para crear un sistema que ofreciera apoyo a una red distribuida de dispositivos heterogéneos de comunicación. El equipo abandonó el C++ e ideo un lenguaje llamado Oak (anteriormente internamente llamado Green) que después se rebautizó como Java. Hacia el otoño de 1992, el equipo había creado un proyecto llamado Star 7 (*7), que era un control remoto personal y manual.

El equipo de desarrollo fue incorporado como FirstPerson, Inc., pero entonces perdieron un concurso para producir un equipo de televisión de alta calidad para Time-Warner. A mediados de 1994 el crecimiento de la popularidad de Web atrajo la atención del equipo. Decidieron que podían construir un excelente visualizador mediante la tecnología Java. Con el objeto de llevar al Web su sistema de programación en tiempo real e independiente del CPU, construyeron un visualizador de Web.

Ese visualizador, llamado WebRunner, se programo utilizando Java y se completo a principios del otoño de 1994. Los ejecutivos de Sun Microsystems quedaron impresionados y vieron las posibilidades tecnológicas y comerciales que podían derivarse del nuevo visualizador: herramientas, servidores y ambientes de desarrollo.

El 23 de mayo de 1995, Sun Microsystems, Inc. Presentó formalmente a Java y HotJava en la Sun World '95 de San Francisco.

Cronología de Java

- San josé California, 11 de abril de 1995: el presidente de Netscape, Jim Clark, y el presidente del consejo de administración de Sun, Scott McNearly, reafirmaron se relación de trabajo y prometieron más noticias en la Sun World '95.
- San Francisco California, 23 de Mayo de 1995: en la Sun World '95 Netscape anuncio que licenciaría el lenguaje de programación Java de Sun para su visualizador Netscape Navigator.
- Verano de 1995: Java y HotJava se encuentran en la etapa alfa del proceso. La versión alfa se libera para Sun Solaris 2.3, 2.4 y 2.5 basada en SPARC. La versión alfa también se libera para Windows NT de Microsoft. Versiones en camino para Windows 95 de Microsoft y MacOS 7.5. hay versiones en camino en proyectos de terceros para plataformas y sistemas operativos, incluyendo Windows 3.1, Amiga, NeXT, Silicon Graphics y Linux.
- Verano de 1995: Sun Microsystem patrocina un concurso de programación de applets para animar la creación de applets de excelencia. Los ganadores serían anunciados en la página de HotJava el 15 de septiembre de 1995.
- Otoño-Invierno de 1995 la tecnología Java es integrada en los visualizadores de Netscape.

CARACTERÍSTICAS DE JAVA:

Simple: basado en C++ eliminando estructuras que se empleaban esporádicamente.

- aritmética de punteros
- no existen referencias
- registros (struct)
- definición de tipos (typedef)
- macros (#define)
- necesidad de liberar memoria (free)

Orientado a Objetos: Como C++, Java puede dar soporte a un enfoque orientado a objetos para la escritura del software. Permitiendo la creación de componentes que pueden volver a emplearse.

Distribuido: Java de diseño para trabajar en un ambiente de redes. Contiene una biblioteca de clases de protocolos de internet TCP/IP.

Robusto: Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error.

Interpretado: El compilador traduce un archivo fuente de clase a un código de byte estos no son específicos para una máquina determinada sino que se interpretan.

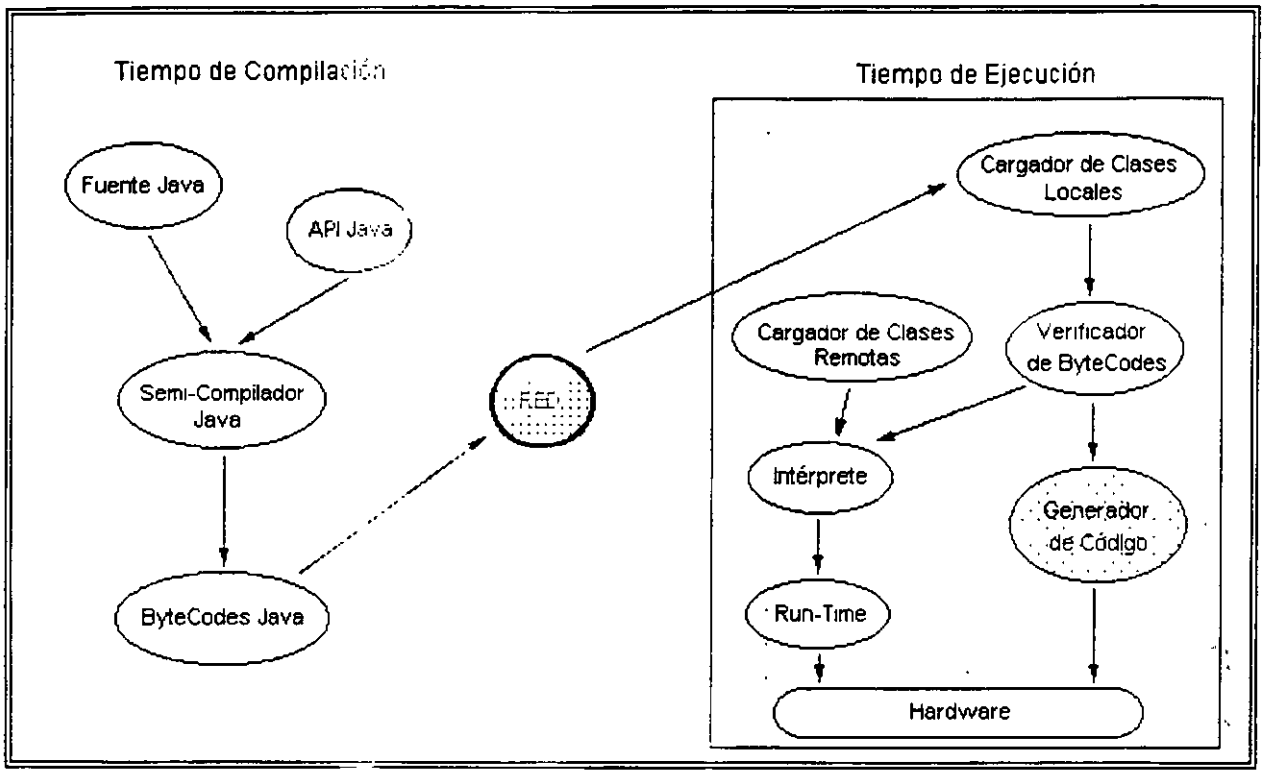
Sólido: No se "quiebra" fácilmente por errores de programación, coloca restricciones a medida que el programador escribe el código. Java no permite que el programador sobre escriba en la memoria ni corrompa otros datos a partir de apuntadores.

Seguro: Manejo cerrado de memoria, rutina de verificación de codigos de byte, seguridad de redes

De arquitectura neutral: al crearse un codigo de byte este se puede interpretar en cualquier maquina con interprete Java.

Portable: Tipos de datos consistentes.Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

Multihilos: Multitareas. Java permite muchas actividades simultáneas en un programa.



Arquitectura neutral

DEFINICIÓN DEL LENGUAJE JAVA

Cuando se programa en Java, se coloca todo el código en métodos, de la misma forma que se escriben funciones en lenguajes como C.

Comentarios
En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea

/* comentarios de una o
    más líneas
*/

/** comentario de documentación, de una o más líneas
*/
```

Los dos primeros tipos de comentarios son los que todo programador conoce y se utilizan del mismo modo. Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java, javadoc. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación de nuestras clases escrita al mismo tiempo que se genera el código.

En este tipo de comentario para documentación, se permite la introducción de algunos tokens o palabras clave, que harán que la información que les sigue aparezca de forma diferente al resto en la documentación.

Identificadores

Los identificadores nombran variables, funciones, clases y objetos; cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de pesos ($). Los siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay longitud máxima.

Son identificadores válidos:

```
identificador
nombre_usuario
Nombre_Usuario
```

```
char s[];
int iArray[];
```

Incluso se pueden construir arrays de arrays:

```
int tabla[][] = new int[4][5];
```

Los límites de los arrays se comprueban en tiempo de ejecución para evitar desbordamientos y la corrupción de memoria.

En Java un array es realmente un objeto, porque tiene redefinido el operador []. Tiene una función miembro: length. Se puede utilizar este método para conocer la longitud de cualquier array.

```
int a[][] = new int[10][3];
a.length;          /* 10 */
a[0].length;       /*  3 */
```

Para crear un array en Java hay dos métodos básicos. Crear un array vacío:

```
int lista[] = new int[50];
```

o se puede crear ya el array con sus valores iniciales:

```
String nombres[] = {
    "Juan","Pepe","Pedro","Maria"
    };
```

Esto que es equivalente a:

```
String nombres[];
 nombres = new String[4];
 nombres[0] = new String( "Juan" );
 nombres[1] = new String( "Pepe" );
 nombres[2] = new String( "Pedro" );
 nombres[3] = new String( "Maria" );
```

No se pueden crear arrays estáticos en tiempo de compilación:

```
int lista[50];  // generará un error en tiempo de compilación
```

Tampoco se puede rellenar un array sin declarar el tamaño con el operador new:

```
int lista[];
```

```
for( int i=0; i < 9; i++ )
    lista[i] = i;
```

Es decir, todos los arrays en Java son estáticos.

Operadores

Los operadores de Java son muy parecidos en estilo y funcionamiento a los de C. En la siguiente tabla aparecen los operadores que se utilizan en Java, por orden de precedencia:

```
.       []      ()
++      --
!       ~       instanceof
*       /       %
+       -
<<      >>      >>>
<       >       <=      >=      ==      !=
&       ^       |
&&      ||
?   :
=       op=     (*=     /=      %=      +=      -=      etc.)
```

Los operadores numéricos se comportan como esperamos:

```
int + int = int
```

Los operadores relacionales devuelven un valor booleano.

Para las cadenas, se pueden utilizar los operadores relacionales para comparaciones además de + y += para la concatenación:

```
String nombre = "nombre" + "Apellido";
```

El operador = siempre hace copias de objetos, marcando los antiguos para borrarlos, y ya se encargara el garbage collector de devolver al sistema la memoria ocupada por el objeto eliminado.

Separadores

Sólo hay un par de secuencias con otros caracteres que pueden aparecer en el código Java; son los separadores simples, que van a definir la forma y función del código. Los separadores admitidos en Java son:

() - paréntesis. Para contener listas de parámetros en la definición y llamada a métodos. También se utiliza para definir precedencia en expresiones, contener expresiones para control de flujo y rodear las conversiones de tipo.

{} - llaves. Para contener los valores de matrices inicializadas automáticamente. También se utiliza para definir un bloque de código, para clases, métodos y ámbitos locales.

[] - corchetes. Para declarar tipos matriz. También se utiliza cuando se referencian valores de matriz.

; - punto y coma. Separa sentencias.

, - coma. Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia for.

. - punto. Para separar nombres de paquete de subpaquetes y clases. También se utiliza para separar una variable o método de una variable de referencia.

CONTROL DE FLUJO

Muchas de las sentencias de control del flujo del programa se han tomado del C:

Sentencias de Salto Condicional

if/else

```
if( Boolean ) {
    sentencias;
    }
else {
    sentencias;
}
```

Selección

switch

```
switch( expr1 ) {
    case expr2:
        sentencias;
        break;
    case expr3:
        sentencias;
        break;
    default:
        sentencias;
        break;
    }
```

Sentencias de Ciclo

Ciclos for

```
for( expr1 inicio; expr2 test; expr3 incremento ) {
    sentencias;
    }
```

También se soporta el operador coma (,) en los ciclos for

```
for( a=0,b=0; a < 7; a++,b+=2 )
```

Ciclos while

```
while( Boolean ) {
      sentencias;
      }
```

Ciclos do/while

```
do {
      sentencias;
      }while( Boolean );
```

Control General del Flujo

```
break [etiqueta]
continue [etiqueta]
return expr;
etiqueta: sentencia;
```

En caso de que nos encontremos con bucles anidados, se permite el uso de etiquetas para poder salirse de ellos, por ejemplo:

```
uno: for( )
     {
      dos: for( )
           {
                continue;              // seguiría en el bucle
interno
            continue uno;         // seguiría en el bucle
principal
            break uno;            // se saldría del bucle
principal
           }
     }
```

En el código de una función siempre hay que ser consecuentes con la declaración que se haya hecho de ella. Por ejemplo, si se declara una función para que devuelva un entero, es imprescindible que se coloque un return final para salir de esa función, independientemente de que haya otros en medio del código que también provoquen la salida de la función. En caso de no hacerlo se generará un Warning, y el código Java no se puede compilar con Warnings.

```
int func()
{
if( a == 0 )
    return 1;
return 0;    // es imprescindible porque se retorna un entero
}
```

OBJETOS EN JAVA

TIPOS DE CLASES

Los tipos de clases que podemos definir son:

abstract

Una clase abstract tiene al menos un método abstracto. Una clase abstracta no se instancia, sino que se utiliza como clase base para la herencia.

final

Una clase final se declara como la clase que termina una cadena de herencia. No se puede heredar de una clase final. Por ejemplo, la clase Math es una clase final.

public

Las clases public son accesibles desde otras clases, bien sea directamente o por herencia. Son accesibles dentro del mismo paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas.

synchronizable

Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintos threads; el sistema se encarga de colocar las banderasa necesarias para evitarlo. Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobreescriban.

VARIABLES Y METODOS DE INSTANCIA

Una clase en Java puede contener variables y métodos. Las variables pueden ser tipos primitivos como int, char, etc. Los métodos son funciones.

Por ejemplo, en el siguiente código podemos observarlo:

```
public class MiClase {
     int i;
     public MiClase() {
          i = 10;
          }
     public void Suma_a_i( int j ) {
          i = i + j;
          }
     }
```

La clase MiClase contiene una variable (i) y dos métodos, `MiClase` que es el constructor de la clase y `Suma_a_i(int j )`.

Ambito de una variable

Los bloques de sentencias compuestas en Java se delimitan con dos llaves. Las variables de Java sólo son válidas desde el punto donde están declaradas hasta el final de la sentencia compuesta que la engloba. Se pueden anidar estas sentencias compuestas, y cada una puede contener su propio conjunto de declaraciones de variables locales. Sin embargo, no se puede declarar una variable con el mismo nombre que una de ámbito exterior.

El siguiente ejemplo intenta declarar dos variables separadas con el mismo nombre. En C y C++ son distintas, porque están declaradas dentro de ámbitos diferentes. En Java, esto es ilegal.

```
Class Ambito {
    int i = 1;          // ámbito exterior
        {               // crea un nuevo ámbito
        int i = 2;      // error de compilación
        }
    }
```

Métodos y Constructores

Los métodos son funciones que pueden ser llamadas dentro de la clase o por otras clases. El constructor es un tipo especifico de método que siempre tiene el mismo nombre que la clase.

Cuando se declara una clase en Java, se pueden declarar uno o más constructores opcionales que realizan la inicialización cuando se instancia (se crea una ocurrencia) un objeto de dicha clase.

Utilizando el código de ejemplo anterior, cuando se crea una nueva instancia de

MiClase, se crean (instancian) todos los métodos y variables, y se llama al constructor de la clase:

```
MiClase mc;
mc = new MiClase();
```

La palabra clave new se usa para crear una instancia de la clase. Antes de ser instanciada con new no consume memoria, simplemente es una declaración de tipo. Después de ser instanciado un nuevo objeto mc, el valor de i en el objeto mc será igual a 10. Se puede referenciar la variable (de instancia) i con el nombre del objeto:

```
mc.i++; // incrementa la instancia de i de mc
```

Al tener mc todas las variables y métodos de MiClase, se puede usar la primera sintaxis para llamar al método Suma_a_i() utilizando el nuevo nombre de clase mc:

```
mc.Suma_a_i( 10 );
```

y ahora la variable mc.i vale 21.

Finalizadores

Java no utiliza destructores ya que tiene una forma de recoger automáticamente todos los objetos que se salen del alcance. No obstante proporciona un método que, cuando se especifique en el código de la clase, el reciclador de memoria (garbage collector) se llamará:

```
// Cierra el canal cuando este objeto es reciclado
protected void finalize() {
    close();
    }
```

HERENCIA

La Herencia es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase Ave, se puede crear la subclase Pato, que es una especialización de Ave.

```
class Pato extends Ave {
    int numero_de_patas;
    }
```

La palabra clave `extends` se usa para generar una subclase (especialización) de un objeto. Una Pato es una subclase de Ave. Cualquier cosa que contenga la definición de Ave será copiada a la clase Pato, además, en Pato se pueden definir sus propios métodos y variables de instancia. Se dice que Pato deriva o hereda de Ave.

Además, se pueden sustituir los métodos proporcionados por la clase base. Utilizando nuestro anterior ejemplo de MiClase, aquí hay un ejemplo de una clase derivada sustituyendo a la función Suma_a_i():

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        }
    }
```

Ahora cuando se crea una instancia de MiNuevaClase, el valor de i también se inicializa a 10, pero la llamada al método Suma_a_i() produce un resultado diferente:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

En Java no se puede hacer herencia múltiple. Por ejemplo, de la clase aparato con motor y de la clase animal no se puede derivar nada, sería como obtener el objeto toro mecánico a partir de una máquina motorizada (aparato con motor) y un toro (aminal). En realidad, lo que se pretende es copiar los métodos, es decir, pasar la funcionalidad del toro de verdad al toro mecánico, con lo cual no sería necesaria la herencia múltiple sino simplemente la compartición de funcionalidad que se encuentra implementada en Java a través de interfaces.

Cuando se crea una nueva clase en Java, se puede especificar el nivel de acceso que se quiere para las variables de instancia y los métodos definidos en la clase:

public

```
public void CualquieraPuedeAcceder(){}
```

Cualquier clase desde cualquier lugar puede acceder a las variables y métodos de instacia públicos.

protected

```
protected void SoloSubClases(){}
```

Sólo las subclases de la clase y nadie más puede acceder a las variables y métodos de instancia protegidos.

private

```
private String NumeroDelCarnetDeIdentidad;
```

Las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases.

friendly (sin declaración específica)

```
void MetodoDeMiPaquete(){}
```

Por defecto, si no se especifica el control de acceso, las variables y métodos de instancia se declaran friendly (amigas), lo que significa que son accesibles por todos los objetos dentro del mismo paquete, pero no por los externos al paquete. Es lo mismo que protected.

VARIABLES Y METODOS ESTATICOS

En un momento determinado se puede querer crear una clase en la que el valor de una variable de instancia sea el mismo (y de hecho sea la misma variable) para todos los objetos instanciados a partir de esa clase. Es decir, que exista una única copia de la variable de instancia. Se usará para ello la palabra clave static.

```
class Documento extends Pagina {
    static int version = 10;
    }
```

El valor de la variable version será el mismo para cualquier objeto instanciado de la clase Documento. Siempre que un objeto instanciado de Documento cambie la variable version, ésta cambiará para todos los objetos.

De la misma forma se puede declarar un método como estático, lo que evita que el método pueda acceder a las variables de instancia no estáticas:

```
class Documento extends Pagina {
    static int version = 10;
    int numero_de_capitulos;
    static void annade_un_capitulo() {
        numero_de_capitulos++;          // esto no funciona
        }
    static void modifica_version( int i ) {
        version++;                      // esto si funciona
        }
    }
```

La modificación de la variable numero_de_capitulos no funciona porque se está violando una de las reglas de acceso al intentar acceder desde un método estático a una variable no estática.

Todas las clases que se derivan, cuando se declaran estáticas, comparten la misma página de variables; es decir, todos los objetos que se generen comparten la misma zona de memoria. Las funciones estáticas se usan para acceder solamente a variables estáticas.

```
class UnaClase {
        int var;
        UnaClase()
             {
             var = 5;
             }
        UnaFuncion()
             {
             var += 5;
             }
        }
```

En el código anterior, si se llama a la función UnaFuncion a través de un puntero a función, no se podría acceder a var, porque al utilizar un puntero a función no se pasa implícitamente el puntero al propio objeto (this). Sin embargo, sí se podría acceder a var si fuese estática, porque siempre estaría en la misma posición de memoria para todos los objetos que se creasen de UnaClase.

Al acceder a variables de instancia de una clase, la palabra clave this hace referencia a los miembros de la propia clase. Volviendo al ejemplo de MiClase, se puede añadir otro constructor de la forma siguiente:

```
public class MiClase {
    int i;
    public MiClase() {
        i = 10;
        }
    // Este constructor establece el valor de i
    public MiClase( int valor ) {
        this.i = valor;  // i = valor
        }
    public void Suma_a_i( int j ) {
        i = i + j;
        }
    }
```

Aquí this.i se refiere al entero i en la clase MiClase.

Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se puede hacer referencia al método padre con la palabra clave super:

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        super.Suma_a_i( j );
        }
    }
```

En el siguiente código, el constructor establecerá el valor de i a 10, después lo cambiará a 15 y finalmente el método Suma_a_i() de la clase padre (MiClase) lo dejará en 25:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

INTERFACES

Un interface contiene una colección de métodos que se implementan en otro lugar. Los métodos de una clase son public, static y final.

Una interface proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Por ejemplo:

```
public interface VideoClip {
     // comienza la reproduccion del video
     void play();
     // reproduce el clip en un bucle
     void bucle();
     // detiene la reproduccion
     void stop();
     }
```

Las clases que quieran utilizar el interface VideoClip utilizarán la palabra implements y proporcionarán el código necesario para implementar los métodos que se han definido para el interface:

```
class MiClase implements VideoClip {
     void play() {
          <código>
          }
     void bucle() {
          <código>
          }
     void stop() {
          <código>
          }
```

Al utilizar implements para el interface es como si se hiciese una acción de copiar-y-pegar del código del interface, con lo cual no se hereda nada, solamente se pueden usar los métodos.

La ventaja principal del uso de interfaces es que una clase interface puede ser implementada por cualquier número de clases, permitiendo a cada clase compartir el interfaz de programación sin tener que ser consciente de la implementación que hagan las otras clases que implementen el interface.

```
class MiOtraClase implements VideoClip {
     void play() {
          <código nuevo>
          }
     void bucle() {
          <código nuevo>
          }
     void stop() {
          <código nuevo>
          }
```

PAQUETES

La palabra clave package permite agrupar clases e interfaces. Los nombres de los paquetes son palabras separadas por puntos y se almacenan en directorios que coinciden con esos nombres.

Por ejemplo, los archivos siguientes, que contienen código fuente Java:

Applet.java, AppletContext.java, AppletStub.java, AudioClip.java

contienen en su código la línea:

```
package java.applet;
```

Y las clases que se obtienen de la compilación de los archivos anteriores, se encuentran con el nombre nombre_de_clase.class, en el directorio:

java/applet

Import

Los paquetes de clases se cargan con la palabra clave import, especificando el nombre del paquete como ruta y nombre de clase. Se pueden cargar varias clases utilizando un asterisco.

```
import java.Date;
import java.awt.*;
```

Si un archivo fuente Java no contiene ningún package, se coloca en el paquete por defecto sin nombre. Es decir, en el mismo directorio que el archivo fuente, y la clase puede ser cargada con la sentencia import:

```
import MiClase;
```

Paquetes de Java

El lenguaje Java proporciona una serie de paquetes que incluyen ventanas, utilidades, un sistema de entrada/salida general, herramientas y comunicaciones. En la versión actual del JDK, los paquetes Java que se incluyen son:

`java.applet`

Este paquete contiene clases diseñadas para usar con applets. Hay una clase Applet y tres interfaces: AppletContext, AppletStub y AudioClip.

`java.awt`

El paquete Abstract Windowing Toolkit (awt) contiene clases para generar widgets y componentes GUI (Interfaz Gráfica de Usuario). Incluye las clases Button, Checkbox, Choice, Component, Graphics, Menu, Panel, TextArea y TextField.

`java.io`

El paquete de entrada/salida contiene las clases de acceso a archivos: FileInputStream y FileOutputStream.

`java.lang`

Este paquete incluye las clases del lenguaje Java propiamente dicho: Object, Thread, Exception, System, Integer, Float, Math, String, etc.

`java.net`

Este paquete da soporte a las conexiones del protocolo TCP/IP y, además, incluye las clases Socket, URL y URLConnection.

`java.util`

Este paquete es una miscelánea de clases útiles para muchas cosas en programación. Se incluyen, entre otras, Date (fecha), Dictionary (diccionario), Random (números aleatorios) y Stack (pila FIFO).

IMPLEMENTANDO UNA APLICACIÓN

La aplicación más pequeña posible es la que simplemente imprimir un mensaje en la pantalla. Tradicionalmente, el mensaje suele ser "Hola Mundo!". Esto es justamente lo que hace el siguiente fragmento de código:

```
//
// Aplicación HolaMundo de ejemplo
//
class HolaMundoApp {
    public static void main( String args[] ) {
        System.out.println( "Hola Mundo!" );
        }
    }
```

HolaMundo

Vamos ver en detalle la aplicación anterior, línea a línea. Esas líneas de código contienen los componenetes mínimos para imprimir Hola Mundo! en la pantalla.

```
//
// Aplicación HolaMundo de ejemplo
//
```

Estas tres primera líneas son comentarios. Hay tres tipos de comentarios en Java, // es un comentario orientado a línea.

```
class HolaMundoApp {
```

Esta línea declara la clase HolaMundoApp. El nombre de la clase especificado en el Archivo fuente se utiliza para crear un Archivo nombredeclase.class en el directorio en el que se compila la aplicación. En nuestro caso, el compilador creará un Archivo llamado HolaMundoApp.class.

```
    public static void main( String args[] ) {
```

Esta línea especifica un método que el intérprete Java busca para ejecutar en primer lugar. Igual que en otros lenguajes, Java utiliza una palabra clave main para especificar la primera función a ejecutar. En este ejemplo tan simple no se pasan argumentos.

`public` significa que el método main puede ser llamado por cualquiera, incluyendo el intérprete Java.

`static` es una palabra clave que le dice al compilador que main se refiere a la propia

clase HolaMundoApp y no a ninguna instancia de la clase. De esta forma, si alguien intenta hacer otra instancia de la clase, el método main no se instanciaría.

`void` indica que main no devuelve nada. Esto es importante ya que Java realiza una estricta comprobación de tipos, incluyendo los tipos que se ha declarado que devuelven los métodos.

`args[]` es la declaración de un array de Strings. Estos son los argumentos escritos tras el nombre de la clase en la línea de comandos:

```
%java HolaMundoApp arg1 arg2 ...
```

```
        System.out.println( "Hola Mundo!" );
```

Esta es la funcionalidad de la aplicación. Esta línea muestra el uso de un nombre de clase y método. Se usa el método `println()` de la clase `out` que está en el paquete `System`.

El método `println()` toma una cadena como argumento y la escribe en el stream de salida estándar; en este caso, la ventana donde se lanza la aplicación.

```
    }
}
```

Finalmente, se cierran las llaves que limitan el método `main()` y la clase `HolaMundoApp`.
Vamos a ver a continuación como podemos ver el resultado de nuestra primera aplicación Java en pantalla.

Generaremos un Archivo con el código fuente de la aplicación, lo compilaremos y utilizaremos el intérprete java para ejecutarlo.

Archivos Fuente Java

Los Archivos fuente en Java terminan con la extensión ".java". Crear un Archivo utilizando cualquier editor de texto ascii que tenga como contenido el código de las ocho líneas de nuestra mínima aplicación, y salvarlo en un Archivo con el nombre de HolaMundoApp.java. Para crear los Archivos con código fuente Java no es necesario un procesador de textos, aunque puede utilizarse siempre que tenga salida a Archivo de texto plano o ascii, sino que es suficiente con cualquier otro editor.

Compilación

El compilador javac se encuentra en el directorio bin por debajo del directorio java, donde se haya instalado el JDK.

Este directorio bin, si se han seguido las instrucciones de instalación, debería formar parte de la variable de entorno PATH del sistema. Si no es así, tendría que revisar la Instalación del JDK. El compilador de Java traslada el código fuente Java a byte-codes, que son los componentes que entiende la Máquina Virtual Java que está incluida en los navegadores con soporte Java y en appletviewer.

Una vez creado el Archivo fuente HolaMundoApp.java, se puede compilar con la línea siguiente:

```
%javac HolaMundoApp.java
```

Si no se han cometido errores al teclear ni se han tenido problemas con el path al Archivo fuente ni al compilador, no debería aparecer mensaje alguno en la pantalla, y cuando vuelva a aparecer el prompt del sistema, se debería ver un Archivo HolaMundoApp.class nuevo en el directorio donde se encuentra el Archivo fuente.

Si ha habido algún problema, en Problemas de compilación al final de esta sección, hemos intentado reproducir los que más frecuentemente se suelen dar, se pueden consultar por si pueden aportar un poco de luz al error que haya aparecido

Ejecución

Para ejecutar la aplicación HolaMundoApp, hemos de recurrir al intérprete java, que también se encuentra en el directorio bin, bajo el directorio java. Se ejecutará la aplicación con la línea:

```
%java HolaMundoApp
```

y debería aparecer en pantalla la respuesta de Java:

%Hola Mundo!

El símbolo % representa al prompt del sistema, y lo utilizaremos para presentar las respuestas que nos ofrezca el sistema como resultado de la ejecución de los comandos que se indiquen en pantalla o para indicar las líneas de comandos a introducir.

Problemas de compilación

A continuación presentamos una lista de los errores más frecuentes que se presentan a la hora de compilar un Archivo con código fuente Java, nos basaremos en errores provocados sobre nuestra minima aplicación Java de la sección anterior, pero podria generalizarse sin demasiados problemas.

```
%javac: Command not found
```

No se ha establecido correctamente la variable PATH del sistema para el compilador javac. El compilador javac se encuentra en el directorio bin, que cuelga del directorio java, que cuelga del directorio donde se haya instalado el JDK (Java Development Kit).

```
%HolaMundoApp.java:3: Method printl(java.lang.String) not found in
class java.io.PrintStream.
 System.out.printl( "HolaMundo!);
```

Error tipográfico, el método es println no printl.

```
%In class HolaMundoApp: main must be public and static
```

Error de ejecución, se olvidó colocar la palabra static en la declaración del método main de la aplicación.

```
%Can't find class HolaMundoApp
```

Este es un error muy sutil. Generalmente significa que el nombre de la clase es distinto al del Archivo que contiene el código fuente, con lo cual el Archivo nombre_Archivo.class que se genera es diferente del que cabria esperar. Por ejemplo, si en nuestro Archivo de código fuente de nuestra aplicación HolaMundoApp.java colocamos en vez de la declaración actual de la clase HolaMundoApp, la linea

```
class HolaMundoapp {
```

se creará un Archivo HolaMundoapp.class, que es diferente del HolaMundoApp.class, que es el nombre esperado de la clase; la diferencia se encuentra en la a minúscula y mayúscula.

MANEJO DE EXCEPCIONES

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 hay un error, la aplicación no debería morirse y generar un core (o un crash en caso del DOS). Se debería lanzar (throw) una excepción que nosotros deberíamos capturar (catch) y resolver la situación de error. Utilizadas en forma adecuada, las excepciones aumentan en gran medida la robustez de las aplicaciones.

La gestión de excepciones de Java se gestiona mediante cinco palabras clave: `try`, `catch`, `throw`, `throws` y `finally`. Básicamente, se intenta (`try`) ejecutar un bloque de código, y si se produce un error, el sistema lanza (`throws`) una excepción que se puede capturar (`catch`) en base al tipo de la excepción o ser tratada finalmente (`finally`) por un gestor por omisión.

Esta es la forma básica de un bloque de gestión de excepciones.

```
try {
// bloque de código
}catch (tipoExcepcion1 e){
     //gestor de excepciónes para tipoexcepcion1
}catch (tipoExcepcion2 e){
     //gestor de excepciones para tipoexcepcion2
     throw(e); //volver a lanzar la excepción
}finally{
}
```

Cuando se produce un error se debería generar, o lanzar, una excepción. Para que un método en Java, pueda lanzar excepciones, hay que indicarlo expresamente.

```
void MetodoAsesino() throws NullPointerException,CaidaException
```

Se pueden definir excepciones propias, no hay por qué limitarse a las predefinidas; bastará con extender la clase Exception y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción.

También pueden producirse excepciones no de forma explícita como en el caso anterior, sino de forma implícita cuando se realiza alguna acción ilegal o no válida.

Las excepciones, pues, pueden originarse de dos modos: el programa hace algo ilegal (caso normal), o el programa explícitamente genera una excepción ejecutando la sentencia throw (caso menos normal). La sentencia throw tiene la
siguiente forma:

```
throw ObtejoExcepction;
```

El objeto `ObjetoException` es un objeto de una clase que extiende la clase Exception.

El siguiente código de ejemplo origina una excepción de división por cero:

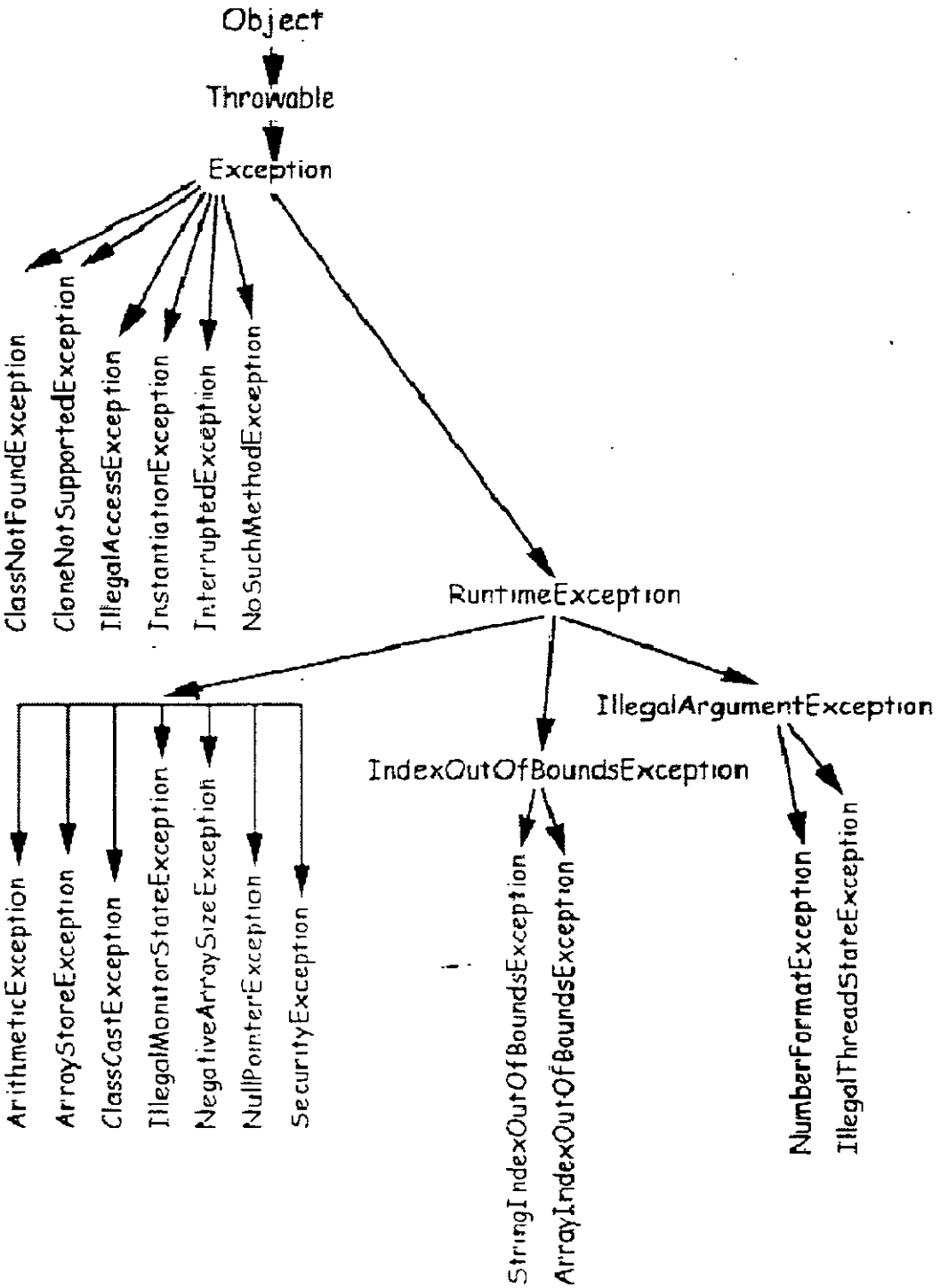
```
class melon {
    public static void main( String[] a ) {
        int i=0, j=0, k;

        k = i/j;     // Origina un error de division-by-zero
        }
    }
```

Si compilamos y ejecutamos esta aplicación Java, obtendremos la siguiente salida por pantalla:

```
> javac melon.java
> java melon
     java.lang.ArithmeticException: / by zero
            at melon.main(melon.java:5)
```

Las excepciones predefinidas, como ArithmeticException, se conocen como excepciones runtime. Actualmente, como todas las excepciones son eventos runtime, sería mejor llamarlas excepciones irrecuperables. Esto contrasta con las excepciones que generamos explícitamente, que suelen ser mucho menos severas y en la mayoría de los casos podemos recuperarnos de ellas. Por ejemplo, si un fichero no puede abrirse, preguntamos al usuario que nos indique otro fichero; o si una estructura de datos se encuentra completa, podremos sobreescribir algún elemento que ya no se necesite.

Las excepciones predefinidas y su jerarquía de clases es la que se muestra en la figura:



Los nombres de las excepciones indican la condición de error que representan. Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:

`ArithmeticException`

Las excepciones aritméticas son típicamente el resultado de una división por 0:

int i = 12 / 0;

`NullPointerException`

Se produce cuando se intenta acceder a una variable o método antes de ser definido:

```
class Hola extends Applet {
    Image img;

    paint( Graphics g ) {
        g.drawImage( img,25,25,this );
        }
    }
```

`IncompatibleClassChangeException`

El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

`ClassCastException`

El intento de convertir un objeto a otra clase que no es válida.

```
y = (Prueba)x;        // donde
x no es de tipo Prueba
```

`NegativeArraySizeException`

Puede ocurrir si hay un error aritmético al intentar cambiar el tamaño de un array.

`OutOfMemoryException`

¡No debería producirse nunca! El intento de crear un objeto con el operador new ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el garbage collector se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

`NoClassDefFoundException`

Se referenció una clase que el sistema es incapaz de encontrar.

`ArrayIndexOutOfBoundsException`

Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

`UnsatisfiedLinkException`

Se hizo el intento de acceder a un método nativo que no existe. Aquí no existe un método a.kk

```
class A {
    native void kk();
    }
```

y se llama a `a.kk()`, cuando debería llamar a `A.kk()`.

`InternalException`

Este error se reserva para eventos que no deberían ocurrir. Por definición, el usuario nunca debería ver este error y esta excepción no debería lanzarse.

CREANDO EXCEPCIONES PROPIAS

También podemos lanzar nuestras propias excepciones, extendiendo la clase System.exception. Por ejemplo, consideremos un programa cliente/servidor. El código cliente se intenta conectar al servidor, y durante 5 segundos se espera a que conteste el servidor. Si el servidor no responde, el servidor lanzaría la excepción de time-out:

```
class ServerTimeOutException extends Exception {}

public void conectame( String nombreServidor ) throws Exception {
    int exito;
    int puerto = 80;

    exito = open( nombreServidor,puerto );
    if( exito == -1 )
        throw ServerTimeOutException;
    }
```

Si se quieren capturar las propias excepciones, se deberá utilizar la sentencia try:

```
public void encuentraServidor() {
   ...
   try {
        conectame( servidorDefecto );
        catch( ServerTimeOutException e ) {
            g.drawString(
                "Time-out del Servidor, intentando alternativa",
                5,5 );
            conectame( servidorAlterno );
            }
    ...
    }
```

Cualquier método que lance una excepción también debe capturarla, o declararla como parte de la interface del método. Cabe preguntarse entonces, el porqué de lanzar una excepción si hay que capturarla en el mismo método. La respuesta es que las excepciones no simplifican el trabajo del control de errores. Tienen la ventaja de que se puede tener muy localizado el control de errores y no tenemos que controlar millones de valores de retorno, pero no van más allá.

CAPTURANDO EXCEPCIONES

Las excepciones lanzadas por un método que pueda hacerlo deben recoger en bloque try/catch o try/finally.

```
int valor;
try {
    for( x=0,valor = 100; x < 100; x ++ )
        valor /= x;
    }
catch( ArithmeticException e ) {
    System.out.println( "Matemáticas locas!" );
    }
catch( Exception e ) {
    System.out.println( "Se ha producido un error" );
    }
```

`try`

Es el bloque de código donde se prevé que se genere una excepción. Es como si dijésemos "intenta estas sentencias y mira a ver si se produce una excepción". El bloque `try` tiene que ir seguido, al menos, por una cláusula `catch` o una cláusula `finally`

`catch`

Es el código que se ejecuta cuando se produce la excepción. Es como si dijésemos "controlo cualquier excepción que coincida con mi argumento". En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

```
catch( Excepcion e ) { ...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

Se pueden controlar grupos de excepciones, es decir, que se pueden controlar, a través del argumento, excepciones semejantes. Por ejemplo:

```
class Limites extends Exception {}
class demasiadoCalor extends Limites {}
class demasiadoFrio extends Limites {}
class demasiadoRapido extends Limites {}
class demasiadoCansado extends Limites {}
    .
    .
    .
    try {
        if( temp > 40 )
            throw( new demasiadoCalor() );
        if( dormir < 8 )
            throw( new demasiado Cansado() );
    } catch( Limites lim ) {
        if( lim instanceof demasiadoCalor )
            {
            System.out.println( "Capturada excesivo calor!" );
            return;
            }
        if( lim instanceof demasiadoCansado )
            {
            System.out.println( "Capturada excesivo cansancio!" );
            return;
            }
    } finally
        System.out.println( "En la clausula finally" );
```

La cláusula `catch` comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque. El operador `instanceof` se utiliza para identificar exactamente cual ha sido la identidad de la excepción.

`finally`

Es el bloque de código que se ejecuta siempre, haya o no excepción. Hay una cierta controversia entre su utilidad, pero, por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejarnos grabado si se producen excepciones y nos hemos recuperado de ellas o no.

Este bloque `finally` puede ser útil cuando no hay ninguna excepción. Es un trozo de código que se ejecuta independientemente de lo que se haga en el bloque `try`.

Cuando vamos a tratar una excepción, se nos plantea el problema de qué acciones vamos a tomar. En la mayoría de los casos, bastará con presentar una indicación de error al usuario y un mensaje avisándolo de que se ha producido un error y que decida si quiere o no continuar con la ejecución del programa.

Por ejemplo, podríamos disponer de un diálogo como el que se presenta en el código siguiente:

```
public class DialogoError extends Dialog {
    DialogoError( Frame padre ) {
        super( padre,true );
        setLayout( new BorderLayout() );
        // Presentamos un panel con continuar o salir
        Panel p = new Panel();
        p.add( new Button( "¿Continuar?" ) );
        p.add( new Button( "Salir" ) );
        add( "Center",new Label(
            "Se ha producido un error. ¿Continuar?" ) )
        add( "South",p );
        }
    public boolean action( Event evt,Object obj ) {
        if( "Salir".equals( obj ) )
            {
            dispose();
            System.exit( 1 );
            }
        return false;
        }
    }
```

Y la invocación, desde algún lugar en que se suponga que se generarán errores, podría ser como sigue:

```
try {
    // Código peligroso
    }
```

```
catch( AlgunaExcepcion e ) {
    VentanaError = new DialogoError( this );
    VentanaError.show();
    }
```

PROPAGACION DE EXCEPCIONES

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque y sigue el flujo de control por el bloque finally (si lo hay) y concluye el control de la excepción.

Si ninguna de las cláusulas catch coincide con la excepción que se ha producido, entonces se ejecutará el código de la cláusula finally (en caso de que la haya). Lo que ocurre en este caso, es exactamente lo mismo que si la sentencia que lanza la excepción no se encontrase encerrada en el bloque try.

El flujo de control abandona este método y retorna prematuramente al método que lo llamó. Si la llamada estaba dentro del ámbito de una sentencia try, entonces se vuelve a intentar el control de la excepción, y así continuamente.

Veamos lo que sucede cuando una excepción no es tratada en la rutina en donde se produce. El sistema Java busca un bloque try..catch más allá de la llamada, pero dentro del método que lo trajo aquí. Si la excepción se propaga de todas formas hasta lo alto de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje. Es decir, podemos suponer que Java nos está proporcionando un bloque catch por defecto, que imprime un mensaje de error y sale.

No hay ninguna sobrecarga en el sistema por incorporar sentencias try al código. La sobrecarga se produce cuando se genera la excepción.

Hemos dicho ya que un método debe capturar las excepciones que genera, o en todo caso, declararlas como parte de su llamada, indicando a todo el mundo que es capaz de generar excepciones. Esto debe ser así para que cualquiera que escriba una llamada a ese método esté avisado de que le puede llegar una excepción, en lugar del valor de retorno normal. Esto permite al programador que llama a ese método, elegir entre controlar la excepción o propagarla hacia arriba en la pila de llamadas. La siguiente línea de código muestra la forma general en que un método declara excepciones que se pueden propagar fuera de él:

```
tipo_de_retorno( parametros ) throws e1,e2,e3 { }
```

Los nombres e1,e2,... deben ser nombres de excepciones, es decir, cualquier tipo que sea asignable al tipo predefinido Throwable. Observar que, como en la llamada al método se especifica el tipo de retorno, se está especificando el tipo de excepción que puede generar (en

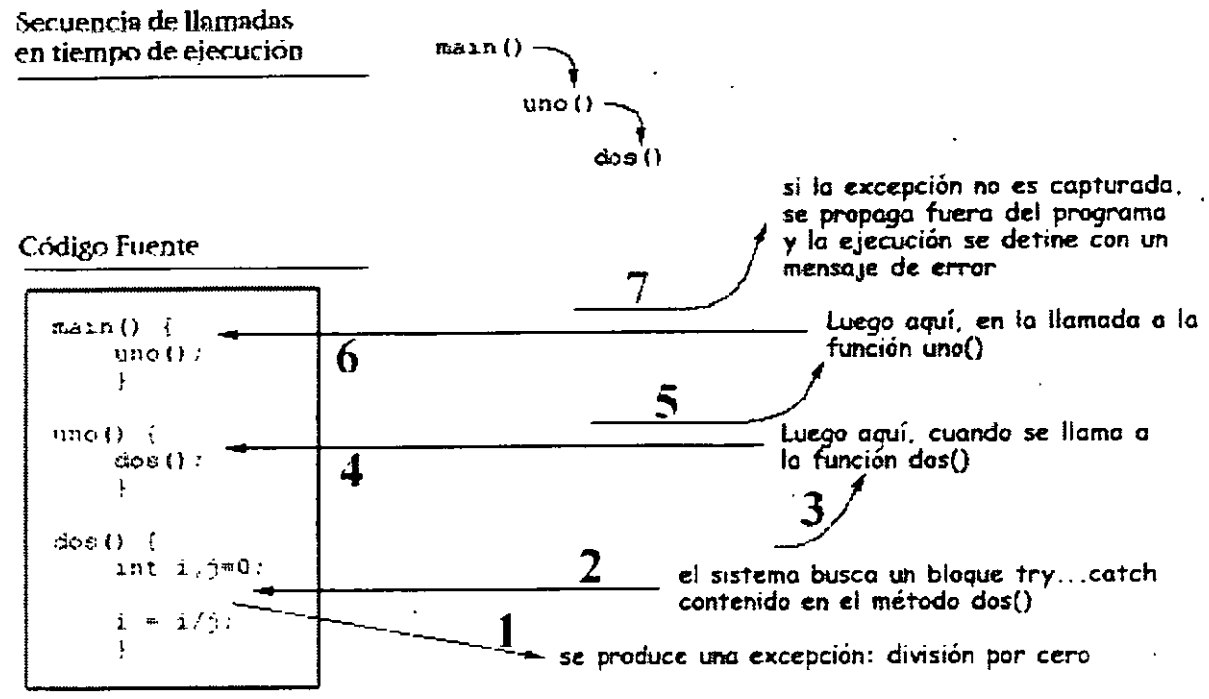
lugar de un objeto exception).

He aquí un ejemplo, tomado del sistema Java de entrada/salida:

```
byte readByte() throws IOException;
short readShort() throws IOException;
char readChar() throws IOException;

void writeByte( int v ) throws IOException;
void writeShort( int v ) throws IOException;
void writeChar( int v ) throws IOException;
```

Lo más interesante aquí es que la rutina que lee un char, puede devolver un char; no el entero que se requiere en C. C necesita que se devuelva un int, para poder pasar cualquier valor a un char, y además un valor extra (-1) para indicar que se ha alcanzado el final del fichero. Algunas de las rutinas Java lanzan una excepción cuando se alcanza el fin del fichero.

En el siguiente diagrama se muestra gráficamente cómo se propaga la excepción que se genera en el código, a través de la pila de llamadas durante la ejecución del código:



Cuando se crea una nueva excepción, derivando de una clase Exception ya existente, se puede cambiar el mensaje que lleva asociado. La cadena de texto puede ser recuperada a través de un método. Normalmente, el texto del mensaje proporcionará información para resolver el problema o sugerirá una acción alternativa. Por ejemplo:

```
class SinGasolina extends Exception {
    SinGasolina( String s ) {    // constructor
        super( s );
        }
    ....

// Cuando se use, aparecerá algo como esto
    try {
        if( j < 1 )
            throw new SinGasolina("Usando deposito de reserva" );
    } catch( SinGasolina e ) {
        System.out.println( o.getMessage() );
        }
```

Esto, en tiempo de ejecución originaría la siguiente salida por pantalla:

```
> Usando deposito de reserva
```

Otro método que es heredado de la superclase Throwable es printStackTrace(). Invocando a este método sobre una excepción se volcará a pantalla todas las llamadas hasta el momento en donde se generó la excepción (no donde se maneje la excepción). Por ejemplo:

```
// Capturando una excepción en un método
class testcap {
    static int slice0[] = { 0,1,2,3,4 };

    public static void main( String a[] ) {
        try {
            uno();
        } catch( Exception e ) {
                System.out.println( "Captura de la excepcion en
main()" );
            e.printStackTrace();
            }
        }

    static void uno() {
        try {
            slice0[-1] = 4;
        } catch( NullPointerException e ) {
            System.out.println( "Captura una excepcion diferente"
);
            }
        }
    }
```

Cuando se ejecute ese código, en pantalla observaremos la siguiente salida:

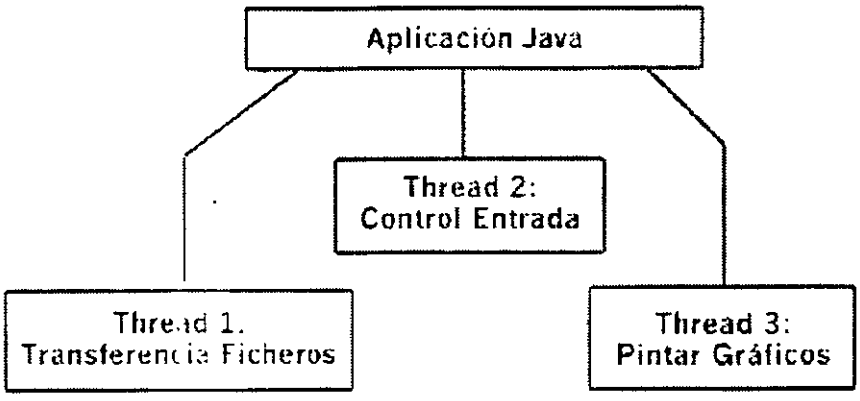
```
> Captura de la excepcion en main()
> java.lang.ArrayIndexOutOfBoundsException: -1
        at testcap.uno(test5p.java:19)
        at testcap.main(test5p.java:9)
```

Con todo el manejo de excepciones podemos concluir que proporciona un método más seguro para el control de errores, además de representar una excelente herramienta para organizar en sitios concretos todo el manejo de los errores y, además, que podemos proporcionar mensajes de error más decentes al usuario indicando qué es lo que ha fallado y por qué, e incluso podemos, a veces, recuperarnos de los errores.

La degradación que se produce en la ejecución de programas con manejo de excepciones está ampliamente compensada por las ventajas que representa en cuanto a seguridad de funcionamiento de esos mismos programas.

MULTITAREA

Considerando el entorno multithread, cada thread (hilo, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. A veces se les llama procesos ligeros o contextos de ejecución. Típicamente, cada thread controla un único aspecto dentro de un programa, como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todos los threads comparten los mismos recursos, al contrario que los procesos en donde cada uno tiene su propia copia de código y datos (separados unos de otros). Gráficamente, los threads se parecen en su funcionamiento a lo que muestra la figura



siguiente:

Programas de flujo único

Un programa de flujo único o mono-hilvanado (single-thread) utiliza un único flujo de control (thread) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchos de los applets y aplicaciones son de flujo único.

Por ejemplo, en nuestra aplicación estándar de saludo:

```
public class HolaMundo {
    static public void main( String args[] ) {
        System.out.println( "Hola Mundo!" );
        }
    }
```

Aquí, cuando se llama a main(), la aplicacion imprime el mensaje y termina. Esto ocurre dentro de un único thread.

Programas de flujo múltiple

En nuestra aplicación de saludo, no vemos el thread que ejecuta nuestro programa. Sin embargo, Java posibilita la creación y control de threads explícitamente. La utilización de threads en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar threads, permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se puede con otros lenguajes de tercera generación. En un lenguaje orientado a Internet como es Java, esta herramienta es vital.

Si se ha utilizado un navegador con soporte Java, ya se habrá visto el uso de múltiples threads en Java. Habrá observado que dos applet se pueden ejecutar al mismo tiempo, o que puede desplazar la página del navegador mientras el applet continúa ejecutándose. Esto no significa que el applet utilice múltiples threads, sino que el navegador es multithreaded.

Las aplicaciones (y applets) multithreaded utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un thread para cada subtarea.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multithreaded permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

Vamos a modificar nuestro programa de saludo creando tres threads individuales, que imprimen cada uno de ellos su propio mensaje de saludo, MultiHola.java:

```
// Definimos unos sencillos threads. Se detendrán un rato
// antes de imprimir sus nombres y retardos

class TestTh extends Thread {
    private String nombre;
    private int retardo;

    // Constructor para almacenar nuestro nombre
    // y el retardo
    public TestTh( String s,int d ) {
        nombre = s;
        retardo = d;
        }

    // El metodo run() es similar al main(), pero para
    // threads. Cuando run() termina el thread muere
    public void run() {
        // Retasamos la ejecución el tiempo especificado
        try {
            sleep( retardo );
        } catch( InterruptedException e ) {
```

```
            ;
            }

        // Ahora imprimimos el nombre
        System.out.println( "Hola Mundo! "+nombre+" "+retardo );
        }
    }

public class MultiHola {
    public static void main( String args[] ) {
        TestTh t1,t2,t3;

        // Creamos los threads
        t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
        t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
        t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );

        // Arrancamos los threads
        t1.start();
        t2.start();
        t3.start();
        }
    }
```

Creación de un Thread

Hay dos modos de conseguir threads en Java. Una es implementando la interface Runnable, la otra es extender la clase Thread.

La implementación de la interface Runnable es la forma habitual de crear threads. Las interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar. La interface define el trabajo y la clase, o clases, que implementan la interface realizan ese trabajo. Los diferentes grupos de clases que implementen la interface tendrán que seguir las mismas reglas de funcionamiento.

Hay una cuantas diferencias entre interface y clase. Primero, una interface solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes. Segundo, una interface no puede implementar cualquier método. Una clase que implemente una interface debe implementar todos los métodos definidos en esa interface. Una interface tiene la posibilidad de poder extenderse de otras interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces. Ademas, una interface no puede ser instanciada con el operador new; por ejemplo, la siguiente sentencia no está permitida:

```
Runnable a = new Runnable();    // No se permite
```

El primer método de crear un thread es simplemente extender la clase Thread:

```
class MiThread extends Thread {
    public void run() {
        . . .
        }
```

El ejemplo anterior crea una nueva clase MiThread que extiende la clase Thread y sobrecarga el método Thread.run() por su propia implementación. El método run() es donde se realizará todo el trabajo de la clase. Extendiendo la clase Thread, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Esta limitación de Java puede ser superada a través de la implementación de Runnable:

```
public class MiThread implements Runnable {
    Thread t;
    public void run() {
        // Ejecución del thread una vez creado
        }
    }
```

En este caso necesitamos crear una instancia de Thread antes de que el sistema pueda ejecutar el proceso como un thread. Además, el método abstracto run() está definido en la interface Runnable tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía tenemos oportunidad de extender la clase MiThread, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un thread , implementarán la interface Runnable, ya que probablemente extenderán alguna de su funcionalidad a otras clases.

No pensar que la interface Runnable está haciendo alguna cosa cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual es una clase para dar idea sobre el diseño de la clase Thread. De hecho, si vemos los fuentes de Java, podremos comprobar que solamente contiene un método abstracto:

```
package java.lang;
public interface Runnable {
    public abstract void run() ;
}
```

Y esto es todo lo que hay sobre la interface Runnable. Como se ve, una interface sólo proporciona un diseño para las clases que vayan a ser implementadas. En el caso de Runnable, fuerza a la definición del método run(), por lo tanto, la mayor parte del trabajo se hace en la clase Thread. Un vistazo un poco más profundo a la definición de la clase Thread nos da idea de lo que realmente está pasando:

```
public class Thread implements Runnable {
    ...
    public void run() {
        if( tarea != null )
            tarea.run() ;
            }
        }
    ...
    }
```

De este trocito de código se desprende que la clase Thread también implemente la interface Runnable. tarea.run() se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un thread) no sea nula y ejecuta el método run() de esa clase. Cuando esto suceda, el método run() de la clase hará que corra como un thread.

Arranque de un Thread

Las aplicaciones ejecutan main() tras arrancar. Esta es la razón de que main() sea el lugar natural para crear y arrancar otros threads. La línea de código:

```
t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
```

crea un nuevo thread. Los dos argumentos pasados representan el nombre del thread y el tiempo que queremos que espere antes de imprimir el mensaje.

Al tener control directo sobre los threads, tenemos que arrancarlos explícitamente. En nuestro ejemplo con:

```
t1.start();
```

start(), en realidad es un método oculto en el thread que llama al método run().

Manipulación de un Thread

Si todo fue bien en la creación del thread, t1 debería contener un thread válido, que controlaremos en el método run().

Una vez dentro de run(), podemos comenzar las sentencias de ejecución como en otros programas. run() sirve como rutina main() para los threads; cuando run() termina, también lo hace el thread. Todo lo que queramos que haga el thread ha de estar dentro de run(), por eso cuando decimos que un método es Runnable, nos obliga a escribir un método run().

En este ejemplo, intentamos inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada a través del constructor):

```
sleep( retardo );
```

El método sleep() simplemente le dice al thread que duerma durante los milisegundos especificados. Se debería utilizar sleep() cuando se pretenda retrasar la ejecución del thread. sleep() no consume recursos del sistema mientras el thread duerme. De esta forma otros threads pueden seguir funcionando. Una vez hecho el retardo, se imprime el mensaje "Hola Mundo!" con el nombre del thread y el retardo.

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un thread sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un thread de animación, querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de thread se puede utilizar el método suspend().

```
t1.suspend();
```

Este método no detiene la ejecución permanentemente. El thread es suspendido indefinidamente y para volver a activarlo de nuevo necesitamos realizar una invocación al método resume():

```
t1.resume();
```

Parada de un Thread

El último elemento de control que se necesita sobre threads es el método stop(). Se utiliza para terminar la ejecución de un thread:

```
t1.stop();
```

Esta llamada no destruye el thread, sino que detiene su ejecución. La ejecución no se puede reanudar ya con t1.start(). Cuando se desasignen las variables que se usan en el thread, el objeto thread (creado con new) quedará marcado para eliminarlo y el garbage collector se encargará de liberar la memoria que utilizaba.

En nuestro ejemplo, no necesitamos detener explícitamente el thread. Simplemente se le deja terminar. Los programas más complejos necesitarán un control sobre cada uno de los threads que lancen, el método stop() puede utilizarse en esas situaciones.

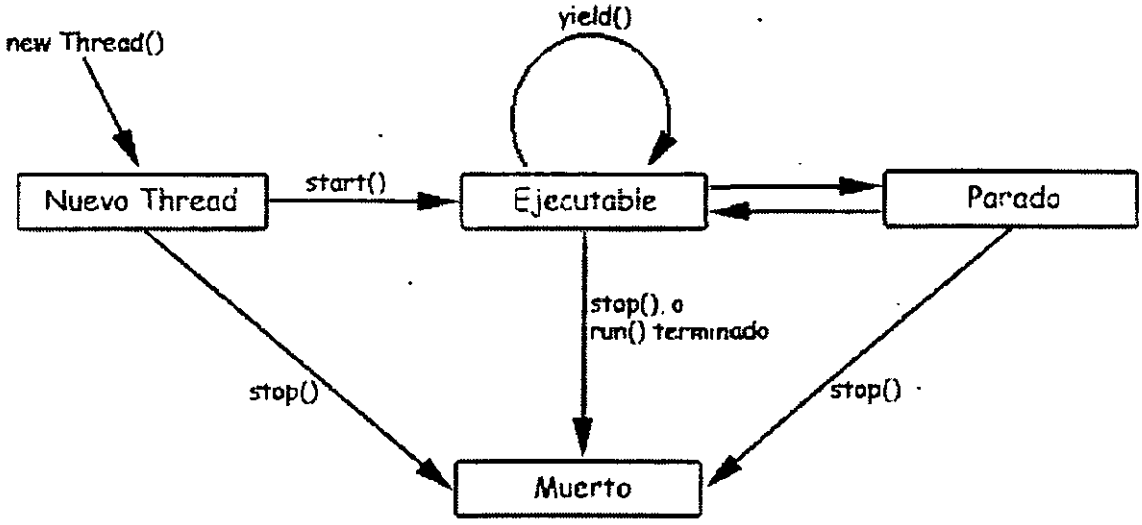
Si se necesita, se puede comprobar si un thread está vivo o no; considerando vivo un thread que ha comenzado y no ha sido detenido.

```
t1.isAlive();
```

Este método devolverá true en caso de que el thread t1 esté vivo, es decir, ya se haya llamado a su método run() y no haya sido parado con un stop() ni haya terminado el método run() en su ejecución.

ESTADOS DE UN THREAD

Durante el ciclo de vida de un thread, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de un thread .



Nuevo Thread

La siguiente sentencia crea un nuevo thread pero no lo arranca, lo deja en el estado de "Nuevo Thread":

```
Thread MiThread = new MiClaseThread();
```

Cuando un thread está en este estado, es simplemente un objeto Thread vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método start(), o detenerse definitivamente, llamando al método stop(); la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo IllegalThreadStateException.

Ejecutable

Ahora veamos las dos línea de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método start() creará los recursos del sistema necesarios para que el thread puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del thread. En este momento nos encontramos en el estado "Ejecutable" del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el thread está aquí no esta corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los threads estén corriendo al mismo tiempo. Java implementa un tipo de scheduling o lista de procesos, que permite que el procesador sea compartido entre todos los procesos o threads que se encuentran en la lista. Sin embargo, para nuestros propósitos, y en la mayoría de

los casos, se puede considerar que este estado es realmente un estado "En Ejecución", porque la impresión que produce ante nosotros es que todos los procesos se ejecutan al mismo tiempo.

Cuando el thread se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método run(), se ejecutarán secuencialmente.

Parado

El thread entra en estado "Parado" cuando alguien llama al método suspend(), cuando se llama al método sleep(), cuando el thread está bloqueado en un proceso de entrada/salida o cuando el thread utiliza su método wait() para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el thread estará Parado.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
```

la línea de código que llama al método sleep():

```
    MiThread.sleep( 10000 );
```

hace que el thread se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread volvería a estar en estado "Ejecutable" y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado Parado, hay una forma específica de volver a estado Ejecutable. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el thread ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado Ejecutable. Llamar al método resume() mientras esté el thread durmiendo no serviría para nada.

Los métodos de recuperación del estado Ejecutable, en función de la forma de llegar al estado Parado del thread, son los siguientes:

Si un thread está dormido, pasado el lapso de tiempo
Si un thread está suspendido, luego de una llamada al método resume()
Si un thread está bloqueado en una entrada/salida, una vez que el comando E/S concluya su ejecución
Si un thread está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse a notify() o notifyAll()

Muerto

Un thread se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). Un thread muere normalmente cuando concluye de forma habitual su método run(). Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
    int i=0;
    while( i < 20 )
        {
        i++;
        System.out.println( "i = "+i );
        }
    }
```

Un thread que contenga a este método run(), morirá naturalmente después de que se complete el bucle y run() concluya.

También se puede matar en cualquier momento un thread, invocando a su método stop(). En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
MiThread.stop();
```

se crea y arranca el thread MiThread, lo dormimos durante 10 segundos y en el momento de despertarse, la llamada a su método stop(), lo mata.

El método stop() envía un objeto ThreadDeath al thread que quiere detener. Así, cuando

un thread es parado de este modo, muere asíncronamente. El thread morirá en el momento en que reciba la excepción ThreadDeath.

Los applets utilizarán el método stop() para matar a todos sus threads cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

El método isAlive()

La interface de programación de la clase Thread incluye el método isAlive(), que devuelve true si el thread ha sido arrancado (con start()) y no ha sido detenido (con stop()). Por ello, si el método isAlive() devuelve false, sabemos que estamos ante un "Nuevo Thread" o ante un thread "Muerto". Si nos devuelve true, sabemos que el thread se encuentra en estado "Ejecutable" o "Parado". No se puede diferenciar entre "Nuevo Thread" y "Muerto", ni entre un thread "Ejecutable" o "Parado".

Java tiene un Scheduler, una lista de procesos, que monitoriza todos los threads que se están ejecutando en todos los programas y decide cuales deben ejecutarse y cuales deben encontrarse preparados para su ejecución. Hay dos características de los threads que el scheduler identifica en este proceso de decisión. Una, la más importante, es la prioridad del thread; la otra, es el indicador de demonio. La regla básica del scheduler es que si solamente hay threads demonio ejecutándose, la Máquina Virtual Java (JVM) concluirá. Los nuevos threads heredan la prioridad y el indicador de demonio de los threads que los han creado. El scheduler determina qué threads deberán ejecutarse
comprobando la prioridad de todos los threads, aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

El scheduler puede seguir dos patrones, preemptivo y no-preemptivo. Los schedulers preemtivos proporcionan un segmento de tiempo a todos los threads que están corriendo en el sistema. El scheduler decide cual será el siguiente thread a ejecutarse y llama a resume() para darle vida durante un período fijo de tiempo. Cuando el thread ha estado en ejecución ese período de tiempo, se llama a suspend() y el siguiente thread en la lista de procesos será relanzado (resume()). Los schedulers no-preemtivos deciden que thread debe correr y lo ejecutan hasta que concluye. El thread tiene control total sobre el sistema mientras esté en ejecución. El método yield() es la forma en que un thread fuerza al scheduler a comenzar la ejecución de otro thread que esté esperando. Dependiendo del sistema en que esté corriendo Java, el scheduler será preemtivo o no-preemptivo.

Prioridades

El scheduler determina el thread que debe ejecutarse en función de la prioridad asignada a cada uno de ellos. El rango de prioridades oscila entre 1 y 10. La prioridad por defecto de un thread es Thread.NORM_PRIORITY, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son Thread.MIN_PRORITY, fijada a 1, y Thread.MAX_PRIORITY, aque tiene un valor de 10. El método getPriority() puede utilizarse para conocer el valor actual de la prioridad de un thread.

Threads Demonio

Los threads demonio también se llaman servicios, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida. Un ejemplo de thread demonio que está ejecutándose continuamente es el recolector de basura (garbage collector). Este thread, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema. Un thread puede fijar su indicador de demonio pasando un valor true al método setDaemon(). Si se pasa false a este método, el thread será devuelto por el sistema como un thread de usuario. No obstante, esto último debe realizarse antes de que se arranque el thread (start()).

Diferencia de threads con fork()

fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si estamos sobrados de memoria y disponemos de una CPU poderosa, y siempre que mantengamos el número de procesos hijos dentro de un límite manejable, porque se hace un uso intensivo de los recursos del sistema. Los applets Java no pueden lanzar ningún proceso en el cliente, porque eso sería una fuente de inseguridad y no está permitido. Las aplicaciones y los applets deben utilizar threads.

La multi-tarea pre-emptiva tiene sus problemas. Un thread puede interrumpir a otro en cualquier momento, de ahí lo de pre-emptive. Imaginarse lo que pasaría si un thread está escribiendo en un array, mientras otro thread lo interrumpe y comienza a escribir en el mismo array. Los lenguajes como C y C++ necesitan de las funciones lock() y unlock() para antes y después de leer o escribir datos. Java también funciona de este modo, pero oculta el bloqueo de datos bajo la sentencia synchronized:

synchronized int MiMetodo();

Otro área en que los threads son muy útiles es en los interfaces de usuario. Permiten incrementar la respuesta del ordenador ante el usuario cuando se encuentra realizando complicados cálculos y no puede atender a la entrada de usuario. Estos cálculos se pueden realizar en segundo plano, o realizar varios en primer plano (música y animaciones) sin que se dé apariencia de pérdida de rendimiento.

Ejemplos

```
class CurrentThreadDemo {
    public static void main(String args[]) {
     Thread t = Thread.currentThread();
     t.setName("Mi Hilo");
     t.setPriority(1);
     System.out.println("Hilo actual: " + t);
     int active = t.activeCount();
     System.out.println("Hilos actuales activos: " + active);
     Thread all[] = new Thread[active];
     t.enumerate(all);
     for (int i = 0; i < active; i++) {
         System.out.println(i + ": " + all[i]);
     }
     try {
         for (int n = 5; n > 0; n--) {
          System.out.println("" + n);
          t.sleep(1000);
         }
     } catch (InterruptedException e) {
         System.out.println("Interumpción");
     }
     t.dumpStack();
    }
}
```

```
class ThreadDemo implements Runnable {
    ThreadDemo() {
     Thread ct = Thread.currentThread();
     Thread t = new Thread(this, "Demo Thread");
     System.out.println("hilo actual: " + ct);
     System.out.println("hilo creado: " + t);
     t.start();
     try {
         Thread.sleep(3000);
     } catch (InterruptedException e) {
         System.out.println("interrupción");
     }
     System.out.println("saliendo del hilo main");
    }
    public void run() {
     try {
         for (int i = 5; i > 0; i--) {
          System.out.println("" + i);
          Thread.sleep(1000);
         }
     } catch (InterruptedException e) {
         System.out.println("Interrumpiendo al hijo");
     }
     System.out.println("saliendo del hilo hijo");
    }
    public static void main(String args[]) {
     new ThreadDemo();
    }
}
```

```
class clicker implements Runnable {
    int click = 0;
    private Thread t;
    private boolean running = true;
    public clicker(int p) {
     t = new Thread(this);
     t.setPriority(p);
    }
    public void run() {
     while (running) {
         click++;
     }
    }
    public void stop() {
     running = false;
    }
    public void start() {
     t.start();
    }
}

class HiLoPri {
    public static void main(String args[]) {
     Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
     clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
     clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
     lo.start();
     hi.start();
     try Thread.sleep(10000); catch (Exception e);
     lo.stop();
     hi.stop();
     System.out.println(lo.click + " vs. " + hi.click);
    }
}
```

FLUJOS DE DATOS

Los usuarios de Unix, y aquellos familiarizados con las líneas de comandos de otros sistemas como DOS, han utilizado un tipo de entrada/salida conocida comúnmente por entrada/salida estándar. El fichero de entrada estándar (stdin) es simplemente el teclado. El fichero de salida estándar (stdout) es típicamente la pantalla (o la ventana del terminal). El fichero de salida de error estándar (stderr) también se dirige normalmente a la pantalla, pero se implementa como otro fichero de forma que se pueda distinguir entre la salida normal y (si es necesario) los mensajes de error.

La clase System

Java tiene acceso a la entrada/salida estándar a través de la clase System. En concreto, los tres ficheros que se implementan son:

Stdin

System.in implementa stdin como una instancia de la clase `InputStream`. Con `System.in`, se accede a los métodos `read()` y `skip()`. El método `read()` permite leer un byte de la entrada. `skip( long n )`, salta n bytes de la entrada.

Stdout

System.out implementa stdout como una instancia de la clase PrintStream. Se pueden utilizar los métodos `print()` y `println()` con cualquier tipo básico Java como argumento.

Stderr

System.err implementa stderr de la misma forma que stdout. Como con System.out, se tiene acceso a los métodos de PrintStream.

Vamos a ver un pequeño ejemplo de entrada/salida en Java. El código siguiente, miType.java, reproduce, o funciona como la utilidad cat de Unix o type de DOS:

```
import java.io.*;

class miType {
          public static void main( String args[] ) throws
IOException {
          int c;
          int contador = 0;

          while( (c = System.in.read() ) != '\n' )
               {
               contador++;
               System.out.print( (char)c );
               }
          System.out.println();     // Línea en blanco
          System.err.println( "Contados "+ contador +" bytes en
total." );
          }
     }
```

Clases comunes de Entrada/Salida

Además de la entrada por teclado y salida por pantalla, se necesita entrada/salida por fichero, como son:

```
FileInputStream
DataInputStream
FileOutputStream
DataOutputStream
```

También existen otras clases para aplicaciones más específicas, que no vamos a tratar, por ser de un uso muy concreto:

```
PipedInputStream
BufferedInputStream
PushBackInputStream
StreamTokenizer
PipedOutputStream
BufferedOutputStream
```

Todos los lenguajes de programación tienen alguna forma de interactuar con los sistemas de ficheros locales; Java no es una excepción.

Cuando se desarrollan applets para utilizar en red, hay que tener en cuenta que la entrada/salida directa a fichero es una violación de seguridad de acceso. Muchos usuarios configurarán sus navegadores para permitir el acceso al sistema de ficheros, pero otros no.

Por otro lado, si se está desarrollando una aplicación Java para uso interno, probablemente será necesario el acceso directo a ficheros.

Ficheros

Antes de realizar acciones sobre un fichero, necesitamos un poco de información sobre ese fichero. La clase File proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros.

Creación de un objeto File

Para crear un objeto File nuevo, se puede utilizar cualquiera de los tres constructores siguientes:

```
File miFichero;
miFichero = new File( "/etc/kk" );
```

o

```
miFichero = new File( "/etc","kk" );
```

o

```
File miDirectorio = new File( "/etc" );
miFichero = new File( miDirectorio,"kk" );
```

El constructor utilizado depende a menudo de otros objetos File necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si en cambio, se utilizan muchos ficheros desde un mismo directorio, el segundo o tercer constructor serán más cómodos. Y si el directorio o el fichero es una variable, el segundo constructor será el más útil.

Comprobaciones y Utilidades

Una vez creado un objeto File, se puede utilizar uno de los siguientes métodos para reunir información sobre el fichero:

Nombres de fichero

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
boolean renameTo( File nuevoNombre )
```

Comprobaciones

```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute()
```

Información general del fichero

```
long lastModified()
long length()
```

Utilidades de directorio

```
boolean mkdir()
String[] list()
```

Vamos a desarrollar una pequeña aplicación que muestra información sobre los ficheros pasados como argumentos en la línea de comandos, InfoFichero.java:

```
import java.io.*;

class InfoFichero {

    public static void main( String args[] ) throws IOException {
        if( args.length > 0 )
            {
            for( int i=0; i < args.length; i++ )
                {
                File f = new File( args[i] );
                System.out.println( "Nombre: "+f.getName() );
                System.out.println( "Camino: "+f.getPath() );
                if( f.exists() )
                    {
                    System.out.print( "Fichero existente " );
                    System.out.print( (f.canRead() ?
                        " y se puede Leer" : "" ) );
                    System.out.print( (f.canWrite() ?
                        " y se puese Escribir" : "" ) );
                    System.out.println( "." );
                    System.out.println( "La longitud del fichero
son "+
                        f.length()+" bytes" );
                    }
                else
                    System.out.println( "El fichero no existe." );
                }
            }
        else
            System.out.println( "Debe indicar un fichero." );
        }
    }
```

```
import java.io.*;

class InfoFichero {

    public static void main( String args[] ) throws IOException {
        if( args.length > 0 )
            {
            for( int i=0; i < args.length; i++ )
                {
                File f = new File( args[i] );
                System.out.println( "Nombre: "+f.getName() );
                System.out.println( "Camino: "+f.getPath() );
                if( f.exists() )
                    {
                    System.out.print( "Fichero existente " );
                    System.out.print( (f.canRead() ?
                        " y se puede Leer" : "" ) );
                    System.out.print( (f.canWrite() ?
                        " y se puese Escribir" : "" ) );
                    System.out.println( "." );
                     System.out.println( "La longitud del fichero
son "+
                        f.length()+" bytes" );
                    }
                else
                   System.out.println( "El fichero no existe." );
                }
            }
        else
            System.out.println( "Debe indicar un fichero." );
        }
    }
```

STREAMS DE ENTRADA

Hay muchas clases dedicadas a la obtención de entrada desde un fichero. Este es el esquema de la jerarquía de clases de entrada por fichero:

Objetos FileInputStream

Los objetos FileInputStream típicamente representan ficheros de texto accedidos en orden secuencial, byte a byte. Con FileInputStream, se puede elegir acceder a un byte, varios bytes o al fichero completo.

Apertura de un FileInputStream

Para abrir un FileInputStream sobre un fichero, se le da al constructor un String o un objeto File:

```
FileInputStream mi FicheroSt;
  miFicheroSt = new FileInputStream( "/etc/kk" );
```

También se puede utilizar:

```
File miFichero; FileInputStream miFicheroSt;
miFichero = new File( "/etc/kk" );
miFicheroSt = new FileInputStream( miFichero );
```

Lectura de un FileInputStream

Una vez abierto el FileInputStream, se puede leer de él. El método read() tiene muchas opciones:

```
int read();
```

Lee un byte y devuelve -1 al final del stream.

```
int read( byte b[] );
```

Llena todo el array, si es posible. Devuelve el número de bytes leídos o -1 si se alcanzó el final del stream.

```
int read( byte b[],int offset,int longitud );
```

Lee longitud bytes en b comenzando por b[offset]. Devuelve el número de bytes leídos o -1 si se alcanzó el final del stream.

Cierre de FileInputStream

Cuando se termina con un fichero, existen dos opciones para cerrarlo: explícitamente, o implícitamente cuando se recicla el objeto (el garbage collector se encarga de ello).

Para cerrarlo explícitamente, se utiliza el método close():

```
miFicheroSt.close();
```

Ejemplo: Visualización de un fichero

Si la configuración de la seguridad de Java permite el acceso a ficheros, se puede ver el contenido de un fichero en un objeto TextArea. El código siguiente contiene los elementos necesarios para mostrar un fichero:

```
FileInputStream fis;
TextArea ta;

public void init() {
    byte b[] = new byte[1024];
    int i;

     // El buffer de lectura se debe hacer lo suficientemente
grande
    // o esperar a saber el tamaño del fichero
    String s;

    try {
        fis = new FileInputStream( "/etc/kk" );
    } catch( FileNotFoundException e ) {
        /* Hacer algo */
        }

    try {
        i = fis.read( b );
    } catch( IOException e ) {
        /* Hacer algo */
        }

    s = new String( b,0 );
    ta = new TextArea( s,5,40 );
    add( ta );
    }
```

Hemos desarrollado un ejemplo, Agenda.java, en el que partimos de un fichero agenda que dispone de los datos que nosotros deseamos de nuestros amigos, como son: nombre, teléfono y dirección. Si tecleamos un nombre, buscará en el fichero de datos si existe ese nombre y presentará la información que se haya introducido. Para probar, intentar que aparezca la información de Pepe.

Objetos DataInputStream

Los objetos DataInputStream se comportan como los FileInputStream. Los streams de datos pueden leer cualquiera de las variables de tipo nativo, como floats, ints o chars. Generalmente se utilizan DataInputStream con ficheros binarios.

Apertura y cierre de DataInputStream

Para abrir y cerrar un objeto DataInputStream, se utilizan los mismos métodos que para FileInputStream:

```
DataInputStream miDStream;
FileInputStream miFStream;

// Obtiene un controlador de fichero
miFStream = new FileInputStream "/etc/ejemplo.dbf" );
//Encadena un fichero de entrada de datos
miDStream = new DataInputStream( miFStream );

// Ahora se pueden utilizar los dos streams de entrada para
// acceder al fichero (si se quiere...)
miFStream.read( b );
i = miDStream.readInt();

// Cierra el fichero de datos explícitamente
//Siempre se cierra primero el fichero stream de mayor nivel
miDStream.close();
miFStream.close();
```

Lectura de un DataInputStream

Al acceder a un fichero como DataInputStream, se pueden utilizar los mismos métodos read() de los objetos FileInputStream.

No obstante, también se tiene acceso a otros métodos diseñados para leer cada uno de los tipos de datos:

```
byte readByte()
int readUnsignedByte()
short readShort()
int readUnsignedShort()
char readChar()
int readInt()
long readLong()
float readFloat()
double readDouble()
String readLine()
```

Cada método leerá un objeto del tipo pedido.

Para el método String readLine(), se marca el final de la cadena con \n, \r, \r\n o con EOF.

Para leer un long, por ejemplo:

```
long numeroSerie;
...
numeroSerie = miDStream.readLong();
```

Streams de entrada de URLs

Además del acceso a ficheros, Java proporciona la posibilidad de acceder a URLs como una forma de acceder a objetos a través de la red. Se utiliza implícitamente un objeto URL al acceder a sonidos e imágenes, con el método getDocumentBase() en los applets:

```
String imagenFich = new String( "imagenes/pepe.gif" );
imagenes[0] = getImage( getDocumentBase(),imagenFich );
```

No obstante, se puede proporcionar directamente un URL, si se quiere:

```
URL imagenSrc;
imagenSrc = new URL( "http://enterprise.com/~info" );
imagenes[0] = getImage( imagenSrc,"imagenes/pepe.gif" );
```

Apertura de un Stream de entrada de URL.

También se puede abrir un stream de entrada a partir de un URL. Por ejemplo, se puede utilizar un fichero de datos para un applet:

```
InputStream is;
byte buffer[] = new byte[24];
is = new URL( getDocumentBase(),datos).openStream();
```

Ahora se puede utilizar is para leer información de la misma forma que se hace con un objeto FileInputStream:

```
is.read( buffer,0,buffer.length );
```

NOTA: Debe tenerse muy en cuenta que algunos usuarios pueden haber configurado la seguridad de sus navegadores para que los applets no accedan a ficheros.

STREAMS DE SALIDA

La contrapartida necesaria de la lectura de datos es la escritura de datos. Como con los Streams de entrada, las clases de salida están ordenadas jerárquicamente:

Examinaremos las clases FileOutputStream y DataOutputStream para complementar los streams de entrada que se han visto. En los ficheros fuente del directorio $JAVA_HOME/src/java/io se puede ver el uso y métodos de estas clases, así como de los streams de entrada ($JAVA_HOME es el directorio donde se haya instalado el Java Development Kit, en sistemas UNIX).

Objetos FileOutputStream

Los objetos FileOutputStream son útiles para la escritura de ficheros de texto. Como con los ficheros de entrada, primero se necesita abrir el fichero para luego escribir en él.

Apertura de un FileOutputStream

Para abrir un objeto FileOutputStream, se tienen las mismas posibilidades que para abrir un fichero stream de entrada. Se le da al constructor un String o un objeto File.

```
FileOutputStream miFicheroSt;
  miFicheroSt = new FileOutputStream( "/etc/kk" );
```

Como con los streams de entrada, también se puede utilizar:

```
File miFichero; FileOutputStream miFicheroSt;

    miFichero = new File( "/etc/kk" );
    miFicheroSt = new FileOutputStream( miFichero );
```

Escritura en un FileOutputStream

Una vez abierto el fichero, se pueden escribir bytes de datos utilizando el método write(). Como con el método read() de los streams de entrada, tenemos tres posibilidades:

```
void write( int b );
```

Escribe un byte.

```
void write( byte b[] );
```

Escribe todo el array, si es posible.

```
void write( byte b[],int offset,int longitud );
```

Escribe longitud bytes en b comenzando por b[offset].

Cierre de FileOutputStream

Cerrar un stream de salida es similar a cerrar streams de entrada. Se puede utilizar el método explícito:

```
miFicheroSt.close();
```

O, se puede dejar que el sistema cierre el fichero cuando se recicle miFicheroSt.

Ejemplo: Almacenamiento de Información

Este programa, Telefonos.java, pregunta al usuario una lista de nombres y números de teléfono. Cada nombre y número se añade a un fichero situado en una localización fija. Para indicar que se ha introducido toda la lista, el usuario especifica "Fin" ante la solicitud de entrada del nombre.

Una vez que el usuario ha terminado de teclear la lista, el programa creará un fichero de salida que se mostrará en pantalla o se imprimirá. Por ejemplo:

```
95-4751232,Juanito
564878,Luisa
123456,Pepe
347698,Antonio
91-3547621,Maria
```

El código fuente del programa es el siguiente:

```
import java.io.*;

class Telefonos {
    static FileOutputStream fos;
    public static final int longLinea = 81;

    public static void main( String args[] ) throws IOException {
        byte tfno[] = new byte[longLinea];
        byte nombre[] = new byte[longLinea];

        fos = new FileOutputStream( "telefono.dat" );
        while( true )
            {
            System.err.println( "Teclee un nombre ('Fin' termina)"
);
            leeLinea( nombre );
            if( "fin".equalsIgnoreCase( new String( nombre,0,0,3 )
) )
                break;

            System.err.println( "Teclee el numero de telefono" );
            leeLinea( tfno );
            for( int i=0; tfno[i] != 0; i++ )
                fos.write( tfno[i] );
            fos.write( ',' );
            for( int i=0; nombre[i] != 0; i++ )
                fos.write( nombre[i] );
            fos.write( '\n' );
            }
        fos.close();
        }

    private static void leeLinea( byte linea[] ) throws IOException
{
        int b = 0;
        int i = 0;

        while( (i < ( longLinea-1) ) &&
            ( ( b = System.in.read() ) != '\n' ) )
            linea[i++] = (byte)b;
        linea[i] = (byte)0;
        } }
```

Streams de salida con buffer

Si se trabaja con gran cantidad de datos, o se escriben muchos elementos pequeños, será una buena idea utilizar un stream de salida con buffer. Los streams con buffer ofrecen los mismos métodos de la clase FileOutputStream, pero toda salida se almacena en un buffer. Cuando se llena el buffer, se envía a disco con una única operación de escritura; o, en caso necesario, se puede enviar el buffer a disco en cualquier momento.

Creación de Streams de salida con buffer

Para crear un stream BufferedOutput, primero se necesita un stream FileOutput normal; entonces se le añade un buffer al stream:

```
FileOutputStream miFileStream;
BufferdOutpurStream miBufferStream;
// Obtiene un controlador de fichero
miFileStream = new FileOutputStream( "/tmp/kk" );
// Encadena un stream de salida con buffer
miBufferStream = new BufferedOutputStream( miFileStream );
```

Volcado y Cierre de Streams de salida con buffer

Al contrario que los streams FileOutput, cada escritura al buffer no se corresponde con una escritura en disco. A menos que se llene el buffer antes de que termine el programa, cuando se quiera volcar el buffer explícitamente se debe hacer mediante una llamada a flush():

```
// Se fuerza el volcado del buffer a disco
miBufferStream.flush();
  // Cerramos el fichero de datos. Siempre se ha de cerrar
primero el
// fichero stream de mayor nivel
miBufferStream.close();
miFileStream.close();
```

Streams DataOutput

Java también implementa una clase de salida complementaria a la clase DataInputStream. Con la clase DataOutputStream, se pueden escribir datos binarios en un fichero.

Apertura y cierre de objetos DataOutputStream

Para abrir y cerrar objetos DataOutputStream, se utilizan los mismos métodos que para los objetos FileOutputStream:

```
DataOutputStream miDataStream;
FileOutputStream miFileStream;
BufferedOutputStream miBufferStream;

// Obtiene un controlador de fichero
miFileStream = new FileOutputStream( "/tmp/kk" );
// Encadena un stream de salida con buffer (por eficiencia)
miBufferStream = new BufferedOutputStream( miFileStream );
// Encadena un fichero de salida de datos
miDataStream = new DataOutputStream( miBufferStream );

// Ahora se pueden utilizar los dos streams de entrada para
// acceder al fichero (si se quiere)
miBufferStream.write( b );
miDataStream.writeInt( i );

// Cierra el fichero de datos explícitamente. Siempre se cierra
// primero el fichero stream de mayor nivel
miDataStream.close();
miBufferStream.close();
miFileStream.close();
```

Escritura en un objeto DataOutputStream

Cada uno de los métodos write() accesibles por los FileOutputStream también lo son a través de los DataOutputStream.

También encontrará métodos complementarios a los de DataInputStream:

```
void writeBoolean( boolean b );
void writeByte( int i );
void writeShort( int i );
void writeChar( int i );
void writeInt( int i );
void writeFloat( float f );
void writeDouble( double d );
void writeBytes( String s );
void writeChars( string s );
```

Para las cadenas, se tienen dos posibilidades: bytes y caracteres. Hay que recordar que los bytes son objetos de 8 bits y los caracteres lo son de 16 bits. Si nuestras cadenas utilizan caracteres Unicode, debemos escribirlas con writeChars().

Contabilidad de la salida

Otra función necesaria durante la salida es el método size(). Este método simplemente devuelve el número total de bytes escritos en el fichero. Se puede utilizar size() para ajustar el tamaño de un fichero a múltiplo de cuatro. Por ejemplo, de la forma siguiente:

```
. . .
int numBytes = miDataStream.size() % 4;
for( int i=0; i < numBytes; i++ )
    miDataStream.write( 0 );
. . .
```

FICHEROS DE ACCESO ALEATORIO

A menudo, no se desea leer un fichero de principio a fin; sino acceder al fichero como una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero. Java proporciona una clase RandomAccessFile para este tipo de entrada/salida.

Creación de un Fichero de Acceso Aleatorio

Hay dos posibilidades para abrir un fichero de acceso aleatorio:

Con el nombre del fichero:

```
miRAFile = new RandomAccessFile( String nombre,String modo );
```

Con un objeto File:

```
miRAFile = new RandomAccessFile( File fichero,String modo );
```

El argumento modo determina si se tiene acceso de sólo lectura (r) o de lectura/escritura (r/w). Por ejemplo, se puede abrir un fichero de una base de datos para actualización:

```
RandomAccessFile miRAFile;
miRAFile = new RandomAccessFile( "/tmp/kk.dbf","rw" );
```

Acceso a la Información

Los objetos RandomAccessFile esperan información de lectura/escritura de la misma manera que los objetos DataInput/DataOutput. Se tiene acceso a todas las operaciones read() y write() de las clases DataInputStream y DataOutputStream.

También se tienen muchos métodos para moverse dentro de un fichero:

```
long getFilePointer();
```

Devuelve la posición actual del puntero del fichero

```
void seek( long pos );
```

Coloca el puntero del fichero en una posición determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 marca el comienzo de ese fichero.

```
long length();
```

Devuelve la longitud del fichero. La posición length() marca el final de ese fichero.

Actualización de Información

Se pueden utilizar ficheros de acceso aleatorio para añadir información a ficheros existentes:

```
    miRAFile = new RandomAccessFile( "/tmp/kk.log","rw" );
    miRAFile.seek( miRAFile.length() );
     // Cualquier write() que hagamos a partir de este punto del
código
    // añadirá información al fichero
```

Vamos a ver un pequeño ejemplo, Log.java, que añade una cadena a un fichero existente:

```
import java.io.*;

// Cada vez que ejecutemos este programita, se incorporara una
nueva
// linea al fichero de log que se crea la primera vez que se
ejecuta
//
class Log {
    public static void main( String args[] ) throws IOException {
        RandomAccessFile miRAFile;
        String s = "Informacion a incorporar\nTutorial de Java\n";

        // Abrimos el fichero de acceso aleatorio
        miRAFile = new RandomAccessFile( "/tmp/java.log","rw" );
        // Nos vamos al final del fichero
        miRAFile.seek( miRAFile.length() );
        // Incorporamos la cadena al fichero
```

```
        miRAFile.writeBytes( s );
        // Cerramos el fichero
        miRAFile.close();
        }
    }
```

COMUNICACIONES EN UNIX

El sistema de Entrada/Salida de Unix sigue el paradigma que normalmente se designa como Abrir-Leer-Escribir-Cerrar.

Antes de que un proceso de usuario pueda realizar operaciones de entrada/salida, debe hacer una llamada a Abrir (open) para indicar, y obtener permisos para su uso, el fichero o dispositivo que quiere utilizar. Una vez que el objeto está abierto, el proceso de usuario realiza una o varias llamadas a Leer (read) y Escribir (write), para conseguir leer y escribir datos. Leer coge datos desde el objeto y los transfiere al proceso de usuario, mientras que Escribir transfiere datos desde el proceso de usuario al objeto. Una vez que todos estos intercambios de información estén concluidos, el proceso de usuario llamará a Cerrar (close) para informar al sistema operativo que ha finalizado la utilización del objeto que antes había abierto.

Cuando se incorporan las características a Unix de comunicación entre procesos (IPC) y el manejo de redes, la idea fue implementar la interface con IPC similar a la que se estaba utilizando para la entrada/salida de ficheros, es decir, siguiendo el paradigma del párrafo anterior. En Unix, un proceso tiene un conjunto de descriptores de entrada/salida desde donde Leer y por donde Escribir. Estos descriptores pueden estar referidos a ficheros, dispositivos, o canales de comunicaciones (sockets). El ciclo de vida de un descriptor, aplicado a un canal de comunicación (socket), está determinado por tres fases (siguiendo el paradigma):

Creación, apertura del socket
Lectura y Escritura, recepción y envío de datospor el socket
Destrucción, cierre del socket

La interface IPC en Unix-BSD está implementada sobre los protocolos de red TCP y UDP. Los destinatarios de los mensajes se especifican como direcciones de socket: cada dirección de socket es un identificador de comunicación que consiste en una dirección Internet y un número de puerto.

Las operaciones IPC se basan en pares de sockets. Se intercambian información transmitiendo datos a través de mensajes que circulan entre un socket en un proceso y otro socket en otro proceso. Cuando los mensajes son enviados, se encolan en el socket hasta que el protocolo de red los haya transmitido. Cuando llegan, los mensajes son encolados en el socket de recepción hasta que el proceso que tiene que recibirlos haga las llamadas necesarias para recoger esos datos.

SOCKETS

Los sockets son puntos finales de enlaces 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.

El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.

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.

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.

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.

Diferencias entre Sockets Stream y Datagrama

Ahora se nos presenta un problema. ¿qué protocolo, o tipo de sockets, debemos usar - UDP o TCP? La decisión depende

de la aplicación cliente/servidor que estemos escribiendo. Vamos a ver algunas diferencias entre los protocolos para ayudar en
la decisión.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket
que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión,
tenemos que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el
establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización
determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los
streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden
por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen
serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el
usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo,
cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los
datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y
transmisión de ficheros (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor
sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas
distribuidos montados sobre redes de área local.

USO DE SOCKETS

Podemos pensar que un Servidor Internet es un conjunto de sockets que proporciona capacidades adicionales del sistema,
los llamados servicios.

Puertos y Servicios

Cada servicio está asociado a un puerto. Un puerto es una dirección numérica a través de la cual se procesa el servicio.
Sobre un sistema Unix, los servicios que proporciona ese sistema se indican en el fichero /etc/services, y algunos ejemplos
son:

```
daytime    13/udp
ftp        21/tcp
telnet     23/tcp     telnet
smtp       25/tcp     mail
http       80/tcp
```

La primera columna indica el nombre del servicio. La segunda columna indica el puerto y el protocolo que está asociado al
servicio. La tercera columna es un alias del servicio; por ejemplo, el servicio smtp, también conocido como mail, es la
implementación del servicio de correo electrónico.

Las comunicaciones de información relacionada con Web tienen lugar a través del puerto 80 mediante protocolo TCP. Para
emular esto en Java, usaremos la clase Socket. La fecha (daytime). Sin embargo, el servicio que coge la fecha y la hora del
sistema, está ligado al puerto 13 utilizando el protocolo UDP. Un servidor que lo emule en Java usaría un objeto
DatagramSocket.

LA CLASE URL

La clase URL contiene contructores y métodos para la manipulación de URL (Universal Resource Locator): un objeto o
servicio en Internet. El protocolo TCP necesita dos tipos de información: la dirección IP y el número de puerto. Vamos a ver
como podemos recibir pues la página Web principal de nuestro buscador favorito al teclear:

```
http://www.yahoo.com
```

En primer lugar, Yahoo tiene registrado su nombre, permitiendo que se use yahoo.com como su dirección IP, o lo que es lo
mismo, cuando indicamos yahoo.com es como si hubiesemos indicado 205.216.146.71, su dirección IP real.

La verdad es que la cosa es un poco más complicada que eso. Hay un servicio, el DNS (Domain Name Service), que traslada
www.yahoo.com a 205.216.146.71, lo que nos permite teclear www.yahoo.com, en lugar de tener que recordar su
dirección IP.

Si queremos obtener la dirección IP real de la red en que estamos corriendo, podemos realizar llamadas a los métodos
getLocalHost() y getAddress(). Primero, getLocalHost() nos devuelve un objeto iNetAddress, que si usamos con
getAddress() generará un array con los cuatro bytes de la dirección IP, por ejemplo:

```
InetAddress direccion = InetAddress.getLocalHost();
byte direccionIp[] = direccion.getAddress();
```

Si la dirección de la máquina en que estamos corriendo es 150.150.112.145, entonces:

```
direccionIp[0] = 150
direccionIp[1] = 150
direccionIp[2] = 112
direccionIp[3] = 145
```

Una cosa interesante en este punto es que una red puede mapear muchas direcciones IP. Esto puede ser necesario para un
Servidor Web, como Yahoo, que tiene que soportar grandes cantidades de tráfico y necesita más de una dirección IP para
poder atender a todo ese tráfico. El nombre interno para la dirección 205.216.146.71, por ejemplo, es www7.yahoo.com. El
DNS puede trasladar una lista de direcciones IP asignadas a Yahoo en www.yahoo.com. Esto es una cualidad útil, pero por
ahora abre un agujero en cuestión de seguridad.

Ya conocemos la dirección IP, nos falta el número del puerto. Si no se indica nada, se utilizará el que se haya definido por
defecto en el fichero de configuración de los servicios del sistema. En Unix se indican en el fichero /etc/services, en
Windows-NT en el fichero services y en otros sistemas puede ser diferente.

El puerto habitual de los servicios Web es el 80, así que si no indicamos nada, entraremos en

el servidor de Yahoo por el
puerto 80. Si tecleamos la URL siguiente en un navegador:

http://www.yahoo.com:80

también recibiremos la página principal de Yahoo. No hay nada que nos impida cambiar el puerto en el que residirá el servidor
Web; sin embargo, el uso del puerto 80 es casi estándar, porque elimina pulsaciones en el teclado y, además, las direcciones
URL son lo suficientemente difíciles de recordar como para añadirle encima el número del puerto.

Si necesitamos otro protocolo, como:

ftp://ftp.microsoft.com

el puerto se derivará de ese protocolo. Así el puerto FTP de Microsoft es el 21, según su fichero services. La primera
parte, antes de los dos puntos, de la URL, indica el protocolo que se quiere utilizar en la conexión con el servidor. El
protocolo http (HyperText Transmission Protocol), es el utilizado para manipular documentos Web. Y si no se especifica
ningún documento, muchos servidores están configurados para devolver un documento de nombre index.html.

Con todo esto, Java permite los siguientes cuatro constructores para la clase URL:

```
public URL( String spec ) throws MalformedURLException;
public URL( String protocol,String host,int port,String file ) throws MalformedURLException;
public URL( String protocol,String host,String file ) throws MalformedURLException;
public URL( URL context,String spec ) throws MalformedURLException;
```

Así que podríamos especificar todos los componenetes del URL como en:

```
URL( "http","www.yahoo.com","80","index.html" );
```

o dejar que los sistemas utilicen todos los valores por defecto que tienen definidos, como en:

```
URL( "http://www.yahoo.com" );
```

y en los dos casos obtendríamos la visualización de la página principal de Yahoo en nuestro navegador.

MODELO DE COMUNICACIONES CON JAVA

En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete java.net. A continuación mostramos un
diagrama de lo que ocurre en el lado del cliente y del servidor:

El modelo de sockets más simple es:

El servidor establece un puerto y espera durante un cierto tiempo (timeout segundos), a que el cliente establezca la
conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método accept().
El cliente establece una conexión con la máquina host a través del puerto que se designe en puerto#
El cliente y el servidor se comunican con manejadores InputStream y OutputStream

Hay una cuestión al respecto de los sockets, que viene impuesta por la implementación del sistema de seguridad de Java.
Actualmente, los applets sólo pueden establecer conexiones con el nodo desde el cual se transfirió su código. Esto está
implementado en el JDK y en el intérprete de Java de Netscape. Esto reduce en gran manera la flexibilidad de las fuentes de
datos disponibles para los applets. El problema si se permite que un applet se conecte a cualquier máquina de la red, es que
entonces se podrían utilizar los applets para inundar la red desde un ordenador con un cliente Netscape del que no se
sospecha y sin ninguna posibilidad de rastreo.

APERTURA DE SOCKETS

Si estamos programando un cliente, el socket se abre de la forma:

```
Socket miCliente;
miCliente = new Socket( "maquina",numeroPuerto );
```

Donde maquina es el nombre de la máquina en donde estamos intentando abrir la conexión y numeroPuerto es el puerto (un
número) del servidor que está corriendo sobre el cual nos queremos conectar. Cuando se selecciona un número de puerto, se
debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios
(superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. Para las
aplicaciones que se desarrollen, asegurarse de seleccionar un puerto por encima del 1023.

En el ejemplo anterior no se usan excepciones; sin embargo, es una gran idea la captura de excepciones cuando se está
trabajando con sockets. El mismo ejemplo quedaría como:

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

Si estamos programando un servidor, la forma de apertura del socket es la que muestra el siguiente ejemplo:

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

A la hora de la implementación de un servidor también necesitamos crear un objeto socket desde el ServerSocket para que
esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

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

CREACION DE STREAMS

Creación de Streams de Entrada

En la parte cliente de la aplicación, se puede utilizar la clase DataInputStream para crear un stream de entrada que esté listo
a recibir todas las respuestas que el servidor le envíe.

```
DataInputStream entrada;
try {
```

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

La clase DataInputStream permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente
portable; dispone de métodos para leer todos esos tipos como: read(), readChar(), readInt(), readDouble() y readLine().
Deberemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperemos recibir del servidor.

En el lado del servidor, también usaremos DataInputStream, pero en este caso para recibir las entradas que se produzcan
de los clientes que se hayan conectado:

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

Creación de Streams de Salida

En el lado del cliente, podemos crear un stream de salida para enviar información al socket del servidor utilizando las clases
PrintStream o DataOutputStream:

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

La clase PrintStream tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos write
y println() tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también
podemos utilizar DataOutputStream:

```
DataOutputStream salida;
```

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

La clase DataOutputStream permite escribir cualquiera de los tipos primitivos de Java. muchos de sus métodos escriben un
tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea writeBytes().

En el lado del servidor, podemos utilizar la clase PrintStream para enviar información al cliente:

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

Pero también podemos utilizar la clase DataOutputStream como en el caso de envío de información desde el cliente.

CIERRE DE SOCKETS

Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la
parte del cliente:

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

Y en la parte del servidor:

```
try {
    salida.close();
    entrada.close();
```

```
        socketServicio.close();
        miServicio.close();
    }catch( IOException e ) {
        System.out.println( e );
        }
```

En el siguiente ejemplo, vamos a desarrollar un servidor similar al que se ejecuta sobre el puerto 7 de las máquinas Unix, el
servidor echo. Básicamente, este servidor recibe texto desde un cliente y reenvía ese mismo texto al cliente. Desde luego, este
es el servidor más simple de los simples que se pueden escribir. El ejemplo que presentamos, ecoServidor.java, maneja
solamente un cliente. Una modificación interesante sería adecuarlo para que aceptase múltiples clientes simultáneos mediante el
uso de threads.

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

class ecoServidor {
    public static void main( String args[] ) {
        ServerSocket s = null;
        DataInputStream sIn;
        PrintStream sOut;
        Socket cliente = null;
        String texto;

        // Abrimos una conexión con breogan 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 ) {
            }

        // Creamos el objeto desde el cual atenderemos y aceptaremos
        // las conexiones de los clientes y abrimos los canales de
        // comunicación de entrada y salida
        try {
            cliente = s.accept();
            sIn = new DataInputStream( cliente.getInputStream() );
            sOut = new PrintStream( cliente.getOutputStream() );

            // Cuando recibamos datos, se los devolvemos al cliente
```

```
            // que los haya enviado
            while( true )
                {
                texto = sIn.readLine();
                sOut.println( texto );
                }
        }catch( IOException e ) {
            System.out.println( e );
            }
        }
}
```

CLASES UTILES EN COMUNICACIONES

Vamos a exponer otras clases que resultan útiles cuando estamos desarrollando programas de comunicaciones, aparte de las
que ya se han visto. El problema es que la mayoría de estas clases se prestan a discusión, porque se encuentran bajo el
directorio sun. Esto quiere decir qué son implementaciones Solaris y, por tanto, específicas del Unix Solaris. Además su API
no está garantizada, pudiendo cambiar. Pero, a pesar de todo, resultan muy interesantes y vamos a comentar un grupo de ellas
solamente que se encuentran en el paquete sun.net

Socket

Es el objeto básico en toda comunicación a través de Internet, bajo el protocolo TCP. Esta clase proporciona
métodos para la entrada/salida a través de streams que hacen la lectura y escritura a través de sockets muy
sencilla.

ServerSocket

Es un objeto utilizado en las aplicaciones servidor para escuchar las peticiones que realicen los clientes
conectados a ese servidor. Este objeto no realiza el servicio, sino que crea un objeto Socket en función del
cliente para realizar toda la comunicación a través de él.

DatagramSocket

La clase de sockets datagrama puede ser utilizada para implementar datagramas no fiables (sockets UDP), no ordenados. Aunque la comunicación por estos sockets es muy rápida porque no hay que perder tiempo estableciendo la conexión entre cliente y servidor.

DatagramPacket

Clase que representa un paquete datagrama conteniendo información de paquete. longitud de paquete, direcciones Internet y números de puerto.

MulticastSocket

Clase utilizada para crear una versión multicast de las clase socket datagrama. Múltiples clientes/servidores pueden transmitir a un grupo multicast (un grupo de direcciones IP compartiendo el mismo número de puerto).

NetworkServer

Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.

NetworkClient

Una clase creada para implementar métodos y variables utilizadas en la creación de un cliente TCP/IP.

SocketImpl

Es un Interface que nos permite crearnos nuestro propio modelo de comunicación. Tendremos que implementar sus métodos cuando la usemos. Si vamos a desarrollar una aplicación con requerimientos especiales de comunicaciones. como pueden se la implementación de un cortafuegos (TCP es un protocolo no seguro). o acceder a equipos especiales (como un lector de código de barras o un GPS diferencial). necesitaremos nuestra propia clase Socket.

BIBLIOGRAFÍA

Naughton, Patrick. Manual de Java. Osborne/Mc Graw Hill. 1996

Pressman, Roger S. Ingeniería del software un enfoque practico. Mc Graw Hill. 1993

Stroustrup, Bjarne. What is Object Oriented Programming? IEEE Software. Mayo 1988.

Wiener, Richard; Pinson, Lewis. An introduction to object-oriented programming Addison-Wesley Publishing Company. 1990.