

El lenguaje de Programación Java™

Java y todas las marcas basadas en Java son marcas registradas propiedad de Sun Microsystems, Inc. En los Estados Unidos y en otros países. El contenido de este libro: “El lenguaje de programación Java™” es independiente de Sun Microsystems, Inc.

MARCAS REGISTRADAS

Sun, el logotipo Sun, Sun Microsystems, JavaSoft, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, el logo Java Coffee Cup, y Visual Java son marcas registradas de Sun Microsystems, Inc.

UNIX es una marca registrada en los Estados Unidos y otros países, empleado con licencia de X/Open Company, Ltd. OPEN LOOK® es una marca registrada de Novell, Inc.

Los interfaces gráficos OPEN LOOK® y Sun(TM) GUI fueron desarrollados por Sun Microsystems, Inc. Para sus usuarios y licenciarios. Sun reconoce el esfuerzo pionero de Xerox en la investigación y el desarrollo del concepto de interface de usuario gráfica. Sun mantiene una licencia no exclusiva de Xerox sobre el interface gráfico de usuario de Xerox, que permite a Sun implementar GUIs OPEN LOOK.

X Window System es una marca registrada del X Consortium, Inc.

OpenStep es una marca propiedad de NeXT.

PREÁMBULO.....	9
1. INTRODUCCIÓN.....	11
1.1 HISTORIA DEL LENGUAJE.....	11
1.2 ¿QUÉ ES JAVA™?.....	12
1.3 ¿QUÉ LO HACE DISTINTO DE LOS DEMÁS LENGUAJES?.....	12
1.3.1 ¿Qué se puede programar con Java?.....	13
1.3.2 ¿Es fácil de aprender?.....	13
1.3.3 Características del lenguaje.....	14
1.4 LA MÁQUINA VIRTUAL JAVA (JVM).....	16
1.5 JAVASCRIPT.....	17
1.6 EL ENTORNO DE DESARROLLO JDK.....	18
1.6.1 ¿Dónde conseguirlo?.....	18
1.6.2 Estructura de directorios.....	19
1.6.3 Configuración.....	19
1.7 PROGRAMACIÓN ORIENTADA A OBJETOS.....	20
1.7.1 ¿Qué es un objeto?.....	20
1.7.2 Clases y objetos.....	21
1.7.3 Propiedades que debe cumplir un lenguaje para ser considerado Orientado a Objetos.....	23
1.8 CUESTIONES.....	27
2. VISIÓN GENERAL Y ELEMENTOS BÁSICOS DEL LENGUAJE.....	29
2.1 HOLA, MUNDO.....	30
2.2 COMENTARIOS.....	33
2.3 TIPOS DE DATOS.....	34
2.4 TIPOS DE DATOS SIMPLES.....	35
2.5 TIPOS DE DATOS REFERENCIALES.....	36
2.6 IDENTIFICADORES.....	37
2.7 DECLARACIÓN DE VARIABLES.....	39
2.8 ÁMBITO DE UNA VARIABLE.....	40
2.8.1 Variables locales.....	40
2.8.2 Atributos.....	43
2.8.3 Parámetros de un método.....	44
2.9 OPERADORES.....	46
2.9.1 Operadores aritméticos.....	47
2.9.2 Operadores relacionales.....	49
2.9.3 Operadores lógicos.....	50
2.9.4 Operadores de bits.....	51
2.9.5 Operadores de asignación.....	53
2.9.6 Precedencia de operadores en Java:.....	54
2.10 CONSTANTES.....	55
2.11 VALORES LITERALES.....	55
2.12 ESTRUCTURAS DE CONTROL.....	60
2.12.1 Estructuras alternativas.....	60

2.12.2	Bucles.....	66
2.12.3	Salto.....	70
2.13	VECTORES.....	72
2.14	CUESTIONES.....	76
3.	LAS CLASES EN JAVA.....	77
3.1	DECLARACIÓN DE CLASE.....	78
3.1.1	<i>Declaración de la superclase (herencia)</i>	79
3.1.2	<i>Lista de interfaces</i>	79
3.1.3	<i>Modificadores de clase</i>	80
3.2	EL CUERPO DE LA CLASE.....	83
3.3	DECLARACIÓN DE ATRIBUTOS.....	84
3.3.1	<i>Atributos static</i>	85
3.3.2	<i>Atributos final</i>	86
3.3.3	<i>Atributos transient</i>	87
3.3.4	<i>Atributos volatile</i>	88
3.3.5	<i>Modificadores de ámbito de atributos</i>	89
3.4	MÉTODOS.....	94
3.4.1	<i>Declaración de método</i>	94
3.4.2	<i>Cuerpo del método</i>	102
3.4.3	<i>Constructores</i>	106
3.4.4	<i>Destruyores</i>	110
3.5	STRINGS.....	110
3.5.1	<i>La clase String</i>	111
3.5.2	<i>La clase StringBuffer</i>	114
3.6	LA LÍNEA DE COMANDOS.....	115
3.7	INTERFACES.....	116
3.7.1	<i>Declaración de interface</i>	117
3.7.2	<i>Cuerpo del interface</i>	118
3.8	LOS PAQUETES.....	120
3.8.1	<i>Declaración de paquetes</i>	120
3.8.2	<i>Cómo hacer uso de los paquetes existentes</i>	122
3.8.3	<i>Paquetes pertenecientes al Java</i>	123
3.9	EJEMPLO COMPLETO.....	125
3.9.1	<i>Generar números aleatorios</i>	127
3.9.2	<i>Creación de un bombo</i>	129
3.9.3	<i>La clase Resultado</i>	131
3.9.4	<i>El programa principal</i>	132
3.9.5	<i>Modificando la clase Resultado</i>	133
3.10	CUESTIONES.....	136
4.	TRATAMIENTO DE EXCEPCIONES.....	137
4.1	LANZAMIENTO DE EXCEPCIONES (THROW).....	138
4.2	TRATAMIENTO DE EXCEPCIONES.....	139
4.2.1	<i>Manejador de excepciones</i>	139
4.3	JERARQUÍA DE EXCEPCIONES.....	142
4.4	VENTAJAS DEL TRATAMIENTO DE EXCEPCIONES.....	151

4.4.1	<i>Separación del código útil del tratamiento de errores</i>	151
4.4.2	<i>Propagación de errores a través de la pila de métodos</i>	153
4.4.3	<i>Agrupación y diferenciación de errores</i>	155
4.4.4	<i>Claridad del código y obligación del tratamiento de errores</i>	156
4.5	CUESTIONES	157
5.	LA ENTRADA/SALIDA EN JAVA	159
5.1	STREAMS	159
5.2	JERARQUÍA DE CLASES EN EL PAQUETE JAVA.IO	159
5.2.1	<i>El paquete java.io en el JDK 1.02</i>	160
5.2.2	<i>El paquete java.io en el JDK 1.1</i>	161
5.3	CLASES IMPORTANTES PARA LA ENTRADA/SALIDA ESTÁNDAR	164
5.3.1	<i>La clase InputStream</i>	164
5.3.2	<i>La clase OutputStream</i>	165
5.3.3	<i>La clase PrintStream</i>	166
5.4	ENTRADA/SALIDA ESTÁNDAR	168
5.5	STREAMS DE CARACTERES	169
5.5.1	<i>La clase Reader</i>	170
5.5.2	<i>La clase Writer</i>	171
5.5.3	<i>La clase BufferedReader</i>	171
5.5.4	<i>La clase PrintWriter</i>	172
5.6	ENTRADA POR TECLADO Y SALIDA POR PANTALLA	173
5.7	FICHEROS	175
5.7.1	<i>La clase File</i>	175
5.7.2	<i>La clase FileDescriptor</i>	177
5.7.3	<i>Acceso a ficheros secuenciales</i>	178
5.7.4	<i>La clase RandomAccessFile</i>	183
5.7.5	<i>Acceso a ficheros aleatorios</i>	185
5.8	SERIALIZACIÓN DE OBJETOS	187
5.8.1	<i>Esquema de funcionamiento de la serialización</i>	188
5.8.2	<i>Personalización en la serialización</i>	193
5.9	CUESTIONES	197
6.	THREADS	199
6.1	¿QUÉ SON LOS THREADS?	199
6.1.1	<i>Un ejemplo, mejor que mil palabras</i>	199
6.1.2	<i>Otro ejemplo</i>	201
6.1.3	<i>Y otro ejemplo más</i>	202
6.2	ESTADO DE UN THREAD	203
6.3	CREACIÓN DE THREADS	204
6.4	OPERACIONES SOBRE THREADS	207
6.4.1	<i>currentThread()</i>	207
6.4.2	<i>isAlive()</i>	207
6.4.3	<i>sleep()</i>	208
6.4.4	<i>suspend()</i>	209
6.4.5	<i>wait()</i>	209
6.4.6	<i>yield()</i>	209

6.4.7	<i>join()</i>	211
6.5	THREADS Y SINCRONISMO.....	212
6.5.1	<i>El paradigma productor/consumidor</i>	213
6.5.2	<i>Sección crítica</i>	214
6.5.3	<i>Monitores</i>	217
6.6	PRIORIDAD DE UN THREAD.....	224
6.7	THREADS DAEMON.....	226
6.8	GRUPOS DE THREADS.....	227
6.8.1	<i>Creación de grupos de threads</i>	227
6.8.2	<i>Operaciones sobre threads agrupados</i>	229
6.9	CUESTIONES.....	233
7.	COMUNICACIONES TCP/IP.....	235
7.1	INTRODUCCIÓN.....	235
7.2	ARQUITECTURA CLIENTE / SERVIDOR.....	236
7.3	LA ABSTRACCIÓN SOCKET.....	237
7.4	SERVICIO SIN CONEXIÓN (DATAGRAM SOCKET).....	239
7.4.1	<i>Creación de un DatagramSocket</i>	239
7.5	SERVICIO ORIENTADO A CONEXIÓN (STREAM SOCKET).....	242
7.5.1	<i>Operaciones en el servidor</i>	243
7.5.2	<i>Espera de conexiones de clientes</i>	243
7.5.3	<i>Operaciones en el cliente</i>	244
7.6	CUESTIONES.....	251
A.	LAS HERRAMIENTAS JAVA.....	253
A.1	JAVAC (COMPILADOR).....	253
A.1.1	<i>Opciones</i>	253
A.2	JAVA (INTÉRPRETE).....	255
A.2.1	<i>Opciones</i>	255
A.3	JAVAW (INTÉRPRETE).....	258
A.4	JDB (DEPURADOR).....	259
A.4.1	<i>Comandos básicos</i>	260
A.5	JAVAH (GENERADOR DE CABECERAS C).....	261
A.6	JAVAP (DESENSAMBLADOR).....	262
A.6.1	<i>Opciones</i>	263
A.7	JAVADOC (GENERADOR DE DOCUMENTACIÓN).....	265
A.7.1	<i>Etiquetas de clase o interface</i>	266
A.7.2	<i>Etiquetas de atributo</i>	269
A.7.3	<i>Etiquetas de constructor o método</i>	269
A.7.4	<i>Opciones</i>	269
B.	CONVERSIÓN DE TIPOS.....	273
B.1	CONVERSIÓN POR AMPLIACIÓN DE TIPOS DE DATOS SIMPLES.....	273
B.2	CONVERSIÓN POR REDUCCIÓN DE TIPOS DE DATOS SIMPLES.....	275
B.3	CONVERSIÓN POR AMPLIACIÓN DE TIPOS DE DATOS REFERENCIALES.....	277
B.4	CONVERSIÓN POR REDUCCIÓN DE TIPOS DE DATOS REFERENCIALES.....	278
B.5	CONVERSIONES DE TIPOS NO PERMITIDAS.....	281

C. LAS CLASES STRING Y STRINGBUFFER.....	283
C.1 LA CLASE STRING.....	283
C.1.1 Constructores de la clase <i>String</i>	283
C.1.2 Métodos de la clase <i>String</i>	284
C.2 LA CLASE STRINGBUFFER.....	289
C.2.1 Constructores de la clase <i>StringBuffer</i>	289
C.2.2 Métodos de la clase <i>StringBuffer</i>	289
D. CLASES ANIDADAS.....	293
D.1 UTILIDAD DE LAS CLASES ANIDADAS.....	294
D.2 CLASES ANIDADAS COMO MIEMBROS DE OTRAS CLASES.....	295
D.3 CLASES INTERIORES A UN BLOQUE O LOCALES A UN MÉTODO.....	299
D.4 CLASES ANÓNIMAS.....	302
E. CLASES SOBRE STREAMS Y FICHEROS.....	305
E.1 LA CLASE INPUTSTREAM.....	305
E.1.1 Constructor.....	305
E.1.2 Métodos.....	305
E.2 LA CLASE OUTPUTSTREAM.....	308
E.2.1 Constructor.....	308
E.2.2 Métodos.....	309
E.3 LA CLASE PRINTSTREAM.....	310
E.3.1 Constructores.....	310
E.3.2 Métodos.....	311
E.4 LA CLASE READER.....	312
E.4.1 Atributo.....	312
E.4.2 Constructores.....	313
E.4.3 Métodos.....	313
E.5 LA CLASE WRITER.....	316
E.5.1 Atributo.....	316
E.5.2 Constructores.....	316
E.5.3 Métodos.....	316
E.6 LA CLASE BUFFERDREADER.....	318
E.6.1 Constructores.....	318
E.6.2 Métodos.....	318
E.6.3 La clase <i>PrintWriter</i>	320
E.7 LA CLASE FILE.....	323
E.8 LA CLASE RANDOMACCESSFILE.....	328
E.8.1 Constructores.....	328
E.8.2 Métodos.....	328
F. EL FORMATO DE STRINGS UTF-8.....	335
F.1 CARACTERES DE UN SOLO BYTE.....	335
F.2 CARACTERES DE DOS BYTES.....	335
F.3 CARACTERES DE TRES BYTES.....	336
G. CLASES EN JAVA.NET.....	337
G.1 CLASS SOCKET.....	337

	<i>G.1.1 CONSTRUCTORES:</i>	338
	<i>G.1.2 MÉTODOS:</i>	340
G.2	CLASS SERVERSOCKET	342
	<i>G.2.1 CONSTRUCTORES:</i>	343
	<i>G.2.2 MÉTODOS:</i>	344
G.3	CLASS DATAGRAMSOCKET	346
	<i>G.3.1 CONSTRUCTORES:</i>	347
	<i>G.3.2 MÉTODOS:</i>	347
G.4	CLASS INETADDRESS	349
	<i>G.4.1 MÉTODOS:</i>	349
G.5	CLASS DATAGRAMPACKET	352
	<i>G.5.1 CONSTRUCTORES:</i>	352
	<i>G.5.2 MÉTODOS:</i>	353

A nuestras familias, por permitirnos robarles el tiempo que hemos dedicado a este libro. Muy especialmente a nuestras esposas M^a Carmen y Susana.

Sin su apoyo incondicional y su paciencia este libro no habría sido posible.

PREÁMBULO.

El presente texto está escrito de forma que pueda utilizarse tanto para aprender el lenguaje de programación Java, como para la consulta de atributos y métodos pertenecientes a clases del JDK¹ 1.1. Si bien se encontrará una descripción mucho más pormenorizada y actualizada en la documentación *on-line* que acompaña a este *software*.

Este libro no está orientado para iniciarse en la programación de *applets* con los que dotar de animación o interacción a páginas web. Por el contrario, el principal objetivo de este texto es presentar los fundamentos de la programación en el lenguaje Java, con el fin de construir cualquier tipo de aplicación (la construcción de *applets* y los aspectos gráficos se abordarán en un texto posterior “El lenguaje de programación Java II”).

En la actualidad existen diferentes paquetes para el desarrollo de aplicaciones en Java con herramientas más visuales que las que proporciona el JDK (que no son visuales en absoluto), sin embargo sigue siendo Sun Microsystems la empresa que controla la especificación de este lenguaje. La misma que desarrolla y distribuye (de momento) gratuitamente este paquete. Por este motivo, en este volumen se utilizarán estas herramientas algo “espartanas” pero indudablemente más avanzadas que otros productos con un *look* más moderno pero con un inevitable atraso en cuanto a las últimas modificaciones en el lenguaje y sus librerías.

Queremos insistir en que Java no es un lenguaje necesariamente ligado a Internet. Las aplicaciones en Java, al igual que las de otros lenguajes, pueden ser de cualquier naturaleza, ejecutarse sin problemas en ordenadores sin conexión a Internet y realizar todo tipo de tareas, desde cálculo científico hasta juegos, pasando por aplicaciones de negocios y oficina.

Lo que diferencia a Java de otros lenguajes de programación es su concepción de partida, en la que se pretende crear un lenguaje “todo-terreno” que se pueda utilizar para programar en todo tipo de sistemas operativos y procesadores.

Los autores deseamos que el lector disfrute con la lectura de este libro, y realizando sus ejemplos, tanto como nosotros lo hemos hecho con su redacción,

¹ JDK son las siglas de *Java Developers Kit*. Es decir, conjunto de herramientas para desarrollar (aplicaciones) en Java.

en la que inevitablemente hemos omitido algunos aspectos que nos hubiera gustado contar pero que han quedado en el tintero por motivos de tiempo y espacio.

Java, como todo nuevo lenguaje, acabará siendo lo que la comunidad de programadores desee que sea. Si su uso se extiende debido a que se aprecian las ventajas que aporta, las limitaciones que presenta en estos momentos se irán reduciendo paulatinamente.

Muchos fabricantes de *software* están migrando sus aplicaciones a este nuevo lenguaje. Posiblemente debido a que valoran más las ventajas y, fundamentalmente la posibilidad de desarrollar una sola aplicación para múltiples plataformas así como la mayor facilidad que ofrece Java para el mantenimiento posterior de las aplicaciones

Donde encuentre el símbolo  significa que hay un ejemplo que puede probarse. Es la forma más fácil de aprender. También es interesante tener iniciada una sesión con algún navegador y acceder a las API's de la documentación del JDK para consulta.

1. INTRODUCCIÓN.

1.1 *Historia del lenguaje.*

El lenguaje Java™ fue creado por Sun Microsystems Inc. en un proceso por etapas que arranca en 1990, año en el que Sun creó un grupo de trabajo, liderado por James Gosling, para desarrollar un sistema para controlar electrodomésticos e incluso PDAs o Asistentes Personales (pequeños ordenadores) que, además, permitiera la conexión a redes de ordenadores. Se pretendía crear un hardware polivalente, con un Sistema Operativo eficiente (SunOS) y un lenguaje de desarrollo denominado **Oak** (roble), el precursor de Java. El proyecto finalizó en 1992 y resultó un completo fracaso debido al excesivo coste del producto, con relación a alternativas similares, tras lo cual el grupo se disolvió.

Por entonces aparece Mosaic y la **World Wide Web**. Después de la disolución del grupo de trabajo, únicamente quedaba del proyecto el lenguaje Oak. Gracias a una acertada decisión de distribuir libremente el lenguaje por la Red de Redes y al auge y la facilidad de acceso a Internet, propiciado por la WWW, el lenguaje se popularizó y se consiguió que una gran cantidad de programadores lo depurasen y terminasen de perfilar la forma y usos del mismo. A partir de este momento, el lenguaje se difunde a una velocidad vertiginosa, añadiéndosele numerosas clases y funcionalidad para TCP/IP. El nombre del lenguaje tuvo que ser cambiado ya que existía otro llamado Oak. El nombre “Java” surgió en una de las sesiones de “brainstorming” celebradas por el equipo de desarrollo del lenguaje. Buscaban un nombre que evocara la esencia de la tecnología (viveza, animación, rapidez, interactividad ...). Java fue elegido de entre muchísimas propuestas. No es un acrónimo, sino únicamente algo humeante, caliente y que a muchos programadores les gusta beber en grandes cantidades: una taza de café (Java en argot Inglés americano²). De esta forma, Sun lanzó las primeras versiones de **Java** a principios de 1995.

Desde entonces, Sun ha sabido manejar inteligentemente el éxito obtenido por su lenguaje, concediéndose licencias a cualquiera sin ningún problema, fomentando

² Casualmente, la pronunciación en inglés de este término es “java”, que puede entenderse fuera de contexto como “¡ya va!”.

Introducción.

su uso entre la comunidad informática y extendiendo las especificaciones y funcionalidad del lenguaje.

1.2 ¿Qué es Java ?

Java es un lenguaje de desarrollo de propósito general, y como tal es válido para realizar todo tipo de aplicaciones profesionales.

Entonces, ¿es simplemente otro lenguaje más? Definitivamente no. Incluye una combinación de características que lo hacen único y está siendo adoptado por multitud de fabricantes como herramienta básica para el desarrollo de aplicaciones comerciales de gran repercusión.

1.3 ¿Qué lo hace distinto de los demás lenguajes?

Una de las características más importantes es que los programas ejecutables, creados por el compilador de Java, son **independientes de la arquitectura**. Se ejecutan indistintamente en una gran variedad de equipos con diferentes microprocesadores y sistemas operativos.

- De momento, es público. Puede conseguirse un JDK (Java Developer's Kit) o Kit de desarrollo de aplicaciones Java gratis. No se sabe si en un futuro seguirá siéndolo.
- Permite escribir *Applets* (pequeños programas que se insertan en una página HTML) y se ejecutan en el ordenador local.
- Se pueden escribir aplicaciones para intrarredes, aplicaciones cliente/servidor, aplicaciones distribuidas en redes locales y en Internet.
- Es fácil de aprender y está bien estructurado.
- Las aplicaciones son fiables. Puede controlarse su seguridad frente al acceso a recursos del sistema y es capaz de gestionar permisos y criptografía. También, según Sun, la seguridad frente a virus a través de redes locales e Internet está garantizada. Aunque al igual que ha

Introducción.

ocurrido con otras tecnologías y aplicaciones, se han descubierto, y posteriormente subsanado, “agujeros” en la seguridad³ de Java.

1.3.1 ¿Qué se puede programar con Java?

Si tenía preconcebida la idea de que con Java sólo se programan applets para páginas web, está completamente equivocado. Ya que Java es un lenguaje de propósito general, puede programarse en él cualquier cosa:

- **Aplicaciones independientes.** Como con cualquier otro lenguaje de propósito general.
- **Applets.** Pequeñas aplicaciones que se ejecutan en un documento HTML, siempre y cuando el navegador soporte Java, como ocurre con los navegadores HotJava y las últimas versiones de Netscape y el explorador de Internet de Microsoft.

1.3.2 ¿Es fácil de aprender?

Sí.

Para el colectivo de programadores que conocen la programación orientada a objetos, el cambio a Java puede ser realmente sencillo. Es un lenguaje bien estructurado, sin punteros y sin necesidad de tener que controlar la asignación de memoria a estructuras de datos u objetos.

Para los programadores en C++ también es sencillo el cambio, ya que la sintaxis es prácticamente la misma que en este lenguaje.

Para todo aquel que no conozca la programación orientada a objetos, este lenguaje es ideal para aprender todos sus conceptos, ya que en cada paso de su aprendizaje se va comprobando que las cosas se hacen en la forma natural de hacerlas, sin sorpresas ni comportamientos extraños de los programas. A medida que se va aprendiendo, se va fomentando en el programador, y sin esfuerzo, un buen estilo de programación orientada a objetos. En realidad, no puede ser de otra forma, ya

³ El aspecto de seguridad tiene importancia fundamental en la creación de *applets* puesto que estos programas son ejecutados en el navegador de cada usuario es necesario poder garantizarle que su sistema y sus datos están seguros de posibles virus y/o fisgoneos de terceras personas que podrían utilizar la red como medio para fines más o menos oscuros.

Introducción.

que Java impide “hacer cosas extrañas” y, además, no permite “abandonar” la programación orientada a objetos, como ocurre con otros lenguajes de programación. Esto es bastante conveniente, de lo contrario, un programador que está aprendiendo puede sentir la tentación de “volver” a lo que conoce (la programación tradicional).

A medida que se van comprobando las ventajas de la programación orientada a objetos, para aquellos que las desconocen, y la facilidad y naturalidad del lenguaje Java, éste va atrapando a quien se acerca a él, y después de algún tiempo trabajando con Java, hay pocos programadores que no lo consideren como “su favorito”.

1.3.3 Características del lenguaje.

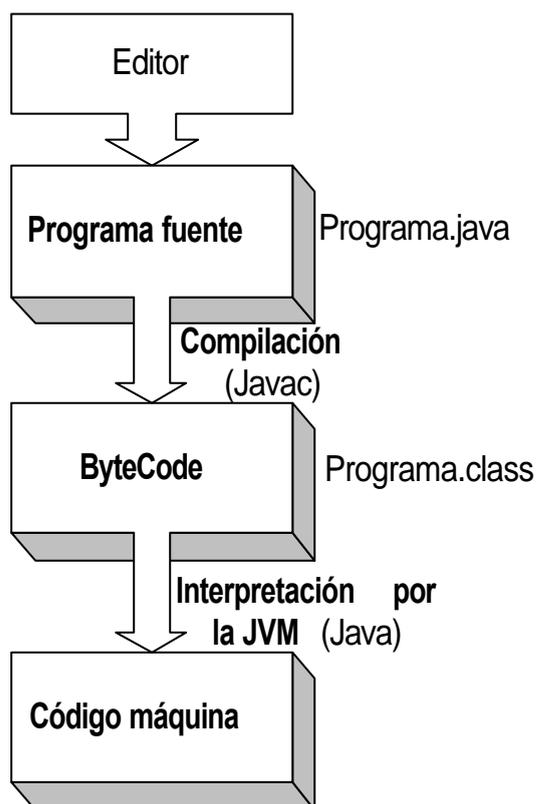
- Es intrínsecamente orientado a objetos.
- Funciona perfectamente en red.
- Aprovecha características de la mayoría de los lenguajes modernos evitando sus inconvenientes. En particular los del C++.
- Tiene una gran funcionalidad gracias a sus librerías (clases).
- NO tiene punteros manejables por el programador, aunque los maneja interna y transparentemente.
- El manejo de la memoria no es un problema, la gestiona el propio lenguaje y no el programador.
- Genera aplicaciones con pocos errores posibles.
- Incorpora *Multi-Threading* (para permitir la ejecución de tareas concurrentes dentro de un mismo programa). Esta característica será tratada con detenimiento en el punto  6 **Threads**. En la página 199.

El lenguaje Java puede considerarse como una evolución del C++. La sintaxis es parecida a la de este lenguaje, por lo que en este libro se hará referencia a dicho lenguaje frecuentemente. A pesar de que puede considerarse como una evolución del C++ no acarrea los inconvenientes del mismo, ya que Java fue diseñado “partiendo de cero”, es decir, no necesitaba ser compatible con versiones anteriores de ningún lenguaje como ocurre con C++ y C.

Introducción.

Gracias a que fue diseñado “partiendo de cero” ha conseguido convertirse en un lenguaje orientado a objetos puro, limpio y práctico. No permite programar mediante otra técnica que no sea la programación orientada a objetos (POO en adelante) y, una vez superado el aprendizaje de la programación orientada a objetos, es realmente sencillo aprender Java.

¿El lenguaje es Compilado o Interpretado? Ni una cosa ni la otra. Aunque estrictamente hablando es interpretado, necesita de un proceso previo de compilación. Una vez “compilado” el programa, se crea un fichero que almacena lo que se denomina *bytecodes* o *j_code* (pseudocódigo prácticamente al nivel de código máquina). Para ejecutarlo, es necesario un “intérprete”, la JVM (*Java Virtual Machine*) **máquina virtual Java**. De esta forma, es posible compilar el programa en una estación UNIX y ejecutarlo en otra con Windows95 utilizando la máquina virtual Java para Windows95. Esta JVM se encarga de leer los *bytecodes* y traducirlos a instrucciones ejecutables directamente en un determinado microprocesador, de una forma bastante eficiente.



Que el programa deba ser “interpretado” no hace que sea poco eficiente en cuanto a velocidad, ya que la interpretación se hace prácticamente al nivel de código máquina. Por ejemplo, es mucho más rápido que cualquier otro programa interpretado como por ejemplo Visual Basic, aunque es más lento que el mismo programa escrito en C++.

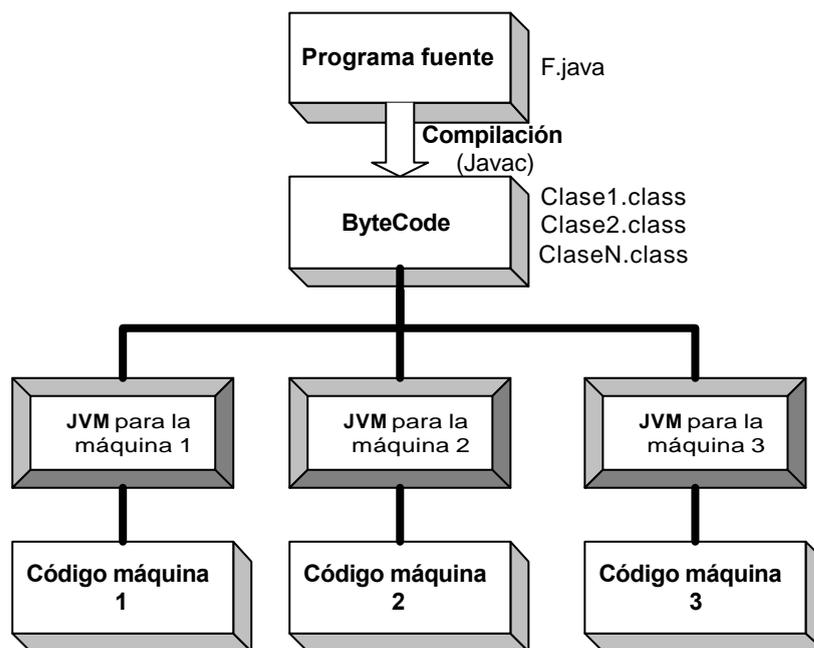
Esta deficiencia en cuanto a la velocidad, puede ser aminorada por los compiladores *Just-In-Time* (JIT). Un compilador JIT transforma los *bytecodes* de un programa o un *applet* en código nativo de la plataforma donde se ejecuta, por lo que es más rápido. Suelen ser incorporados por los navegadores, como Netscape o Internet Explorer.

Introducción.

El lenguaje Java es robusto. Las aplicaciones creadas en este lenguaje son susceptibles de contener pocos errores, principalmente porque la gestión de memoria y punteros es realizada por el propio lenguaje y no por el programador. Bien es sabido que la mayoría de los errores en las aplicaciones vienen producidos por fallos en la gestión de punteros o la asignación y liberación de memoria. Además, el lenguaje contiene estructuras para la detección de excepciones (errores de ejecución previstos) y permite obligar al programador a escribir código fiable mediante la declaración de excepciones posibles para una determinada clase reutilizable.

1.4 La Máquina Virtual Java (JVM).

La máquina virtual Java es la idea revolucionaria⁴ del lenguaje. Es la entidad que proporciona la independencia de plataforma para los programas Java “compilados” en *byte-code*.



⁴ Bueno, no completamente revolucionario, otros sistemas en el pasado, como por ejemplo el Pascal UCSD compilaban a un código intermedio (p-code) que luego era interpretado al ejecutar el programa. No obstante, esta estrategia no es la habitual en la mayoría de lenguajes, al menos no lo ha sido hasta ahora.

Introducción.

Un mismo programa fuente compilado en distintas plataformas o sistemas operativos, genera el mismo fichero en byte-code. Esto es lógico, ya que se supone que el compilador de Java traduce el fichero fuente a código ejecutable por una máquina que únicamente existe en forma virtual (aunque se trabaja en la construcción de microprocesadores que ejecuten directamente el byte-code).

Evidentemente, si un mismo programa en byte-code puede ser ejecutado en distintas plataformas es porque existe un traductor de ese byte-code a código nativo de la máquina sobre la que se ejecuta. Esta tarea es realizada por la JVM. Existe una versión distinta de esta JVM para cada plataforma. Esta JVM se carga en memoria y va traduciendo “al vuelo”, los byte-codes a código máquina. La JVM no ocupa mucho espacio en memoria, piénsese que fue diseñada para poder ejecutarse sobre pequeños electrodomésticos como teléfonos, televisores, etc.

1.5 JavaScript.

Atención: No hay que confundir Java con JavaScript.

JavaScript es una variación del lenguaje Java. Desarrollado por Netscape y Sun, fue diseñado para simplificar la creación de contenidos interactivos en páginas web sin necesidad de tener que programar applets en Java. En lugar de ello se utilizan técnicas más sencillas mediante el lenguaje JavaScript que es más flexible, aunque mucho menos potente.

El lenguaje JavaScript es totalmente interpretado por el navegador. El código fuente se incluye como parte de la página web en HTML y es el navegador el encargado de “interpretar” el código fuente.

¿No podría hacerse en Java? Sí, pero no de forma tan sencilla como con JavaScript. Este lenguaje fue pensado para acercar Java a programadores inexpertos y creadores de contenidos HTML sin conocimientos avanzados de programación.

Diferencias entre Java y JavaScript:

- Java es compilado, mientras que JavaScript es totalmente interpretado.
- Java es orientado a objetos. JavaScript utiliza objetos, pero no permite la programación orientada a objetos.

Introducción.

- En JavaScript no es necesario declarar las variables y en Java sí.
- En JavaScript, las comprobaciones de validez de referencias a objetos se realizan en tiempo de ejecución, mientras que en Java se realiza en tiempo de compilación.
- JavaScript tiene un número limitado de tipos de datos y clases.
- Los applets Java se transmiten como código aparte de la página Web. En JavaScript, los applets se transmiten conjuntamente con la página web (embebidos en ella).

1.6 El entorno de desarrollo JDK.

La herramienta básica para empezar a desarrollar aplicaciones o *applets* en Java es el JDK (*Java Developer s Kit*) o Kit de Desarrollo Java, que consiste, básicamente, en un compilador y un intérprete (JVM) para la línea de comandos. No dispone de un entorno de desarrollo integrado (IDE), pero es suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones.

1.6.1 ¿Dónde conseguirlo?

El Kit de desarrollo puede obtenerse en las direcciones siguientes:

- <http://www.sun.com>
- <http://www.javasoft.com>

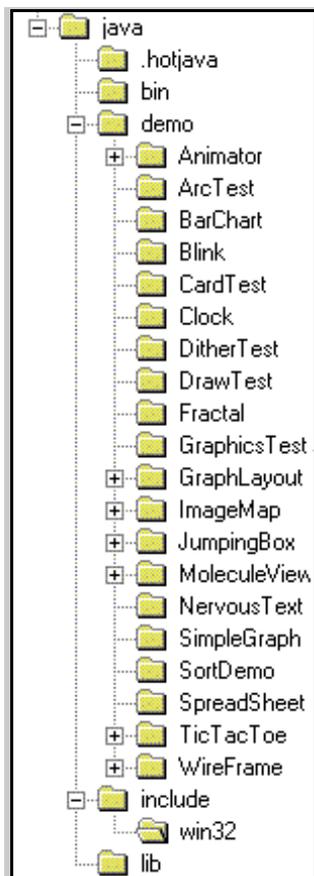
El entorno para Windows95/NT está formado por un fichero ejecutable que realiza la instalación, creando toda la estructura de directorios. El kit contiene básicamente:

- El compilador: **javac.exe**
- El depurador: **jdb.exe**
- El intérprete: **java.exe** y **javaw.exe**
- El visualizador de applets: **appletviewer.exe**
- El generador de documentación: **javadoc.exe**

Introducción.

- Un desensamblador de clases: **javap.exe**
- El generador de archivos fuentes y de cabecera (.c y .h) para clases nativas en C: **javah.exe**

1.6.2 Estructura de directorios.



.hotJava: Es donde se almacenan los ficheros de configuración del navegador de Sun HotJava.

bin: Es donde se almacenan los ficheros ejecutables del entorno de desarrollo: javac, jdb, java y appletviewer.

demo: Contiene varios directorios con ejemplos de programas y applets escritos en Java.

include: Contiene cabeceras para utilizar funciones escritas en C.

lib: Contiene las librerías de clases proporcionadas por el lenguaje. Estas librerías se encuentran comprimidas en el fichero **classes.zip**.



Atención, este último fichero no se debe descomprimir.

1.6.3 Configuración.

En este caso se va a suponer que el JDK ha sido instalado en el directorio `c:\java`. En otro caso, bastaría con sustituir éste por el directorio realmente utilizado.

Introducción.

En primer lugar, para poder ejecutar el compilador `javac` y poder interpretar los programas “ejecutables” resultantes, hay que añadir en el **path** la localización de los ejecutables del entorno de desarrollo, modificando el fichero `autoexec.bat` de la siguiente forma:

```
path = c:\java\bin; ....
```

También hay que añadir en él una variable **CLASSPATH** que contendrá el/los directorios donde se almacenan las clases.

```
set CLASSPATH = c:\java\lib; c:\Java\MisClases; . ; ....
```

Es conveniente incluir el directorio actual (`.`) para poder acceder a las clases que se están creando en un determinado directorio en un momento dado.

1.7 Programación orientada a objetos.

Antes de empezar a estudiar en detalle el lenguaje Java, es imprescindible conocer los conceptos y características particulares de la programación orientada a objetos (POO). No sólo es importante que este punto sea leído por quien desconoce la POO sino que también es importante que lo lea quien la conoce, ya que existen distintas terminologías para nombrar los mismos conceptos.

La programación orientada a objetos es una evolución lógica de la programación estructurada, en la que el concepto de variables locales a un procedimiento o función, que no son accesibles a otros procedimientos y funciones, se hace extensible a los propios subprogramas que acceden a estas variables. Pero la programación orientada a objetos va mucho más allá. En realidad, cambia la concepción de la metodología de diseño de los programas.

En la programación orientada a objetos, se definen objetos que conforman una aplicación. Estos objetos están formados por una serie de características y operaciones que se pueden realizar sobre los mismos. Estos objetos no están aislados en la aplicación, sino que se comunican entre ellos.

1.7.1 ¿Qué es un objeto?

La respuesta a esta pregunta en términos ajenos a la programación parece simple. Un objeto es una persona, animal o cosa. Se distingue de otros objetos por tener unas determinadas características y “sirve” para algo, o dicho de otra forma, se pueden realizar distintas operaciones con/sobre ese objeto.

Introducción.

Por ejemplo: Una casa es un objeto.

CARACTERÍSTICAS: Número de pisos, altura total en metros, color de la fachada, número de ventanas, número de puertas, ciudad, calle y número donde está ubicada, etc.

OPERACIONES: Construir, destruir, pintar fachada, modificar alguna de las características, como por ejemplo, abrir una nueva ventana, etc.

Evidentemente, cada objeto puede definirse en función de multitud de características y se pueden realizar innumerables operaciones sobre él. Ya en términos de programación, será misión del programador determinar qué características y que operaciones interesa mantener sobre un objeto. Por ejemplo, sobre el objeto casa puede no ser necesario conocer su ubicación y por lo tanto, dichas características no formarán parte del objeto definido por el programador. Lo mismo podría decirse sobre las operaciones.

En terminología de programación orientada a objetos, a las características del objeto se les denomina **ATRIBUTOS** y a las operaciones **MÉTODOS**. Cada uno de estos métodos es un procedimiento o una función perteneciente a un objeto.

TERMINOLOGÍA INFORMAL	TERMINOLOGÍA FORMAL
CARACTERÍSTICAS	ATRIBUTOS
OPERACIONES (PROCEDIMIENTOS Y FUNCIONES)	MÉTODOS

Un objeto está formado por una serie de características o datos (atributos) y una serie de operaciones (métodos). No puede concebirse únicamente en función de los datos o de las operaciones sino en su conjunto.

1.7.2 Clases y objetos.

En la POO hay que distinguir entre dos conceptos íntimamente ligados, la **CLASE** y el **OBJETO**.

Introducción.

De forma análoga a cómo se definen las variables en un lenguaje de programación, cuando se declara un objeto hay que definir el tipo de objeto al que pertenece. Este tipo es la CLASE.

En C, se definen dos variables X e Y de tipo entero de la forma siguiente:

```
int X,Y;
```

En este caso, X e Y son variables, y el tipo de dichas variables es ENTERO.

La forma de declarar objetos en Java es la misma:

```
Ccasa casa1,casa2;
```

En este caso, `casa1` y `casa2` son efectivamente variables, pero un tanto especiales, son OBJETOS.

Además, el tipo de objetos es `Ccasa`. Este tipo es la CLASE del objeto⁵.

ANALOGÍA

VARIABLE OBJETO

TIPO CLASE

Al declarar `casa1` y `casa2` como objetos pertenecientes a la clase `Ccasa`, se está indicando que `casa1` y `casa2` tendrán una serie de atributos (datos) como son `nPuertas`, `nVentanas` y `color`; y, además, una serie de métodos (operaciones que se pueden realizar sobre ellos) como son: `abrirVentanas()`, `cerrarVentanas()`, etc.

⁵ Aunque no todos los textos emplean la misma notación, nosotros denominaremos **objeto** a la instancia de una **clase**. De este modo clase y tipo son los conceptos similares en programación orientada a objetos y programación estructurada al igual que hay similitud entre los conceptos de objeto y variable.

1.7.3 Propiedades que debe cumplir un lenguaje para ser considerado Orientado a Objetos.

- ENCAPSULAMIENTO.
- HERENCIA.
- POLIMORFISMO.

1.7.3.1 Encapsulamiento.

El encapsulamiento consiste en la propiedad que tienen los objetos de ocultar sus atributos, e incluso los métodos, a otras partes del programa u otros objetos.

La forma natural de construir una clase es la de definir una serie de atributos que, en general, no son accesibles fuera del mismo objeto, sino que únicamente pueden modificarse a través de los métodos que son definidos como accesibles desde el exterior de esa clase.



```
class Ccasa {
    int nPuertas, nVentanas;
    String color;
    public Ccasa(int np, int nv, String co) {
        nPuertas=np;
        nVentanas=nv;
        color=co;
    }
    public void pintar(String co) {
        color=co;
    }
    public void abrirVentanas(int n) {
        nVentanas=nVentanas+n;
    }
    public void cerrarVentanas(int n) {
        nVentanas=nVentanas-n;
        if (nVentanas<0)
            nVentanas=0;
    }
    public void abrirPuertas(int n) {
        nPuertas=nPuertas+n;
    }
    public void cerrarPuertas(int n) {
```

Introducción.

```
nPuertas=nPuertas-n;
if (nPuertas<0)
    nPuertas=0;
}
...
Ccasa casa1,casa2;
...
```

La forma normal de declarar la CLASE `Ccasa` consiste en definir una serie de atributos no accesibles desde cualquier parte del programa, sino únicamente a través de determinados métodos. Así, si se quiere abrir una nueva ventana en la casa `casa1`, la filosofía tradicional de un programador consistiría en realizar lo siguiente:

```
casal.N_VENTANAS := casal.N_VENTANAS + 1;
```

Sin embargo, la forma natural de hacerlo en POO es llamando al método `casal.abrirVentanas(1)`. Ese método (procedimiento) se encargará de incrementar en 1 el atributo `nVentanas`.

Esto no quiere decir que el atributo `nVentanas` no pueda ser accedido de la forma tradicional (si se hubiera definido como “`public`”) pero, para que el lenguaje pueda ser considerado como OO, debe permitir la posibilidad de prohibir el acceso a los atributos directamente.

1.7.3.2 Herencia.

Es una de las principales ventajas de la POO. Esta propiedad permite definir clases descendientes de otras, de forma que la nueva clase (la clase descendiente) hereda de la clase antecesora todos sus ATRIBUTOS y MÉTODOS. La nueva clase puede definir nuevos atributos y métodos o incluso puede redefinir atributos y métodos ya existentes (por ejemplo: cambiar el tipo de un atributo o las operaciones que realiza un determinado método).

Es la forma natural de definir objetos en la vida real. La mayoría de la gente diría, por ejemplo, que un chalet es una casa con jardín. Tiene las mismas características y propiedades u operaciones que pueden realizarse sobre una casa y, además, incorpora una nueva característica, el **jardín**. En otras ocasiones, se añadirá funcionalidad (métodos) y no atributos. Por ejemplo: un pato es un ave que nada. Mantiene las mismas características que las aves y únicamente habría que declarar un método sobre la nueva clase (el método **nadar**).

Introducción.

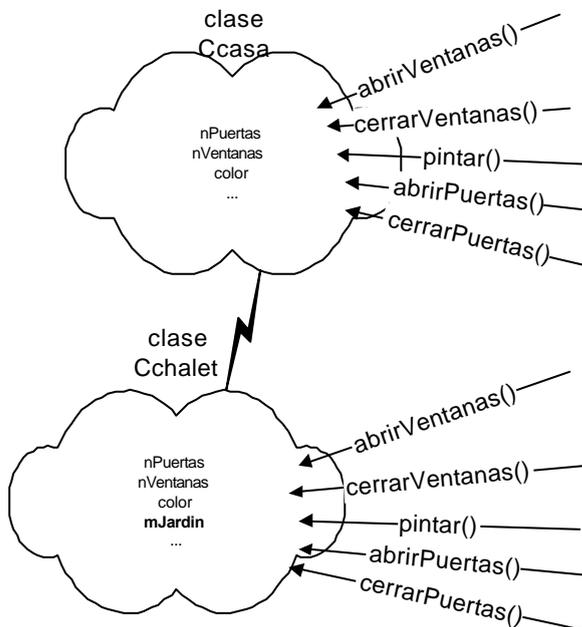
Esta propiedad permite la reutilización del código, siendo muy fácil aprovechar el código de clases ya existentes, modificándolas mínimamente para adaptarlas a las nuevas especificaciones.

Ejemplo: Supongamos que tenemos construida la clase `Ccasa` y queremos definir una nueva clase que represente a los chalets. En este caso puede ser conveniente definir un nuevo atributo que represente los metros cuadrados de jardín. En lugar de volver a definir una nueva clase “desde cero”, puede aprovecharse el código escrito para la clase `Ccasa` de la siguiente forma.



```
class Cchalet extends Ccasa {
    int mJardin;
    public Cchalet(int np, int nv, String co, int nj) {
        super(np,nv,co);
        mJardin=nj;
    }
}

...
Cchalet chalet1,chalet2;
...
```



Como puede verse, únicamente hay que declarar que la nueva clase `Cchalet` es descendiente de `Ccasa` (`extends Ccasa`) y declarar el nuevo atributo.

Evidentemente, el método constructor⁶ hay que redefinirlo para poder inicializar el nuevo atributo `mJardin`. Pero los métodos para Abrir/Cerrar puertas o ventanas no es necesario

⁶ El constructor es un método especial, con el mismo nombre que la clase y cuya función es inicializar los atributos de cada objeto (instancia).

Introducción.

definirlos, son heredados de la clase `Ccasa` y pueden utilizarse, por ejemplo de la siguiente forma:

```
chalet1.pintar("Blanco");
```

1.7.3.3 Polimorfismo.

El polimorfismo permite que un mismo mensaje enviado a objetos de clases distintas haga que estos se comporten también de forma distinta (objetos distintos pueden tener métodos con el mismo nombre o incluso un mismo objeto puede tener nombres de métodos idénticos pero con distintos parámetros)⁷.



```
class Ccasa {
...
public Ccasa(int np, int nv, String co) {
    nPuertas=np;
    nVentanas=nv;
    color=co;
}
public Ccasa() {
    nPuertas=0;
    nVentanas=0;
    color="";
}
...
}
```

Tiene dos métodos con el mismo nombre pero parámetros distintos. En el primer caso se inicializarán los atributos del objeto con los parámetros del método y en el segundo caso se inicializarán a cero, por ejemplo.

Además, si tenemos dos objetos `casal` y `chalet1` y se llama al método `chalet1.abrirVentanas(2)` se ejecutará el código del procedimiento `abrirVentanas` de la clase `Cchalet` y no de la clase `Ccasa`.

⁷ Académicamente esto no es rigurosamente cierto. En realidad lo descrito aquí tiene más que ver con la sobrecarga que con el polimorfismo, pero el concepto de polimorfismo es bastante más complicado y a veces es mejor una mentira fácil de entender que una verdad incomprensible.

1.8 CUESTIONES

1. El lenguaje Java fue desarrollado con una finalidad específica, muy distinta de su uso actual. ¿Realmente se trata de un lenguaje de programación de propósito general?
2. ¿Es Java un lenguaje compilado o interpretado? ¿Por qué?
3. ¿Qué es un *applet*? ¿Para qué sirve?
4. ¿Cuál es la finalidad de un compilador JIT (*Just in time*)?
5. ¿Qué es el *bytecode*?
6. ¿Es posible conseguir herramientas de desarrollo gratuitas para construir aplicaciones y *applets* en Java? Si es así, ¿dónde?
7. Se te encarga la tarea de realizar un programa que actúe como una calculadora simple, con las cuatro operaciones básicas. Piensa qué clases definirías y con qué atributos y métodos.
8. ¿Cuál es la función de la variable de entorno CLASSPATH en la instalación del JDK?
9. Una urbanización está constituida por un conjunto de cuatro casas, cada una con una orientación, posición y aspecto diferente. Implementa la clase `Urbanizacion`.

2. VISIÓN GENERAL Y ELEMENTOS BÁSICOS DEL LENGUAJE.

En Java, prácticamente todo son clases (objetos). El lenguaje obliga a la programación orientada a objetos y no permite la posibilidad de programar mediante ninguna otra técnica que no sea ésta. Por esta razón, un programa estará formado por uno o varios ficheros fuente y en cada uno de ellos habrá definida una o varias clases.

En un fichero fuente puede declararse una o más clases y tendrá un aspecto similar a este:

```
class Clase1 {
```

```
}
```

```
class Clase2 {
```

```
}
```

```
class ClaseN {
```

```
}
```

Una clase está formada por una parte correspondiente a la declaración, y otra correspondiente al cuerpo de la misma:

```
Declaración de clase {
```

```
    Cuerpo de clase
```

```
}
```

Visión General y elementos básicos del lenguaje.

En la plantilla anterior se ha simplificado el aspecto de la *Declaración de clase*, pero sí que puede asegurarse que la misma contendrá, como mínimo, la palabra reservada `class` y el nombre que recibe la clase.

El cuerpo de las clases comienza con una llave abierta({) y termina con una llave cerrada (}).

Dentro del cuerpo de la clase se declaran los **atributos** y los **métodos** de la clase.

Para que un programa se pueda ejecutar, debe contener una clase que tenga un método `main` con la siguiente declaración:

```
public static void main( String argumentos [] )
```

2.1 *Hola, mundo.*

En primer lugar, vamos a ver qué estructura debería tener el típico programa con el que se empieza en cualquier lenguaje de programación y que proporciona una primera visión de la sintaxis del lenguaje y que comienza a transmitir una serie de sentimientos de aceptación o de rechazo en el programador. El programa simplemente mostrará en pantalla el mensaje “hola, mundo” y terminará.

Este ejemplo habrá que crearlo mediante un editor cualquiera y guardar el fichero con el nombre **Hola.java** (ATENCIÓN: Es importante que tenga el mismo nombre que la clase “Hola”, distinguiendo entre mayúsculas y minúsculas).



```
class Hola {  
    public static void main(String argumentos[]) {  
        System.out.println("hola, mundo");  
    }  
}
```

El lenguaje Java, al igual que C, distingue entre mayúsculas y minúsculas, por lo que es importante transcribirlo literalmente.

También hay que comentar que en el nombre del fichero fuente también se hace distinción entre mayúsculas y minúsculas a la hora de

Visión General y elementos básicos del lenguaje.

compilarlo, aunque el sistema operativo empleado no haga esta distinción. La extensión del mismo debe ser **.java**

Para compilar el programa, hay que teclear:

```
javac Hola.java
```

El compilador creará un fichero que se llama `Hola.class` que contiene el código *bytecode* (“pseudoejecutable”)

Para ejecutarlo:

```
java Hola
```

Se escribe únicamente el nombre de la clase `Hola` y no `Hola.class` u `Hola.java`. El intérprete Java buscará el fichero `Hola.class` en los directorios especificados en la variable de entorno `CLASSPATH` (como se vio en el punto 1.6.3 **Configuración**. En la página 19) y, si lo encuentra y tiene método `main()`, lo ejecutará.

El resultado de la ejecución será la visualización en pantalla del mensaje:

```
hola, mundo
```

Explicación del programa `Hola.java` línea a línea:

Este simple programa, a pesar de no ser muy extenso, contiene muchos de los conceptos de la programación orientada a objetos en Java. No se pretende que, a partir del mismo, se aprendan y comprendan la totalidad de los aspectos de la POO y la sintaxis del Java, pero sí que puede afirmarse que una vez comprendida y asimilada su filosofía, se estará en un punto bastante cercano a los conocimientos básicos necesarios para entenderlos.

La primera línea del programa declara una clase llamada `Hola`, que es descendiente, al no indicar nosotros que herede de otra clase, de la clase `Object`. Entre las llaves de la clase `Hola`, se declaran los atributos y los métodos de la clase. Todo lo que se encuentre entre la llave abierta (`{`) y la llave cerrada (`}`) pertenece a la clase `Hola`. En nuestro ejemplo, únicamente tiene un método: `main`.

Las llaves no sirven únicamente para marcar el inicio y el fin de una clase. Se utilizan para marcar principio y final de bloques de código y se interpretan

Visión General y elementos básicos del lenguaje.

mediante el método LIFO (*Last In First Out*) donde el último en entrar es el primero en salir. En el ejemplo existen, además de las llaves abierta y cerrada de la clase, otras llaves abierta y cerrada dentro del método `main()`. Que se interpreten mediante el método LIFO significa que la llave cerrada (`}`) del método `main()` parentiza el bloque abierto por la última llave (`{`) antes de ésta.

Todo programa independiente escrito en Java empieza a ejecutarse (como en C) a partir del método `main()`. Se pueden compilar clases que no posean método `main()`, pero el intérprete Java no podrá ejecutarlas inicialmente, aunque sí pueden ejecutarse si son llamadas desde otro método en ejecución. Veremos en posteriores capítulos que la forma de crear los *applets* es distinta y no necesita declarar este método.

Declaración del método `main()`:

- `public`: indica que el método `main()` es público y, por tanto, puede ser llamado desde otras clases. Todo método `main()` debe ser público para poder ejecutarse desde el intérprete Java (JVM).
- `static`: indica que la clase no necesita ser instanciada para poder utilizar el método al que califica. (No se crea ninguna instancia u objeto de la clase `Hola`). También indica que el método es el mismo para todas las instancias que pudieran crearse.
- `void`: indica que la función `main` no devuelve ningún valor.
- El método `main` debe aceptar siempre, como parámetro, un vector de *strings*, que contendrá los posibles argumentos que se le pasen al programa en la línea de comandos, aunque como es nuestro caso, no se utilice.

Si no se han comprendido muy bien todos estos conceptos no importa, de momento. Hay una regla sencilla que nunca falla:

El método `main()` siempre se declara de la misma forma:

```
public static void main(String argumentos[])
```

Visión General y elementos básicos del lenguaje.

La instrucción que realmente realiza el trabajo efectivo de nuestro programa es la siguiente:

```
System.out.println("hola, mundo");
```

Esta línea indica que se va a ejecutar el método `println()`, encargado de mostrar un valor a través de la salida estándar (en nuestro caso, un `String`) y después realizar un retorno de carro y nueva línea. Este método pertenece al atributo `out`. Este atributo se encuentra incluido dentro de la clase `System`. Como curiosidad (de momento), cabe mencionar que esta clase es `static` (no hemos declarado ningún objeto de la clase `System`).

¿Un poco complicado para simplemente mostrar un mensaje en pantalla? Es posible que sí lo sea para aquellos que aún no conocen la programación orientada a objetos, pero una vez conocidos y asimilados los conceptos de la POO (Programación Orientada a Objetos) el programa es bastante sencillo y lógico. Para aquellos que conozcan C++ pensarán que es bastante parecido a lo que habrían escrito en su “lenguaje favorito”. Este colectivo comprobará que es mucho más fácil que C y tiene algunas ventajas.

Después de haber tomado contacto con Java, vamos a describir los elementos que componen el lenguaje.

2.2 Comentarios.

Los comentarios en los programas fuente son muy importantes en cualquier lenguaje. Sirven para aumentar la facilidad de comprensión del código y para recordar ciertas cosas sobre el mismo. Son porciones del programa fuente que no serán compiladas, y, por tanto, no ocuparán espacio en el fichero “ejecutable”. Únicamente sirven para documentar.

Visión General y elementos básicos del lenguaje.

- Si un comentario debe ocupar más de una línea, hay que anteponerle `/*` y al final `*/`.



Ejemplo:

```
/* Esto es un
comentario que
ocupa tres líneas */
```

- Si el comentario que se desea escribir es de una sola línea, basta con poner dos barras inclinadas: `//`



Ejemplo:

```
for (i=0; i<20;i++) // comentario de bucle
{ System.out.println("Adiós");
}
```

No puede ponerse código después de un comentario introducido por `//` en la misma línea, ya que desde la aparición de las dos barras inclinadas `//` hasta el final de la línea es considerado como comentario e ignorado por el compilador.

- Existe otro tipo de comentario que sirve para generar documentación automáticamente en formato HTML mediante la herramienta **javadoc** (☞ Véase A.7 javadoc (Generador de documentación). en la página 265). Puede ocupar varias líneas y se inicia con `/**` para terminar con `*/`.

2.3 Tipos de datos.

En Java existen dos tipos principales de datos:

- 1) Tipos de datos simples.
- 2) Referencias a objetos.

Los tipos de datos simples son aquellos que pueden utilizarse directamente en un programa, sin necesidad del uso de clases (POO). Estos tipos son:

Visión General y elementos básicos del lenguaje.

byte **short** **int** **long**
float **double** **char** **boolean**

El segundo tipo está formado por todos los demás. Se les llama **referencias** porque en realidad lo que se almacena en los mismos son punteros a zonas de memoria donde se encuentran almacenadas las estructuras de datos que los soportan. Dentro de este grupo se encuentran las clases (objetos) y también se incluyen las interfaces, los vectores y los *Strings*.

Pueden realizarse conversiones entre los distintos tipos de datos (incluso entre simples y referenciales), bien de forma implícita o de forma explícita. (↪ Véase: B Conversión de tipos. En la página 273).

2.4 Tipos de datos simples.

Los tipos de datos simples soportados por Java son los siguientes:

TIPO	Descripción	Formato	long.	Rango
byte	byte	C-2*	1 byte	- 128 ... 127
short	entero corto	C-2	2 bytes	- 32.768 ... 32.767
int	entero	C-2	4 bytes	- 2.147.483.648 ... 2.147.483.647
long	entero largo	C-2	8 bytes	- 9.223.372.036.854.775.808... 9.223.372.036.854.775.807
float	real en coma flotante de s.p. **	IEEE 754	32 bits	$\pm 3,4 \cdot 10^{-38} \dots \pm 3,4 \cdot 10^{38}$
double	real en coma flotante de d.p.	IEEE 754	64 bits	$\pm 1,7 \cdot 10^{-308} \dots \pm 1,7 \cdot 10^{308}$
char	carácter	Unicode	2 bytes	0 ... 65.535
boolean	lógico		1 bit	<i>true</i> / <i>false</i>

*C-2 = Complemento a dos. **s.p. = Simple Precisión / d.p. = Doble Precisión

No existen más datos simples en Java. Incluso éstos que se enumeran son envueltos por clases equivalentes (*java.lang.Integer*, *java.lang.Double*, *java.lang.Byte*, etc.), que pueden tratarlos como si fueran objetos en lugar de datos simples.

Visión General y elementos básicos del lenguaje.

A diferencia de otros lenguajes de programación como el C, en Java los tipos de datos simples no dependen de la plataforma ni del sistema operativo. Un entero de tipo `int` siempre tendrá 4 bytes, por lo que no tendremos sorpresas al migrar un programa de un sistema operativo a otro. Es más, ni siquiera hay que volverlo a compilar.

Eso sí, Java no realiza una comprobación de los rangos. Por ejemplo: si a una variable de tipo `short` con el valor 32.767 se le suma 1, el resultado será -32.768 y no se producirá ningún error de ejecución.

Nota: A diferencia de otros lenguajes de programación, los `Strings` en Java no son un tipo simple de datos sino un objeto que se estudiará en otro punto (↪ Ver punto 3.5. **STRINGS**. En página 110). Los valores de tipo `String` van entre comillas dobles ("Hola"), mientras que los de tipo `char` van entre comillas simples ('K').

Los valores que pueden asignarse a variables y que pueden ser utilizados en expresiones directamente reciben el nombre de literales. ↪ Véase el punto 2.11 Valores literales. En la página 55 para conocer cómo indicar el tipo de dato asociado a cada literal.

2.5 Tipos de datos referenciales.

El resto de tipos de datos que no son simples, son considerados referenciales. Estos tipos son básicamente las clases, en las que se basa la programación orientada a objetos.

Al declarar un objeto perteneciente a una determinada clase, se está reservando una zona de memoria⁸ donde se almacenarán los atributos y otros datos pertenecientes a dicho objeto. Lo que se almacena en el objeto en sí, es un puntero (referencia) a dicha zona de memoria.

Dentro de estos tipos pueden considerarse las interfaces, los `Strings` y los vectores, que son unas clases un tanto especiales, y que se verán en detalle posteriormente.

⁸ En realidad, el momento en el que se realiza la reserva física del espacio de memoria es cuando se instancia el objeto realizando la llamada a su constructor, y no en el momento de la declaración.

Visión General y elementos básicos del lenguaje.

Existe un tipo referencial especial nominado por la palabra reservada `null` que puede ser asignado a cualquier variable de cualquier clase y que indica que el puntero no tiene referencia a ninguna zona de memoria (el objeto no está inicializado).

Además, todos los tipos de datos simples vistos en el punto anterior pueden ser declarados como referenciales (objetos), ya que existen clases que los engloban.

Estas clases son:

Tipo de datos simple	Clase equivalente
<code>byte</code>	<code>java.lang.Byte</code>
<code>short</code>	<code>java.lang.Short</code>
<code>int</code>	<code>java.lang.Integer</code>
<code>long</code>	<code>java.lang.Long</code>
<code>float</code>	<code>java.lang.Float</code>
<code>double</code>	<code>java.lang.Double</code>
<code>char</code>	<code>java.lang.Character</code>
<code>boolean</code>	<code>java.lang.Boolean</code>

2.6 Identificadores.

Los identificadores son los nombres que se les da a las variables, clases, interfaces, atributos y métodos de un programa.

Reglas para la creación de identificadores:

1. Java hace distinción entre mayúsculas y minúsculas, por lo tanto, nombres o identificadores como var1, Var1 y VAR1 son distintos.
2. Pueden estar formados por cualquiera de los caracteres del código Unicode, por lo tanto, se pueden declarar variables con el nombre: añoDeCreación, raím, etc. (se acabó la época de los nombres de variable como ano_de_creacion), aunque eso sí, el primer carácter no puede ser un dígito numérico y no pueden utilizarse espacios en blanco ni símbolos coincidentes con operadores.
3. La longitud máxima de los identificadores es prácticamente ilimitada.
4. No puede ser una palabra reservada del lenguaje ni los valores lógicos true o false.
5. No pueden ser iguales a otra identificador declarado en el mismo ámbito.
6. Por convenio, los nombres de las variables y los métodos deberían empezar por una letra minúscula y los de las clases por mayúscula. Además, si el identificador está formado por varias palabras la primera se escribe en minúsculas (excepto para las clases) y el resto de palabras se hace empezar por mayúscula (por ejemplo: añoDeCreación). Estas reglas no son obligatorias, pero son convenientes ya que ayudan al proceso de codificación de un programa, así como a su legibilidad. Es más sencillo distinguir entre clases y métodos o variables.

Ejemplos válidos
añoDeNacimiento2
otra_variable
NombreDeUnaVariableMuyLargoNoImporta
BotónPulsación

Visión General y elementos básicos del lenguaje.

Ejemplos NO válidos	Razón
3valores	(número como primer carácter)
Dia&mes	&
Dos más	(espacio)
Dos-valores	-

Ya que el lenguaje permite identificadores todo lo largos que se desee, es aconsejable crearlos de forma que tengan sentido y faciliten su interpretación. El nombre ideal para un identificador es aquel que no se excede en longitud (lo más corto posible) siempre que califique claramente el concepto que intenta representar. Siempre dentro de unos límites; por ejemplo, no sería muy adecuado utilizar un identificador de un índice de un bucle como **indiceDeTalBucle** en lugar de simplemente **i**.

Hay que evitar identificadores como a1, a2, a3, a4, va1, xc32, xc21, xsda, ... y en general todos aquellos identificadores que no “signifiquen” nada.

2.7 Declaración de variables.

La declaración de una variable se realiza de la misma forma que en C. Siempre contiene el nombre (identificador de la variable) y el tipo de dato al que pertenece. El ámbito de la variable depende de la localización en el programa donde es declarada.

Ejemplo:

```
int x;
```

Las variables pueden ser inicializadas en el momento de su declaración, siempre que el valor que se les asigne coincida con el tipo de dato de la variable.

Ejemplo:

```
int x = 0;
```

2.8 *Ámbito de una variable.*

El ámbito de una variable es la porción de programa donde dicha variable es visible para el código del programa y, por tanto, referenciable.

El ámbito de una variable depende del lugar del programa donde es declarada, pudiendo pertenecer a cuatro categorías distintas.

- Variable local.
- Atributo.
- Parámetro de un método.
- Parámetro de un tratador de excepciones.

Como puede observarse, NO existen las variables globales. Esto no es un “defecto” del lenguaje sino todo lo contrario. La utilización de variables globales resulta peligrosa, ya que puede ser modificada en cualquier parte del programa y por cualquier procedimiento. Además, a la hora de utilizarlas hay que buscar dónde están declaradas para conocerlas y dónde son modificadas para evitar sorpresas en los valores que pueden contener.

Los ámbitos de las variables u objetos en Java siguen los criterios “clásicos”, al igual que en la mayoría de los lenguajes de programación como Pascal, C++, etc. No existen sorpresas.

Si una variable no ha sido inicializada, tiene un valor asignado por defecto. Este valor es, para las variables de tipo referencial (objetos), el valor `null`. Para las variables de tipo numérico, el valor por defecto es cero (`0`), las variables de tipo `char`, el valor `'\u0000'` y las variables de tipo `boolean`, el valor `false`.

2.8.1 *Variables locales.*

Una variable local se declara dentro del cuerpo de un método de una clase y es visible únicamente dentro de dicho método.

Se puede declarar en cualquier lugar del cuerpo, incluso después de instrucciones ejecutables, aunque es una buena costumbre declararlas justo al principio.

Visión General y elementos básicos del lenguaje.

NOTA: También pueden declararse variables dentro de un bloque parentizado por llaves { ... }. En ese caso, sólo serán “visibles” dentro de dicho bloque.



```
class Character {
    char ch;
    public Character(char c) {
        ch=c;
    }
    public void repetir(int num) {
        int i;
        for (i=0;i<num;i++)
            System.out.println(ch);
    }
}

class Ej1 {
    public static void main(String argumentos[]) {
        Character character;
        character = new Character('H');
        character.repetir(20);
    }
}
```

En este ejemplo existe una variable local: `int i;` definida en el método `repetir` de la clase `Character`, por lo tanto, únicamente es visible dentro del método `repetir`.

También existe una variable local en el método `main`. En este caso, la variable local es un objeto:

`Character character;` que sólo será visible dentro del método en el que está declarada (`main`).

Es importante hacer notar que una declaración como la anterior le indica al compilador el tipo de la variable `character` pero no crea un objeto. El operador que crea el objeto es `new`, que necesita como único parámetro el nombre del constructor (que será el procedimiento que asigna valor a ese objeto recién instanciado).

Cuando se pretende declarar múltiples variables del mismo tipo pueden declararse, en forma de lista, separadas por comas:

Visión General y elementos básicos del lenguaje.

Ejemplo:

```
int x,y,z;
```

- Declara tres variables x, y, z de tipo entero.
- Podrían haberse inicializado en su declaración de la forma:

```
int x=0,y=0,z=3;
```

No es necesario que se declaren al principio del método. Puede hacerse en cualquier lugar del mismo, incluso de la siguiente forma:

```
public void repetir(int num) {  
    for (int i=0;i<num;i++)  
        System.out.println(ch);  
}
```

En el caso de las variables locales, éstas no se inicializan con un valor por defecto, como se ha comentado en el punto anterior, sino que es necesario asignarles algún valor antes de poder utilizarlas en cualquier instrucción, de lo contrario el compilador genera un error, de tal forma que es **imposible** hacer uso de una variable local no inicializada sin que se percate de ello el compilador.

Las variables locales pueden ser anteceditas por la palabra reservada `final`⁹. En ese caso, sólo permiten que se les asigne un valor una única vez.

Ejemplo:

```
final int x=0;
```

No permitirá que a `x` se le asigne ningún otro valor. Siempre contendrá 0.

No es necesario que el valor se le asigne en el momento de la declaración, podría haberse inicializado en cualquier otro lugar, pero una sola vez:

⁹ Esta posibilidad no existe en el JDK 1.02. Aparece por primera vez en la versión 1.1.

Visión General y elementos básicos del lenguaje.

Ejemplo:

```
final int x;  
...  
x=y+2;
```

Después de la asignación `x=y+2`, no se permitirá asignar ningún otro valor a `x`.

2.8.2 Atributos.

Los atributos de una clase son las características que se van a tener en cuenta sobre un objeto y por lo tanto su ámbito está circunscrito, en principio, dentro de la clase a la cual caracterizan. Se declaran de la misma forma que las variables locales pero, además, pueden tener algunos modificadores que afectan al ámbito de los mismos y que se verán en otro capítulo (↪ 3.3 **Declaración de atributos**. Página 84).

En el ejemplo anterior, `ch` es un atributo de la clase `Character` y por lo tanto es “manipulable” en cualquier método de dicha clase, como de hecho ocurre en los métodos `repetir()` y `Carácter()`.

Para acceder a un atributo de un objeto desde algún método perteneciente a otra clase u objeto se antepone el nombre del objeto y un punto al nombre de dicho atributo. Por ejemplo:

```
character.ch
```

Con los nombres de los métodos se hace lo mismo.

Ejemplo:

```
character.repetir(20);
```

2.8.3 Parámetros de un método.

Los parámetros se declaran en la cabecera del método de la siguiente forma:

```
[Modificadores_de_método] Tipo_devuelto Nombre_de_método (lista_de_parámetros)
{
}
}
```

La *lista_de_parámetros* consiste en una serie de variables, separadas por comas y declarando el tipo al que pertenecen.

Ejemplo:

```
public static void miMétodo(int v1, int v2, float v3, String
v4, ClaseObjeto v5);
```

Nótese que aunque existan varios parámetros pertenecientes al mismo tipo o clase, no pueden declararse abreviadamente, como ocurre con las variables locales y los atributos, indicando el tipo y a continuación la lista de parámetros separados por comas. Así, es ilegal la siguiente declaración del método anterior:

```
public static void miMétodo(int v1, v2, float v3, String v4,
ClaseObjeto v5); (ILEGAL)
```

La declaración de un parámetro puede ir antecedita, como ocurre con las variables locales, por la palabra reservada `final`. En ese caso, el valor de dicho parámetro no podrá ser modificado en el cuerpo del método.

Los parámetros de un método pueden ser de dos tipos:

- **Variables de tipo simple de datos:** En este caso, el paso de parámetros se realiza siempre por valor. Es decir, el valor del parámetro de llamada no puede ser modificado en el cuerpo del método (El método trabaja con una copia del valor utilizado en la llamada).
- **Variables de tipo objeto (referencias):** En este caso, lo que realmente se pasa al método es un puntero al objeto y, por lo tanto, el

Visión General y elementos básicos del lenguaje.

valor del parámetro de llamada sí que puede ser modificado dentro del método (El método trabaja directamente con el valor utilizado en la llamada), a no ser que se anteponga la palabra reservada `final`.

Tipo del parámetro	Método de pase de parámetro
Tipo simple de datos (ejemplo: int, char, boolean, double, etc.)	POR VALOR
Tipo referencial (Objetos de una determinada clase, vectores y Strings)	POR REFERENCIA



```
class Objeto {
    int variable;
    public Objeto(int var){
        variable=var;
    }
}

class Parametros {
    public static void main (String argumentos[]) {
        int var1;
        Objeto obj;
        obj = new Objeto(1);
        var1 = 2;
        System.out.println("Valor del objeto = "+obj.variable);
        System.out.println("Valor de la variable = "+var1);
        modifica(var1,obj);
        System.out.println("-Después de llamar a modifica()-");
        System.out.println("Valor del objeto = "+obj.variable);
        System.out.println("Valor de la variable = "+var1);
    }
    static void modifica (int vv, Objeto oo) {
        vv++;
        oo.variable++;
    }
}
```

Visión General y elementos básicos del lenguaje.

La salida del programa sería la siguiente:

```
Valor del objeto = 1
```

```
Valor de la variable = 2
```

```
-Después de llamar a modifica()-
```

```
Valor del objeto = 2
```

```
Valor de la variable = 2
```

Como puede verse, después de la llamada, el valor del objeto `obj` sí ha sido modificado (se ha realizado un pase de parámetro por referencia), mientras que el valor de la variable `var1` no ha sido modificado (se ha realizado un paso de parámetro por valor).

2.9 Operadores.

Los operadores son partes indispensables en la construcción de expresiones.

Existen muchas definiciones técnicas para el término expresión. Puede decirse que una expresión es una combinación de operandos ligados mediante operadores. Los operandos pueden ser variables, constantes, funciones, literales, etc. y los operadores todos los enumerados en este punto.

2.9.1 Operadores aritméticos:

Operador	Formato	Descripción
+	<code>op1 + op2</code>	Suma aritmética de dos operandos
-	<code>op1 - op2</code> <code>-op1</code>	Resta aritmética de dos operandos Cambio de signo
*	<code>op1 * op2</code>	Multiplicación de dos operandos
/	<code>op1 / op2</code>	División entera de dos operandos
%	<code>op1 % op2</code>	Resto de la división entera (o módulo)
++	<code>++op1</code> <code>op1++</code>	Incremento unitario
--	<code>-- op1</code> <code>op1--</code>	Decremento unitario

El operador - puede utilizarse en su versión unaria (- op1) y la operación que realiza es la de invertir el signo del operando.

Como en C, los operadores unarios ++ y -- realizan un incremento y un decremento respectivamente. Estos operadores admiten notación prefija y postfija.

- `++op1`: En primer lugar realiza un incremento (en una unidad) de `op1` y después ejecuta la instrucción en la cual está inmerso.
- `op1++`: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un incremento (en una unidad) de `op1`.
- `--op1`: En primer lugar realiza un decremento (en una unidad) de `op1` y después ejecuta la instrucción en la cual está inmerso.

Visión General y elementos básicos del lenguaje.

- `op1--`: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un decremento (en una unidad) de `op1`.

La diferencia entre la notación prefija y la postfija no tiene importancia en expresiones en las que únicamente existe dicha operación:

`++contador;` es equivalente a: `contador++;`
`--contador;` `contador--;`

Es importante no emplear estos operadores en expresiones que contengan más de una referencia a la variable incrementada, puesto que esta práctica puede generar resultados erróneos fácilmente.

La diferencia es apreciable en instrucciones en las cuáles están incluidas otras operaciones.

Por ejemplo:

```
cont = 1;          cont = 1;
do {              do {
    ...}          ...}
while ( cont++ < 3 );  while ( ++cont < 3 );
```

En el primer caso, el bucle se ejecutará 3 veces, mientras que en el segundo se ejecutará 2 veces.

Otro ejemplo:

```
a = 1;          a = 1;
b = 2 + a++;   b = 2 + ++a;
```

- En el primer caso, después de las operaciones, `b` tendrá el valor 3 y `a` el valor 2.
- En el segundo caso, después de las operaciones, `b` tendrá el valor 4 y `a` el valor 2.

2.9.2 Operadores relacionales:

Operador	Formato	Descripción
>	op1 > op2	Devuelve <code>true</code> (cierto) si <code>op1</code> es mayor que <code>op2</code>
<	op1 < op2	Devuelve <code>true</code> (cierto) si <code>op1</code> es menor que <code>op2</code>
>=	op1 >= op2	Devuelve <code>true</code> (cierto) si <code>op1</code> es mayor o igual que <code>op2</code>
<=	op1 <= op2	Devuelve <code>true</code> (cierto) si <code>op1</code> es menor o igual que <code>op2</code>
==	op1 == op2	Devuelve <code>true</code> (cierto) si <code>op1</code> es igual a <code>op2</code>
!=	op1 != op2	Devuelve <code>true</code> (cierto) si <code>op1</code> es distinto de <code>op2</code>

Los operadores relacionales actúan sobre valores enteros, reales y caracteres (`char`); y devuelven un valor del tipo `boolean` (`true` o `false`).



```
class Relacional {
    public static void main(String arg[]) {
        double op1,op2;
        op1=1.34;
        op2=1.35;
        System.out.println("op1="+op1+" op2="+op2);
        System.out.println("op1>op2 = "+(op1>op2));
        System.out.println("op1<op2 = "+(op1<op2));
        System.out.println("op1==op2 = "+(op1==op2));
        System.out.println("op1!=op2 = "+(op1!=op2));
        char op3,op4;
        op3='a'; op4='b';
        System.out.println("'a'>'b' = "+(op3>op4));
    }
}
```

Visión General y elementos básicos del lenguaje.

```
}  
}
```

Salida por pantalla:

```
op1=1.34 op2=1.35  
op1>op2 = false  
op1<op2 = true  
op1==op2 = false  
op1!=op2 = true  
'a'>'b' = false
```

Nota: Los operadores `==` y `!=` actúan también sobre valores lógicos (`boolean`).

2.9.3 Operadores lógicos:

Operador	Formato	Descripción
<code>&&</code>	<code>op1 && op2</code>	Y lógico. Devuelve <code>true</code> (cierto) si son ciertos <code>op1</code> y <code>op2</code>
<code> </code>	<code>op1 op2</code>	O lógico. Devuelve <code>true</code> (cierto) si son ciertos <code>op1</code> o <code>op2</code>
<code>!</code>	<code>! op1</code>	Negación lógica. Devuelve <code>true</code> (cierto) si es <code>false op1</code> .

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso (`true` / `false`).

Por ejemplo, el siguiente programa:



```
class Bool {  
    public static void main ( String argumentos[] ) {  
        boolean a=true;
```

Visión General y elementos básicos del lenguaje.

```
boolean b=true;
boolean c=false;
boolean d=false;
System.out.println("true Y true   = " + (a && b) );
System.out.println("true Y false  = " + (a && c) );
System.out.println("false Y false = " + (c && d) );
System.out.println("true O true   = " + (a || b) );
System.out.println("true O false  = " + (a || c) );
System.out.println("false O false = " + (c || d) );
System.out.println("NO true   = " + !a);
System.out.println("NO false = " + !c);
System.out.println("(3 > 4) Y true = " + ((3 >4) && a) );
}
}
```

produciría la siguiente salida por pantalla:

```
true Y true   = true
true Y false  = false
false Y false = false
true O true   = true
true O false  = true
false O false = false
NO true   = false
NO false = true
(3 > 4) Y true = false
```

2.9.4 Operadores de bits:

Operador	Formato	Descripción
>>	op1 >> op2	Desplaza op1, op2 bits a la derecha
<<	op1 << op2	Desplaza op1, op2 bits a la izquierda
>>>	op1 >>> op2	Desplaza op1, op2 bits a la derecha (sin signo)

Visión General y elementos básicos del lenguaje.

<code>&</code>	<code>op1 & op2</code>	Realiza un Y (AND) a nivel de bits
<code> </code>	<code>op1 op2</code>	Realiza un O (OR) a nivel de bits
<code>^</code>	<code>op1 ^ op2</code>	Realiza un O exclusivo (XOR) a nivel de bits
<code>~</code>	<code>~op1</code>	Realiza el complemento de <code>op1</code> a nivel de bits.

Los operadores de bits actúan sobre valores enteros (`byte`, `short`, `int` y `long`) o caracteres (`char`).



```
class Bits {
    public static void main ( String argumentos[] ) {
        byte a=12;
        byte b=-12;
        byte c=6;
        System.out.println("12 >> 2 = " + ( a >> 2 ) );
        System.out.println("-12 >> 2 = " + ( b >> 2 ) );
        System.out.println("-12 >>> 2 = " + ( b >>> 2 ) );
        System.out.println("12 << 2 = " + ( a << 2 ) );
        System.out.println("-12 << 2 = " + ( b << 2 ) );
        System.out.println("12 & 6 = " + ( a & c ) );
        System.out.println("12 | 6 = " + ( a | c ) );
        System.out.println("12 ^ 6 = " + ( a ^ c ) );
        System.out.println("~12 = " + ~a);
    }
}
```

Visión General y elementos básicos del lenguaje.

Produciría la siguiente salida por pantalla:¹⁰

```
12 >> 2 = 3          00001100 (12)          00000011 (3)
-12 >> 2 = -3       11110100 (-12)         11111101 (-3)
-12 >>> 2 = 1073741821 11110100 (-12)
00111111 11111111 11111111 11111101
12 << 2 = 48        00001100 (12)          00110000 (48)
-12 << 2 = -48     11110100 (-12)         11001011 (-48)
12 & 6 = 4          00001100 & 00000110 00000100 (4)
12 | 6 = 14        00001100 | 00000110 00001110 (14)
12 ^ 6 = 10        00001100 ^ 00000110 00001010 (10)
~12 = -13          ~00001100          11110011 (-13)
```

2.9.5 Operadores de asignación:

El operador de asignación es el símbolo igual (=).

op1 = Expresión

Asigna el resultado de evaluar la expresión de la derecha a **op1**.

Además del operador de asignación existen unas abreviaturas cuando el operando que aparece a la izquierda del símbolo de asignación también aparece a la derecha del mismo:

¹⁰ Los números negativos se almacenan en Complemento a dos (c-2), lo que significa que para almacenar el número negativo se toma el positivo en binario, se cambian unos por ceros y viceversa, y después se le suma 1 en binario.

Visión General y elementos básicos del lenguaje.

Operador	Formato	Equivalencia
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>
<code>&=</code>	<code>op1 &= op2</code>	<code>op1 = op1 & op2</code>
<code> =</code>	<code>op1 = op2</code>	<code>op1 = op1 op2</code>
<code>^=</code>	<code>op1 ^= op2</code>	<code>op1 = op1 ^ op2</code>
<code>>>=</code>	<code>op1 >>= op2</code>	<code>op1 = op1 >> op2</code>
<code><<=</code>	<code>op1 <<= op2</code>	<code>op1 = op1 << op2</code>
<code>>>>=</code>	<code>op1 >>>= op2</code>	<code>op1 = op1 >>> op2</code>

2.9.6 Precedencia de operadores en Java:

Operadores postfijos	<code>[] . (paréntesis)</code>
Operadores unarios	<code>++expr --expr -expr ~ !</code>
Creación o conversión de tipo	<code>new (tipo)expr</code>
Multiplicación y división	<code>* / %</code>
Suma y resta	<code>+ -</code>
Desplazamiento de bits	<code><< >> >>></code>
Relacionales	<code>< > <= >=</code>
Igualdad y desigualdad	<code>== !=</code>
AND a nivel de bits	<code>&</code>
XOR a nivel de bits	<code>^</code>
OR a nivel de bits	<code> </code>
AND lógico	<code>&&</code>
OR lógico	<code> </code>
Condicionales al estilo C	<code>? :</code>
Asignación	<code>= += -= *= /= %= ^= &= = >>= <<= >>>=</code>

2.10 Constantes.

Las constantes en Java son declaradas como atributos de tipo `final`. (↪ Ver el punto: 3.3.2 Atributos `final`. En página 86).

2.11 Valores literales.

A la hora de tratar con valores de los tipos de datos simples (y `Strings`) se utiliza lo que se denomina “literales”. Los literales son elementos que sirven para representar un valor en el código fuente del programa.

En Java existen literales para los siguientes tipos de datos:

- Lógicos (`boolean`).
- Carácter (`char`).
- Enteros (`byte`, `short`, `int` y `long`).
- Reales (`double` y `float`).
- Cadenas de caracteres (`String`).

Literales lógicos

Son únicamente dos: las palabras reservadas `true` y `false`.

Ejemplo:

```
boolean activado = false;
```

Literales de tipo entero.

Los literales de tipo entero: `byte`, `short`, `int` y `long` pueden expresarse en decimal (base 10), octal (base 8) o hexadecimal (base 16). Además, puede añadirse al final del mismo la letra `L` para indicar que el entero es considerado como **long** (64 bits).

Visión General y elementos básicos del lenguaje.

- Enteros en el sistema decimal (base 10): En Java, el compilador identifica un entero decimal (base 10) al encontrar un número cuyo primer dígito es cualquier símbolo decimal excepto el cero (del **1** al **9**). A continuación pueden aparecer dígitos del **0** al **9**.
- Enteros en el sistema octal (base 8): Un número será considerado por el compilador como octal siempre que el primer dígito sea el **cero**. A continuación del cero puede aparecer cualquier dígito numérico comprendido entre **0** y **7**. Si aparece un **8** ó **9**, se generará un error por el compilador.
- Enteros en el sistema hexadecimal (base 16): Un número en hexadecimal empieza por **cero** seguido de la letra **x** (mayúscula o minúscula). A continuación puede aparecer cualquier dígito numérico comprendido entre **0** y **9** ó las letras **A,B,C,D,E** ó **F**, siendo indiferente el que las letras se encuentren en mayúsculas o minúsculas.

Ejemplos:

Sistema de numeración		
Decimal	Octal	Hexadecimal
34	042	0x22
962	01702	0x3c2
8186	017772	0x1FFA

La letra **L** al final de un literal de tipo entero puede aplicarse a cualquier sistema de numeración e indica que el número decimal sea tratado como un entero largo (de 64 bits). Esta letra **L** puede ser mayúscula o minúscula, aunque es aconsejable utilizar la mayúscula ya que de lo contrario puede confundirse con el dígito uno (**1**) en los listados.

Visión General y elementos básicos del lenguaje.

Ejemplo:

```
long max1 = 9223372036854775807L;    //valor máximo para un
                                     // entero largo (64 bits)
long max2 = 07777777777777777777L; // en este caso max1,
                                     // max2 y max3
long max3 = 0x7fffffffffffffffL; //almacenan el mismo valor.
```

Literales de tipo real.

Los literales de tipo real sirven para indicar valores `float` o `double`. A diferencia de los literales de tipo entero, no pueden expresarse en octal o hexadecimal.

Existen dos formatos de representación: mediante su parte entera, el punto decimal (.) y la parte fraccionaria; o mediante notación exponencial o científica:

Ejemplos equivalentes:

```
3.1415
0.31415e1
.31415e1
0.031415E+2
.031415e2
314.15e-2
31415E-4
```

Al igual que los literales que representan enteros, se puede poner una letra como sufijo. Esta letra puede ser una **F** o una **D** (mayúscula o minúscula indistintamente).

- **F** Trata el literal como de tipo `float`.
- **D** Trata el literal como de tipo `double`.

Ejemplo:

```
3.1415F
.031415d
```

Literales de tipo carácter

Los literales de tipo carácter se representan siempre entre comillas simples.

Entre las comillas simples puede aparecer:

- Un símbolo (letra) siempre que el carácter esté asociado a un código Unicode. Ejemplos: 'a', 'B', '{', 'ñ', 'á'.
- Una “secuencia de escape”. Las secuencias de escape son combinaciones del símbolo contrabarra \ seguido de una letra, y sirven para representar caracteres que no tienen una equivalencia en forma de símbolo.

Las posibles secuencias de escape son:

Secuencia de escape	Significado
'\''	Comilla simple.
'\"'	Comillas dobles.
'\\'	Contrabarra.
'\b'	Backspace (Borrar hacia atrás).
'\n'	Cambio de línea.
'\f'	Form feed.
'\r'	Retorno de carro.
'\t'	Tabulador.

- La contrabarra \ seguida de 3 dígitos octales. Ejemplo:

'\141' equivalente a 'a'
'\101' equivalente a 'A'
'\042' equivalente a '\"'

Visión General y elementos básicos del lenguaje.

- La contrabarra \ seguida de u y 4 dígitos hexadecimales (el código Unicode). Ejemplo:

'\u0061' equivalente a '\141' equivalente a 'a'
'\u0041' equivalente a '\101' equivalente a 'A'
'\u0022' equivalente a '\042' equivalente a '\"'

Siempre que sea posible, es aconsejable utilizar las secuencias de escape en lugar del código Unicode (en octal o hexadecimal), ya que el precompilador de Java sustituye los códigos Unicode por su valor y si se ha utilizado, por ejemplo, el código Unicode equivalente al cambio de línea, puede interpretarse como tal y producir un error al creer que se abren comillas y se cierran en la línea siguiente del fichero.

Literales de tipo String.

Los *Strings* o cadenas de caracteres no forman parte de los tipos de datos elementales en Java, sino que son instanciados a partir de la clase `java.lang.String`, pero aceptan su inicialización a partir de literales de este tipo, por lo que se tratan en este punto.

Un literal de tipo *string* va encerrado entre comillas dobles (“ ”) y debe estar incluido completamente en una sola línea del programa fuente (no puede dividirse en varias líneas). Entre las comillas dobles puede incluirse cualquier carácter del código Unicode (o su código precedido del carácter \) además de las secuencias de escape vistas anteriormente en los literales de tipo carácter. Así, por ejemplo, para incluir un cambio de línea dentro de un literal de tipo *string* deberá hacerse mediante la secuencia de escape `\n` :

Ejemplo:

```
System.out.println("Primera línea\nSegunda línea del string\n");  
System.out.println("Hol\u0061");
```

Visión General y elementos básicos del lenguaje.

La visualización del *string* anterior mediante `println()` produciría la siguiente salida por pantalla:

```
Primera línea
Segunda línea del string
Hola
```

La forma de incluir los caracteres: comillas dobles (“) y contrabarra (\) es mediante las secuencias de escape `\"` y `\\` respectivamente (o mediante su código Unicode precedido de `\\`).

Si el *string* es demasiado largo y debe dividirse en varias líneas en el fichero fuente, puede utilizarse el operador de concatenación de *strings* `+` de la siguiente forma:

```
"Este String es demasiado largo para estar en una línea del " +
"fichero fuente y se ha dividido en dos."
```

2.12 Estructuras de control.

Las estructuras de control son construcciones hechas a partir de palabras reservadas del lenguaje que permiten modificar el flujo de ejecución de un programa. De este modo, pueden crearse construcciones de alternativas y bucles de repetición de bloques de instrucciones.

Hay que señalar que un bloque de instrucciones se encontrará encerrado mediante llaves `{.....}` si existe más de una instrucción.

2.12.1 Estructuras alternativas.

Las estructuras alternativas son construcciones que permiten alterar el flujo secuencial de un programa, de forma que en función de una condición o el valor de una expresión, el mismo pueda ser desviado en una u otra alternativa de código.

Visión General y elementos básicos del lenguaje.

Las estructuras alternativas disponibles en Java son:

- Alternativa `if-else`.
- Alternativa `switch`.

2.12.1.1 `if-else`.

Forma simple:

```
if (expresión)  
    Bloque instrucciones
```

El bloque de instrucciones se ejecuta si, y sólo si, la *expresión* (que debe ser lógica) se evalúa a `true`, es decir, se cumple una determinada condición.



```
if (cont == 0)  
    System.out.println("he llegado a cero");
```

La instrucción `System.out.println("he llegado a cero");` sólo se ejecuta en el caso de que `cont` contenga el valor cero.

Con cláusula `else`:

```
if (expresión)  
    Bloque instrucciones 1  
else  
    Bloque instrucciones 2
```

Visión General y elementos básicos del lenguaje.

El bloque de instrucciones 1 se ejecuta si, y sólo si, la *expresión* se evalúa a `true`. Y en caso contrario, si la expresión se evalúa a `false`, se ejecuta el bloque de instrucciones 2.



```
if (cont == 0)
    System.out.println("he llegado a cero");
else
    System.out.println("no he llegado a cero");
```

Si `cont` vale cero, se mostrará en el mensaje “he llegado a cero”. Si `cont` contiene cualquier otro valor distinto de cero, se mostrará el mensaje “no he llegado a cero”.

if-else anidados:

En muchas ocasiones, se anidan estructuras alternativas `if-else`, de forma que se pregunte por una condición si anteriormente no se ha cumplido otra sucesivamente. Por ejemplo: supongamos que realizamos un programa que muestra la nota de un alumno en la forma (insuficiente, suficiente, bien, notable o sobresaliente) en función de su nota numérica. Podría codificarse de la siguiente forma:



```
class Nota {
    public static void main (String argumentos[]) {
        int nota;
        if (argumentos.length<1) {
            // argumentos.length contiene el número de elementos
            // del array argumentos, que contiene los parámetros
            // suministrados en la línea de comandos.
            System.out.println("Uso: Nota num");
            System.out.println("Donde num = n° entre 0 y 10");
        }
        else
        {
            nota=Integer.valueOf(argumentos[0]).intValue();
            // la instrucción anterior convierte un
```

Visión General y elementos básicos del lenguaje.

```
// String a entero.
if (nota<5)
    System.out.println("Insuficiente");
else
    if (nota<6)
        System.out.println("Suficiente");
    else
        if (nota<7)
            System.out.println("Bien");
        else
            if (nota<9)
                System.out.println("Notable");
            else
                System.out.println("Sobresaliente");
    }
}
```

En Java, como en C y a diferencia de otros lenguajes de programación, en el caso de que el bloque de instrucciones conste de una sola instrucción (no necesita ser parentizado), se pone un punto y coma (;) justo antes de la cláusula `else`.

2.12.1.2 switch.

Forma simple:

```
switch (expresión) {
    case valor1: instrucciones1;
    case valor2: instrucciones2;

    case valorN: instruccionesN;
}
```

En este caso, a diferencia del anterior, si *instrucciones1* ó *instrucciones2* ó *instruccionesN* están formados por un bloque de instrucciones sencillas, no es necesario parentizarlas mediante las llaves ({ ... }).

En primer lugar se evalúa la *expresión* cuyo resultado puede ser un valor de cualquier tipo. El programa comprueba el primer valor (*valor1*). En el caso de que

Visión General y elementos básicos del lenguaje.

el valor resultado de la expresión coincida con *valor1*, se ejecutarán las *instrucciones1*. Pero ¡ojo! También se ejecutarían las instrucciones *instrucciones2* .. *instruccionesN* hasta encontrarse con la palabra reservada `break`. Si el resultado de la expresión no coincide con *valor1*, evidentemente no se ejecutarían *instrucciones1*, se comprobaría la coincidencia con *valor2* y así sucesivamente hasta encontrar un valor que coincida o llegar al final de la construcción `switch`. En caso de que no exista ningún valor que coincida con el de la *expresión*, no se ejecuta ninguna acción.

Si lo que se desea es que únicamente se ejecuten las instrucciones asociadas a cada valor y no todas las de los demás `case` que quedan por debajo, la construcción `switch` sería la siguiente:

```
switch (expresión) {  
    case valor1: instrucciones1;  
                break;  
    case valor2: instrucciones2;  
                break;  
  
    case valorN: instruccionesN;  
}
```



```
class DiaSemana {  
    public static void main(String argumentos[]) {  
        int día;  
        if (argumentos.length<1) {  
            System.out.println("Uso: DiaSemana num");  
            System.out.println("Donde num = nº entre 1 y 7");  
        }  
        else {  
            día=Integer.valueOf(argumentos[0]).intValue();  
            switch (día) {  
                case 1: System.out.println("Lunes");  
                        break;  
                case 2: System.out.println("Martes");  
                        break;  
                case 3: System.out.println("Miércoles");  
                        break;  
                case 4: System.out.println("Jueves");  
                        break;  
            }  
        }  
    }  
}
```

Visión General y elementos básicos del lenguaje.

```
        case 5: System.out.println("Viernes");
                break;
        case 6: System.out.println("Sábado");
                break;
        case 7: System.out.println("Domingo");
    }
}
}
```

Nótese que en el caso de que se introduzca un valor no comprendido entre 1 y 7, no se realizará ninguna acción. Esto puede corregirse utilizando el siguiente formato:

Con cláusula por defecto:

```
switch (expresión) {
    case valor1: instrucciones1;
    case valor2: instrucciones2;

    case valorN: instruccionesN;
    default: instruccionesPorDefecto;
}
```

En este caso, *instruccionesPorDefecto* se ejecutarán en el caso de que ningún valor *case* coincida con el valor de *expresión*. O en caso de ejecutar algunas instrucciones en alguno de los *case*, que no haya ninguna instrucción *break* desde ese punto hasta la cláusula *default*.



```
class DiasMes {
    public static void main(String argumentos []) {
        int mes;
        if (argumentos.length<1) {
            System.out.println("Uso: DiasMes num");
            System.out.println("Donde num = nº del mes");
        }
    }
}
```

Visión General y elementos básicos del lenguaje.

```
else {
    mes=Integer.valueOf(argumentos[0]).intValue();
    switch (mes) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12: System.out.println("El mes "+mes +
            " Tiene 31 días");
                break;
        case 4:
        case 6:
        case 9:
        case 11: System.out.println("El mes "+mes +
            " Tiene 30 días");
                break;
        case 2: System.out.println("El mes "+mes +
            " Tiene 28 ó 29 días");
                break;
        default: System.out.println("El mes "+mes +
            " no existe");
    }
}
}
```

En este ejemplo, únicamente se ejecutará la cláusula `default` en el caso en que el valor de `mes` sea distinto a un número comprendido entre 1 y 12, ya que todas estas posibilidades se encuentran contempladas en las respectivas cláusulas `case` y, además, existe un `break` justo antes de la cláusula `default`, por lo que en ningún otro caso se ejecutará la misma.

2.12.2 Bucles.

Los bucles son estructuras de repetición. Bloques de instrucciones que se repiten un número de veces mientras se cumpla una condición o hasta que se cumpla una condición.

Visión General y elementos básicos del lenguaje.

Existen tres construcciones para estas estructuras de repetición:

- Bucle `for`.
- Bucle `do-while`.
- Bucle `while`.

Como regla general puede decirse que se utilizará el bucle `for` cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones. Se utilizará el bucle `do-while` cuando no se conoce exactamente el número de veces que se ejecutará el bucle pero se sabe que por lo menos se ha de ejecutar una. Se utilizará el bucle `while-do` cuando es posible que no deba ejecutarse ninguna vez.

Estas reglas son generales y algunos programadores se sienten más cómodos utilizando principalmente una de ellas. Con mayor o menor esfuerzo, puede utilizarse cualquiera de ellas indistintamente.

2.12.2.1 Bucle `for`.

**`for (inicialización ; condición ; incremento)`
`bloque instrucciones`**

- La cláusula *inicialización* es una instrucción que se ejecuta una sola vez al inicio del bucle, normalmente para inicializar un contador.
- La cláusula *condición* es una expresión lógica, que se evalúa al inicio de cada nueva iteración del bucle. En el momento en que dicha expresión se evalúe a `false`, se dejará de ejecutar el bucle y el control del programa pasará a la siguiente instrucción (a continuación del bucle `for`).
- La cláusula *incremento* es una instrucción que se ejecuta en cada iteración del bucle como si fuera la última instrucción dentro del bloque de instrucciones. Generalmente se trata de una instrucción de incremento o decremento de alguna variable.

Cualquiera de estas tres cláusulas puede estar vacía, aunque SIEMPRE hay que poner los puntos y coma (;).

Visión General y elementos básicos del lenguaje.

El siguiente programa muestra en pantalla la serie de Fibonacci¹¹ hasta el término que se indique al programa como argumento en la línea de comandos. (NOTA: siempre se mostrarán, por lo menos, los dos primeros términos).



```
class Fibonacci {
    public static void main(String argumentos[]) {
        int numTerm,v1=1,v2=1,aux,cont;
        if (argumentos.length<1) {
            System.out.println("Uso: Fibonacci num");
            System.out.println("Donde num = n° de términos");
        }
        else {
            numTerm=Integer.valueOf(argumentos[0]).intValue();
            System.out.print("1,1");
            for (cont=2;cont<numTerm;cont++) {
                aux=v2;
                v2+=v1;
                v1=aux;
                System.out.print(", "+v2);
            }
            System.out.println();
        }
    }
}
```

2.12.2.2 Bucle *do-while*.

```
do
    bloque instrucciones
while (Expresión);
```

En este tipo de bucle, *bloque instrucciones* se ejecuta siempre una vez por lo menos, y ese bloque de instrucciones se ejecutará mientras *Expresión* se evalúe a **true**. Por lo tanto, entre las instrucciones que se repiten deberá existir alguna que, en algún momento, haga que la *Expresión* se evalúe a **false**, de lo contrario el bucle sería infinito.

¹¹ La serie de Fibonacci es una serie de números enteros que comienza con 1, 1. A partir del tercer término, el siguiente término de la serie se calcula como la suma de los dos anteriores. De esta forma, la serie comienza con los siguientes enteros: 1,1,2,3,5,8,13,21,34,55, ...

Visión General y elementos básicos del lenguaje.

Ejemplo: El mismo que antes (Fibonacci).



```
class Fibonacci2 {
    public static void main(String argumentos[]) {
        int numTerm,v1=0,v2=1,aux,cont=1;
        if (argumentos.length<1) {
            System.out.println("Uso: Fibonacci num");
            System.out.println("Donde num = n° de términos");
        }
        else {
            numTerm=Integer.valueOf(argumentos[0]).intValue();
            System.out.print("1");
            do {
                aux=v2;
                v2+=v1;
                v1=aux;
                System.out.print(", "+v2);
            } while (++cont<numTerm);
            System.out.println();
        }
    }
}
```

En este caso únicamente se muestra el primer término de la serie antes de iniciar el bucle, ya que el segundo siempre se mostrará, porque el bucle do-while siempre se ejecuta una vez por lo menos.

2.12.2.3 Bucle *while*.

while (Expresión)
bloque instrucciones

Al igual que en el bucle do-while del apartado anterior, el bloque de instrucciones se ejecuta mientras se cumple una condición (mientras *Expresión* se evalúe a **true**), pero en este caso, la condición se comprueba ANTES de empezar a ejecutar por primera vez el bucle, por lo que si *Expresión* se evalúa a **false** en la primera iteración, entonces el bloque de instrucciones no se ejecutará ninguna vez.

Visión General y elementos básicos del lenguaje.

Ejemplo: Fibonacci:



```
class Fibonacci3 {
    public static void main(String argumentos[]) {
        int numTerm,v1=1,v2=1,aux,cont=2;
        if (argumentos.length<1) {
            System.out.println("Uso: Fibonacci num");
            System.out.println("Donde num = n° de términos");
        }
        else {
            numTerm=Integer.valueOf(argumentos[0]).intValue();
            System.out.print("1,1");
            while (cont++<numTerm) {
                aux=v2;
                v2+=v1;
                v1=aux;
                System.out.print(", "+v2);
            }
            System.out.println();
        }
    }
}
```

Como puede comprobarse, las tres construcciones de bucle (`for`, `do-while` y `while`) pueden utilizarse indistintamente realizando unas pequeñas variaciones en el programa.

2.12.3 Saltos.

En Java existen dos formas de realizar un salto incondicional en el flujo “normal” de un programa. A saber, las instrucciones `break` y `continue`.

2.12.3.1 `break`.

La instrucción `break` sirve para abandonar una estructura de control, tanto de las alternativas (`if-else` y `switch`) como de las repetitivas o bucles (`for`, `do-while` y `while`). En el momento que se ejecuta la instrucción `break`, el control del programa sale de la estructura en la que se encuentra.

Visión General y elementos básicos del lenguaje.



```
class Break {
    public static void main(String argumentos[]) {
        int i;
        for (i=1; i<=4; i++) {
            if (i==3) break;
            System.out.println("Iteración: "+i);
        }
    }
}
```

Este ejemplo produciría la siguiente salida por pantalla:

```
Iteración: 1
Iteración: 2
```

Aunque el bucle, en principio indica que se ejecute 4 veces, en la tercera iteración, *i* contiene el valor 3, se cumple la condición del `if (i==3)` y por lo tanto se ejecuta el `break` y se sale del bucle `for`.

2.12.3.2 *continue*.

La instrucción `continue` sirve para transferir el control del programa desde la instrucción `continue` directamente a la cabecera del bucle (`for`, `do-while` o `while`) donde se encuentra.



```
class Continue {
    public static void main(String argumentos[]) {
        int i;
        for (i=1; i<=4; i++) {
            if (i==3) continue;
            System.out.println("Iteración: "+i);
        }
    }
}
```

Visión General y elementos básicos del lenguaje.

Este programa es muy similar al anterior, pero en lugar de utilizar la instrucción `break`, se ha utilizado `continue`. El resultado es el siguiente:

```
Iteración: 1
Iteración: 2
Iteración: 4
```

Puede comprobarse la diferencia con respecto al resultado del ejemplo del apartado anterior. En este caso no se abandona el bucle, sino que se transfiere el control a la cabecera del bucle donde se continúa con la siguiente iteración.

Tanto el salto `break` como en el salto `continue`, pueden ser evitados mediante distintas construcciones pero en ocasiones esto puede empeorar la legibilidad del código. De todas formas existen programadores que odian este tipo de saltos y no los utilizan en ningún caso.

2.13 Vectores.

Para manejar colecciones de objetos del mismo tipo estructurados en una sola variable se utilizan los vectores¹². En Java, los vectores son en realidad objetos y por lo tanto se puede llamar a sus métodos (como se verá en el capítulo siguiente).

Existen dos formas equivalentes de declarar vectores en Java:

- 1) ***tipo nombreDelVector[];***
- 2) ***tipo[] nombreDelVector;***

Ejemplo:

```
int vector1[], vector2[], entero; //entero no es un vector
int[] otroVector;
```

También pueden utilizarse vectores de más de una dimensión:

¹² También llamados arrays, matrices o tablas.

Visión General y elementos básicos del lenguaje.

Ejemplo:

```
int matriz[][];  
int[][] otraMatriz;
```

Los vectores, al igual que las demás variables pueden ser inicializados en el momento de su declaración. En este caso, no es necesario especificar el número de elementos máximo reservado. Se reserva el espacio justo para almacenar los elementos añadidos en la declaración.

Ejemplo:

```
String Días[]={ "Lunes", "Martes", "Miércoles", "Jueves",  
               "Viernes", "Sábado", "Domingo" };
```

Una simple declaración de un vector no reserva espacio en memoria, a excepción del caso anterior, en el que sus elementos obtienen la memoria necesaria para ser almacenados. Para reservar la memoria hay que llamar explícitamente a un constructor `new`¹³ de la siguiente forma:

`new tipoElemento[numElementos];`

Ejemplo:

```
int matriz[][];  
matriz = new int[4][7];
```

También se puede indicar el número de elementos durante su declaración:

Ejemplo:

```
int vector[] = new int[5];
```

¹³ ↪ Véase punto En la página 106.

Visión General y elementos básicos del lenguaje.

Para hacer referencia a los elementos particulares del vector, se utiliza el identificador del vector junto con el índice del elemento entre corchetes. El índice del primer elemento es el cero (0) y el del último, el número de elementos menos uno.

Ejemplo:

```
j = vector[0];    vector[4] = matriz[2][3];14
```

El intento de acceder a un elemento fuera del rango de la matriz, a diferencia de lo que ocurre en C, provoca una excepción (error) que, de no ser manejado por el programa, será el compilador quien aborte la operación.

Para obtener el número de elementos de un vector en tiempo de ejecución se accede al atributo de la clase vector `length`. No olvidemos que los vectores en Java son tratados como un objeto.



```
class Array1 {
    public static void main (String argumentos[]) {
        String colores[] = {"Rojo", "Verde", "Azul",
                            "Amarillo", "Negro"};

        int i;
        for (i=0; i<colores.length; i++)
            System.out.println(colores[i]);
    }
}
```

El ejemplo anterior produce la siguiente salida por pantalla:

```
Rojo
Verde
Azul
Amarillo
Negro
```

¹⁴ Para acceder a un elemento en un vector multidimensional, a diferencia de otros lenguajes, los índices van entre corchetes por separado y NO de la forma: `matriz[2,3]`

Visión General y elementos básicos del lenguaje.

Veamos otro ejemplo que calcula la letra del NIF¹⁵ a partir de un DNI que se suministra al programa en la línea de comandos:



```
class LetraNif {
    public static void main(String argumentos[]) {
        int dni;
        char tabla[]={'T','R','W','A','G','M','Y','F','P',
                    'D','X','B','N','J','Z','S','Q','V',
                    'H','L','C','K','E'};
        if (argumentos.length<1)
            System.out.println("Uso: LetraNif dni");
        else {
            dni=Integer.valueOf(argumentos[0]).intValue();
            System.out.println(dni+"-"+tabla[dni%23]);
        }
    }
}
```

La ejecución siguiente:

```
java LetraNif 123456789
```

Produce la salida por pantalla:

```
123456789-B
```

NOTA: Los Strings o cadenas de caracteres en Java son objetos pertenecientes a las clases [String](#) o [StringBuffer](#) y no vectores de tipo [char](#).

¹⁵ La letra de un NIF se puede averiguar de la siguiente forma: Se calcula el resto de dividir el DNI entre 23. Si el resto es cero, la letra del NIF es la T; si el resto es 1, la letra es la R; 2 W, 3 A ... y así sucesivamente conforme a una tabla que se encuentra declarada en forma de vector en el ejemplo.

2.14 CUESTIONES

1. ¿Cómo se denominan las variables que se declaran fuera de una clase?
2. ¿Cuál es la diferencia entre los conceptos atributo y método de una clase?
3. ¿Cuál es el nombre del método que debemos incluir en alguna clase para que el programa generado sea ejecutable?
4. ¿Qué diferencias crees que existen entre los tipos de datos básicos y los referenciales?
5. ¿Con qué carácter no pueden comenzar los nombres de variables (o de métodos)? ¿Cuál es la recomendación en Java para los nombres de atributos y métodos respecto del uso de letras mayúsculas y minúsculas?
6. Si deseas realizar un bucle que se ejecutará un número conocido de veces. ¿Qué construcción del lenguaje emplearás? ¿Y si desconoces el número de veces?
7. ¿Cuál es la diferencia entre las instrucciones `break` y `continue`?
8. ¿Qué código de representación emplea Java para los caracteres?
9. ¿Cómo se pasan los parámetros de tipos básicos a un método, por valor o por referencia?

3. LAS CLASES EN JAVA.

Como se vio en un principio en el capítulo anterior, todo en Java son clases (objetos). Si no se tienen claros los conceptos básicos de la programación orientada a objetos, antes de seguir adelante sería más que aconsejable volver la vista hacia el punto ↪ 1.7. Programación orientada a objetos. En la página 20 en caso de habérselo saltado.

Desde un punto de vista simplista, una clase es un conjunto de valores (atributos) junto con las funciones y procedimientos que operan sobre los mismos (métodos), todo ello tratado como una entidad. Estas clases constituyen los bloques principales en los cuales se encuentra contenido el código.

En un fichero fuente puede declararse una o más clases:

```
class Clase1 {  
}  
  
class Clase2 {  
}  
  
class ClaseN {  
}
```

El cuerpo de las clases comienza con una llave abierta ({) y termina con una llave cerrada (}).

```
Declaración de clase {  
Cuerpo de clase  
}
```

Las clases en Java.

La declaración de una clase define un tipo de dato referencial.

Dentro del cuerpo de la clase se declaran los atributos de la clase y los métodos.

3.1 Declaración de clase.

La declaración mínima para una clase es la siguiente:

```
class NombreClase
```

Una declaración de este tipo indica que la clase no descende de ninguna otra, aunque en realidad, todas las clases declaradas en un programa escrito en Java son descendientes, directa o indirectamente, de la clase `Object` que es la raíz de toda la jerarquía de clases en Java.



```
class ObjetoSimpleCreado {
    String variable = "Una variable";
    int entero = 14;
    public String obtnerString() {
        return variable;
    }
}

class ObjetoSimple {
    public static void main(String arumentos[]) {
        ObjetoSimpleCreado varObj = new ObjetoSimpleCreado();
        System.out.println(varObj.toString());
    }
}
```

Muestra en pantalla la siguiente línea de texto:

```
ObjetoSimpleCreado@13937d8
```

En este caso, la clase `ObjetoSimpleCreado` ha sido declarada como no descendiente de ninguna otra clase, pero a pesar de ello, hereda de la superclase

Las clases en Java.

`Object` (`java.lang.Object`) todos sus métodos, entre los que se encuentran el método `toString()` que, en este caso, devuelve el siguiente valor: "ObjetoSimpleCreado@13937d8" (el nombre de la clase junto con el puntero al objeto). Este método, que heredan todas las clases que puedan declararse, debería ser redefinido por el programador para mostrar un valor más significativo.

Si en lugar de la instrucción `System.out.println(varObj.toString());` se hubiera utilizado la siguiente: `System.out.println(varObj.obtenerString())` la salida por pantalla habría sido:

```
Una variable
```

3.1.1 Declaración de la superclase (herencia).

La superclase es la clase de la cual hereda otra clase todos sus atributos y métodos. La forma de declarar que una clase hereda de otra es:

```
class NombreClase extends NombreSuperclase
```

Ejemplo:

```
class Nif extends Dni
```

Declara una clase `Nif` que hereda todos los atributos y los métodos de la clase `Dni`.

3.1.2 Lista de interfaces.

Una interface es un conjunto de constantes y métodos, pero de éstos últimos únicamente el formato, no su implementación. Cuando una clase declara una lista de interfaces, asume que se van a redefinir todos los métodos definidos en la interface. Esta posibilidad corrige el inconveniente de que, en Java, no pueden declararse clases descendientes de más de una superclase.

```
class NombreClase implements Interface1, Interface2, , InterfaceN
```

Las clases en Java.

Por ejemplo:

```
class Nif extends Dni implements OperacionesAritméticas,  
OperacionesLógicas
```

En la interface `OperacionesAritméticas` pueden estar definidos, por ejemplo, los métodos `suma()`, `resta()`, etc. Mediante esta declaración, comprometo a la clase `Nif` (más bien a su programador) a redefinir los métodos `suma()`, `resta()`, etc., de lo contrario el compilador mostrará el correspondiente mensaje de error. También deberán redefinirse los métodos de la interfaz `OperacionesLógicas`.

☞ Ver punto 3.7 **Interfaces**. En página 116 para declarar interfaces.

3.1.3 Modificadores de clase.

Los modificadores de clase son palabras reservadas que se anteponen a la declaración de clase. Los modificadores posibles son los siguientes:

- `public`.
- `abstract`.
- `final`.

La sintaxis general es la siguiente:

```
modificador class NombreClase [ extends NombreSuperclase ]  
[ implements listaDeInterfaces ]
```

Si no se especifica ningún modificador de clase, la clase será visible en todas las declaradas en el mismo paquete¹⁶. Si no se especifica ningún paquete, se considera que la clase pertenece a un paquete por defecto al cual pertenecen todas las clases que no declaran explícitamente el paquete al que pertenecen.

¹⁶ ☞ Ver el punto correspondiente 3.8 **Los paquetes**, en la página 120.

3.1.3.1 *public*.

Cuando se crean varias clases que se agrupan formando un paquete (*package*), únicamente las clases declaradas `public` pueden ser accedidas desde otro paquete.

Toda clase `public` debe ser declarada en un fichero fuente con el nombre de esa clase pública: `NombreClase.java`. De esta afirmación se deduce que en un fichero fuente puede haber más de una clase, pero sólo una con el modificador `public`.

3.1.3.2 *abstract*.

Las clases abstractas no pueden ser instanciadas¹⁷. Sirven únicamente para declarar subclases que deben redefinir aquellos métodos que han sido declarados `abstract`. Esto no quiere decir que todos los métodos de una clase abstracta deban ser abstractos¹⁸, incluso es posible que ninguno de ellos lo sea. Aún en este último caso, la clase será considerada como abstracta y no podrán declararse objetos de esta clase.

Cuando alguno de los métodos de una clase es declarado abstracto, la clase debe ser obligatoriamente abstracta, de lo contrario, el compilador genera un mensaje de error.



```
abstract class Animal {
    String nombre;
    int patas;
    public Animal(String n, int p) {
        nombre=n;
        patas=p;
    }
    abstract void habla();
    // método abstracto que debe ser redefinido por las subclases
```

¹⁷ Instanciar una clase significa declarar un objeto cuyo tipo sea esa clase.

¹⁸ Un método abstracto únicamente declara su signatura o cabecera (su nombre y parámetros) y no el cuerpo del método. Se verá en el punto correspondiente 3.4.1.2 **Métodos `abstract`**. en página **¡Error! Marcador no definido.**

Las clases en Java.

```
}  
  
class Perro extends Animal {  
// La clase perro es una subclase de la clase abstracta Animal  
String raza;  
public Perro(String n, int p, String r) {  
    super(n,p);  
    raza=r;  
}  
public void habla() {  
// Este método es necesario redefinirlo para poder instanciar  
// objetos de la clase Perro  
    System.out.println("Me llamo "+nombre+": GUAU, GUAU");  
    System.out.println("mi raza es "+raza);  
}  
}  
  
class Gallo extends Animal {  
// La clase Gallo es una subclase de la clase abstracta Animal  
public Gallo(String n, int p) {  
    super(n,p);  
}  
public void habla() {  
// Este método es necesario redefinirlo para poder instanciar  
// objetos de la clase Gallo  
    System.out.println("Soy un Gallo, Me llamo "+nombre);  
    System.out.println("Kikirikiiii");  
}  
}  
  
class Abstracta {  
    public static void main(String argumentos[]) {  
        Perro toby = new Perro("Toby",4,"San Bernardo");  
        Gallo kiko = new Gallo("Kiko",2);  
        kiko.habla();  
        System.out.println();  
        toby.habla();  
    }  
}
```

Salida por pantalla del programa:

```
Soy un Gallo, Me llamo Kiko  
Kikirikiiii
```

```
Me llamo Toby: GUAU, GUAU  
mi raza es San Bernardo
```

Las clases en Java.

El intento de declarar un objeto del tipo `Animal`, que es `abstract`, habría generado un mensaje de error por el compilador.

Las clases abstractas se crean para ser superclases de otras clases. En este ejemplo, se ha declarado el método `habla()` como abstracto porque queremos que todos los animales puedan hablar, pero no sabemos qué es lo que van a decir (qué acciones se van a realizar), por lo que es declarada de tipo `abstract`. Las clases que heredan de `Animal` deben implementar un método `habla()` para poder heredar las características de `Animal`.

3.1.3.3 *final*.

Una clase declarada `final` impide que pueda ser superclase de otras clases. Dicho de otra forma, ninguna clase puede heredar de una clase `final`.

Esto es importante cuando se crean clases que acceden a recursos del sistema operativo o realizan operaciones de seguridad en el sistema. Si estas clases no se declaran como `final`, cualquiera podría redefinirlas y aprovecharse para realizar operaciones sólo permitidas a dichas clases pero con nuevas intenciones, posiblemente oscuras.

A diferencia del modificador `abstract`, pueden existir en la clase métodos `final` sin que la clase que los contiene sea `final` (sólo se protegen algunos métodos de la clase que no pueden ser redefinidos)¹⁹.

Una clase no puede ser a la vez `abstract` y `final` ya que no tiene sentido, pero sí que puede ser `public abstract` o `public final`.

3.2 *El cuerpo de la clase.*

Una vez declarada la clase, se declaran los atributos y los métodos de la misma dentro del cuerpo.

¹⁹ ↪ Ver ejemplo en el punto 3.4.1.3 **Métodos final**. en página 99.

Las clases en Java.

```
Declaración de clase {  
    Declaración de atributos  
    Declaración de clases interiores  
    Declaración de Métodos  
}
```

La declaración de clases interiores (también conocidas como clases anidadas) no es imprescindible para programar en Java..

3.3 Declaración de atributos.

Los atributos sirven, en principio, para almacenar valores de los objetos que se instancian a partir de una clase.

La sintaxis general es la siguiente:

```
[modificadorDeÁmbito] [static] [final] [transient] [volatile] tipo  
    nombreAtributo
```

Existen dos tipos generales de atributos:

- Atributos de objeto.
- Atributos de clase.

Los atributos de objetos son variables u objetos que almacenan valores distintos para instancias distintas de la clase (para objetos distintos).

Los atributos de clase son variables u objetos que almacenan el mismo valor para todos los objetos instanciados a partir de esa clase.

Dicho de otra forma: mientras que a partir de un atributo de objeto se crean tantas copias de ese atributo como objetos se instancien, a partir de un atributo de clase sólo se crea una copia de ese atributo que será compartido por todos los objetos que se instancien.

Si no se especifica lo contrario, los atributos son de objeto y no de clase. Para declarar un atributo de clase se utiliza la palabra reservada `static`.

La declaración mínima de los atributos es:

tipo nombreAtributo

Si existen varios atributos del mismo tipo (en la misma clase), se separan sus nombres mediante comas (,):



```
class Punto {
    int x, y;
    String nombre;
    ...
}
```

3.3.1 Atributos *static*.

Mediante la palabra reservada `static` se declaran atributos de clase.



```
class Persona {
    static int numPersonas=0; // atributo de clase
    String nombre; // atributo de objeto
    public Persona (String n) {
        nombre = n;
        numPersonas++;
    }
    public void muestra() {
        System.out.print("Soy "+nombre);
        System.out.println(" pero hay "+ (numPersonas-1) +
            " personas más.");
    }
}

class Static {
    public static void main(String argumentos[]) {
        Persona p1,p2,p3;
        // se crean tres instancias del atributo nombre
        // sólo se crea una instancia del atributo numPersonas
        p1 = new Persona("Pedro");
        p2 = new Persona("Juan");
        p3 = new Persona("Susana");
        p2.muestra();
    }
}
```

Las clases en Java.

```
        pl.muestra();  
    }  
}
```

Salida por pantalla:

```
Soy Juan pero hay 2 personas más.  
Soy Pedro pero hay 2 personas más.
```

En este caso, `numPersonas` es un atributo de clase y por lo tanto es compartido por todos los objetos que se crean a partir de la clase `Persona`. Todos los objetos de esta clase pueden acceder al mismo atributo y manipularlo. El atributo nombre es un atributo de objeto y se crean tantas instancias como objetos se declaren del tipo `Persona`. Cada variable declarada de tipo `Persona` tiene un atributo nombre y cada objeto puede manipular su propio atributo de objeto.

En el ejemplo, se crea un atributo `numPersonas` y tres atributos nombre (tantos como objetos de tipo `Persona`).

3.3.2 Atributos *final*.

La palabra reservada `final` calificando a un atributo o variable sirve para declarar constantes, no se permite la modificación de su valor. Si además es `static`, se puede acceder a dicha constante simplemente anteponiendo el nombre de la clase, sin necesidad de instanciarla creando un objeto de la misma.

El valor de un atributo final debe ser asignado en la declaración del mismo. Cualquier intento de modificar su valor generará el consiguiente error por parte del compilador.



```
class Circulo {  
    final double PI=3.14159265;  
    int radio;  
    Circulo(int r) {  
        radio=r;  
    }  
    public double area() {  
        return PI*radio*radio;  
    }  
}  
  
class Final {
```

Las clases en Java.

```
public static void main(String argumentos[]) {  
    Circulo c = new Circulo(15);  
    System.out.println(c.area());  
}  
}
```

Podría ser útil en algunos casos declarar una clase con constantes:



```
class Constantes {  
    static final double PI=3.14159265;  
    static final String NOMBREEMPRESA = "Ficticia S.A.";  
    static final int MAXP = 3456;  
    static final byte CODIGO = 1;  
}
```

Para acceder a estas constantes, no es necesario instanciar la clase `Constantes`, ya que los atributos se han declarado `static`. Simplemente hay que anteponer el nombre de la clase: `Constantes.PI`, `Constantes.CODIGO`, etc. para utilizarlas.

3.3.3 Atributos *transient*.

Los atributos de un objeto se consideran, por defecto, persistentes. Esto significa que a la hora de, por ejemplo, almacenar objetos en un fichero²⁰, los valores de dichos atributos deben también almacenarse.

Aquellos atributos que no forman parte del estado persistente del objeto porque almacenan estados transitorios o puntuales del objeto, se declaran como `transient` (podemos denominarlos transitorios).

²⁰ Hasta la versión 1.0 del JDK esta posibilidad no estaba contemplada y la máquina virtual Java simplemente ignoraba la etiqueta `transient`.

Las clases en Java.

Por ejemplo:



```
class Transient {
    int var1, var2;
    transient int numVecesModificado=0;
    void modifica(int v1, int v2) {
        var1=v1;
        var2=v2;
        numVecesModificado++;
    }
}
```

En este caso, la variable `numVecesModificado` almacena el número de veces que se modifica en el objeto los valores de los atributos `var1` y `var2`. A la hora de almacenar el objeto en un fichero para su posterior recuperación puede que no interese el número de veces que ha sido modificado. Declarando dicho atributo como `transient` este valor no se almacenará y será reinicializado al recuperarlo.

3.3.4 Atributos *volatile*.

Si una clase contiene atributos de objeto que son modificados asincrónicamente por distintos *threads*²¹ que se ejecutan concurrentemente, se pueden utilizar atributos `volatile` para indicarle a la máquina virtual Java este hecho, y así cargar el atributo desde memoria antes de utilizarlo y volver a almacenarlo en memoria después, para que cada *thread* puede “verlo” en un estado coherente. Esto nos ayudará a mantener la coherencia de las variables que puedan ser utilizadas concurrentemente.

²¹ Un thread (o hilo de ejecución) funciona de forma paralela a otros threads del mismo proceso. Los threads se ejecutan **asincrónicamente** cuando pueden ejecutarse “paralelamente”, sus instrucciones se entremezclan en el tiempo unas con otras, sin que se pueda predecir el orden de ejecución de las instrucciones de uno de los threads respecto de los otros.



```
class Volatil {  
    volatile int contador;  
    . . .  
}
```

Los atributos `volatile` son ignorados por la versión 1.0 del compilador del JDK.

No obstante, para poder comprender en su totalidad el sentido de este tipo de atributos, es necesario que esperemos a conocer qué son y como se emplean los *threads*. (☞ 6 **Threads**. Página 199).

3.3.5 Modificadores de ámbito de atributos.

Los modificadores de ámbito de atributo especifican la forma en que puede accederse a los mismos desde otras clases. Estos modificadores de ámbito son:

- `private`.
- `public`.
- `protected`.
- El ámbito por defecto.

3.3.5.1 Atributos *private*.

El modificador de ámbito `private` es el más restrictivo de todos. Todo atributo `private` es visible únicamente dentro de la clase en la que se declara. No existe ninguna forma de acceder al mismo si no es a través de algún método (no `private`) que devuelva o modifique su valor.

Una buena metodología de diseño de clases es declarar los atributos `private` siempre que sea posible, ya que esto evita que algún objeto pueda modificar su valor si no es a través de alguno de sus métodos diseñados para ello.

3.3.5.2 Atributos *public*.

El modificador de ámbito `public` es el menos restrictivo de todos. Un atributo `public` será visible en cualquier clase que desee acceder a él, simplemente anteponiendo el nombre de la clase.

Las aplicaciones bien diseñadas minimizan el uso de los atributos `public` y maximizan el uso de atributos `private`. La forma apropiada de acceder y modificar atributos de objetos es a través de métodos que accedan a los mismos, aunque en ocasiones, para acelerar el proceso de programación, se declaran de tipo `public` y se modifican sus valores desde otras clases.



```
final class Empleado {  
    public String nombre;  
    public String dirección;  
    private int sueldo;  
}
```

En este ejemplo existen dos atributos `public` (nombre y dirección) y uno `private` (sueldo). Los atributos nombre y dirección podrán ser modificados por cualquier clase, por ejemplo de la siguiente forma:

```
emple1.nombre="Pedro López";
```

Mientras que el atributo `sueldo` no puede ser modificado directamente por ninguna clase que no sea `Empleado`. En realidad, para que la clase estuviera bien diseñada, se deberían haber declarado `private` los tres atributos y declarar métodos para modificar los atributos. De estos métodos, el que modifica el atributo `sueldo` podría declararse de tipo `private` para que no pudiera ser utilizado por otra clase distinta de `Empleado`.

3.3.5.3 Atributos *protected*.

Los atributos `protected` pueden ser accedidos por las clases del mismo paquete (*package*) y por las subclases del mismo paquete, pero no pueden ser accedidas por subclases de otro paquete, aunque sí pueden ser accedidas las variables `protected` heredadas de la primera clase.

Las clases en Java.

Esto parece un poco confuso, veámoslo con un ejemplo:



```
package PProtegido;

public class Protegida {
    protected int valorProtegido;
    public Protegida(int v) {
        valorProtegido=v;
    }
}

package PProtegido;

public class Protegida2 {
    public static void main(String argumentos[]) {
        Protegida p1= new Protegida(0);
        p1.valorProtegido = 4;
        System.out.println(p1.valorProtegido);
    }
}
```

En este caso, por pertenecer la clase `Protegida2` al mismo paquete que la clase `Protegida`, se puede acceder a sus atributos `protected`. En nuestro caso, `valorProtegido`.



```
package OtroPaquete;

import PProtegido.*;

public class Protegida3 {
    public static void main(String argumentos[]) {
        Protegida p1= new Protegida(0);
        p1.valorProtegido = 4;
        System.out.println(p1.valorProtegido);
    }
}
```

En este caso, se importa el paquete `Pprotegido` para poder acceder a la clase `Protegida`, pero el paquete en el que se declara `Protegida3` es distinto al que

Las clases en Java.

contiene `Protegida`, por lo que no se puede acceder a sus atributos `protected`. El compilador Java mostraría un error al acceder a `valorProtegido` en la línea `p1.valorProtegido = 4;` (puesto que intenta acceder, para modificar, un atributo protegido) y en la línea `System.out.println(p1.valorProtegido);` (por el mismo motivo, a pesar de que se trate sólo de leer el valor).

Sin embargo:



```
package OtroPaquete;

import PProtegido.*;

public class Protegida4 extends Protegida {
    public Protegida4(int v) {
        super(v);
    }
    public void modifica(int v) {
        valorProtegido=v;
    }
}
```

En este caso, se ha declarado una subclase de `Protegida`. Esta clase puede acceder a sus atributos (incluso a `valorProtegido`), por ejemplo, a través del método `modifica`, pero:



```
package OtroPaquete;

public class EjecutaProtegida4 {
    public static void main(String argumentos[]) {
        Protegida4 p1= new Protegida4(0);
        p1.valorProtegido = 4;
        System.out.println(p1.valorProtegido);
    }
}
```

En este caso, a pesar de que la variable es del tipo `Protegida4` (subclase de `Protegida`), y la clase `EjecutaProtegida4` pertenece al mismo paquete que `Protegida4`, no se puede acceder a los atributos `private`. Sólo los métodos de la clase `Protegida4` pueden hacerlo. Así:

Las clases en Java.



```
package OtroPaquete;

public class EjecutaProtegida4_2 {

    public static void main(String argumentos[]) {

        Protegida4 p1= new Protegida4(0);

        p1.modifica(4);

    }

}
```

Esta clase sí que puede modificar el atributo `protected` pero únicamente a través del método de la clase `Ejecuta4` denominado `modifica`.

En resumen: Un atributo protegido sólo puede ser modificado por clases del mismo paquete, ahora bien, si se declara una subclase entonces esa subclase es la encargada de proporcionar los medios para acceder al atributo protegido.

3.3.5.4 *El ámbito por defecto de los atributos.*

Los atributos que no llevan ningún modificador de ámbito pueden ser accedidos desde las clases del mismo paquete, pero no desde otros paquetes.

3.3.5.5 Resumen de ámbitos de atributos.

Modificador	Acceso desde:			
	misma clase	subclases	mismo paquete	todo el mundo
private	SI	NO	NO	NO
public	SI	SI	SI	SI
protected	SI	SEGÚN	SI	NO
por defecto	SI	NO	SI	NO

3.4 Métodos

Sintaxis general de los métodos:

```
Declaración de método {  
    Cuerpo del método  
}
```

3.4.1 Declaración de método.

La declaración mínima sin modificadores de un método es:

TipoDevuelto NombreMétodo (ListaParámetros)

Donde:

- *TipoDevuelto* es el tipo de dato devuelto por el método (función). Si el método no devuelve ningún valor, en su lugar se indica la palabra reservada `void`. Por ejemplo: `void noDevuelveNada`
- *NombreMétodo* es un identificador válido en Java.
- *ListaParámetros* si tiene parámetros, es una sucesión de pares **tipo - valor** separados por comas. Ejemplo: `int mayor(int x , int y)`. Los parámetros pueden ser también objetos. **Los tipos simples de datos se pasan siempre por valor y los objetos y vectores por referencia.**

Las clases en Java.

Cuando se declara una subclase, esa subclase hereda, en principio, todos los atributos y métodos de la superclase (clase padre). Estos métodos pueden ser redefinidos en la clase hija simplemente declarando métodos con los mismos identificadores, parámetros y tipo devuelto que los de la superclase. Si desde uno de estos métodos redefinidos se desea realizar una llamada al método de la superclase, se utiliza el identificador de la superclase y se le pasan los parámetros.

Ejemplo: suponiendo que se ha declarado una clase como heredera de otra (SuperC) en la que existe el método `int mayor(int x, int y)`, se puede redefinir este método simplemente declarando un método con el mismo identificador y parámetros `int mayor(int x, int y)`. Si desde dentro del método redefinido se desea hacer referencia al método original se podría utilizar: `var = SuperC(x,y)`;

También se pueden declarar métodos para una misma clase con los mismos identificadores pero con parámetros distintos.



```
class Mayor {
// Declara dos métodos con el mismo identificador
// uno de ellos acepta dos enteros y el otro dos
// enteros largos.
// ambos métodos devuelven el valor mayor de los
// dos enteros que se pasan como parámetros.
    static int mayor(int x, int y) {
        if (x>y)
            return x;
        else
            return y;
    }
    static long mayor(long x, long y) {
        if (x>y)
            return x;
        else
            return y;
    }
}
class ConstantesMayor {
    static final int INT = 15;
    static final long LONG = 15;
}

class SubMayor extends Mayor {
// modifica la clase Mayor de la siguiente forma:
// los métodos devuelven el valor mayor de entre
// los dos parámetros que se le pasan, pero
// siempre, como mínimo devuelve el valor
```

Las clases en Java.

```
// de las constantes INT y LONG
static int mayor(int x, int y) {
    // llama al método mayor de la superclase
    int m = Mayor.mayor(x,y);
    return Mayor.mayor(m,ConstantesMayor.INT);
}
static long mayor(long x, long y) {
    // llama al método mayor de la superclase
    long m = Mayor.mayor(x,y);
    return Mayor.mayor(m,ConstantesMayor.LONG);
}
}

class EjecutaMayor {
    public static void main(String argumentos[]) {
        int int1=12,int2=14;
        long long1=20, long2=10;
        System.out.println("ENTEROS:");
        System.out.println("mayor de 12 y 14 = " +
            Mayor.mayor(int1,int2));
        System.out.println("mayor de 12 y 14 y 15 = " +
            SubMayor.mayor(int1,int2));
        System.out.println("ENTEROS LARGOS:");
        System.out.println("mayor de 20 y 10 = " +
            Mayor.mayor(long1,long2));
        System.out.println("mayor de 20 y 10 y 15 = " +
            SubMayor.mayor(long1,long2));
    }
}
```

Declaración completa de métodos. Sintaxis general:

**[ModificadorDeÁmbito] [static] [abstract] [final] [native] [synchronized]
TipoDevuelto NombreMétodo ([ListaParámetros]) [throws
ListaExcepciones]**

3.4.1.1 Métodos *static*.

Los métodos `static` son métodos de clase²² (no de objeto) y por tanto, no necesita instanciarse la clase (crear un objeto de esa clase) para poder llamar a ese

²² ↪ Ver el punto 3.3 **Declaración de atributos**, en página 84. Ahí se hace una distinción entre atributos de clase y de objeto. Esta distinción también es válida para los métodos.

Las clases en Java.

método. Se ha estado utilizando hasta ahora siempre que se declaraba una clase ejecutable, ya que para poder ejecutar el método `main()` no se declara ningún objeto de esa clase.

Los métodos de clase (`static`) únicamente pueden acceder a sus atributos de clase (`static`) y nunca a los atributos de objeto (no `static`). Ejemplo:



```
class EnteroX {
    int x;
    static int x() {
        return x;
    }
    static void setX(int nuevaX) {
        x = nuevaX;
    }
}
```

Mostraría el siguiente mensaje de error por parte del compilador:

```
MetodoStatic1.java:4: Can't make a static reference to
nonstatic variable x in class EnteroX.
```

```
        return x;
            ^
```

```
MetodoStatic1.java:7: Can't make a static reference to
nonstatic variable x in class EnteroX.
```

```
        x = nuevaX;
            ^
```

2 errors

Las clases en Java.



Sí que sería correcto:

```
class EnteroX {
    static int x;
    static int x() {
        return x;
    }
    static void setX(int nuevaX) {
        x = nuevaX;
    }
}
```

Al ser los métodos `static`, puede accederse a ellos sin tener que crear un objeto `EnteroX`:



```
class AccedeMetodoStatic {
    public static void main(String argumentos[]) {
        EnteroX.setX(4);
        System.out.println(EnteroX.x());
    }
}
```

3.4.1.2 Métodos *abstract*.

Los métodos `abstract` se declaran en las clases `abstract`²³. Es decir, si se declara algún método de tipo `abstract`, entonces, la clase debe declararse obligatoriamente como `abstract`.

Cuando se declara un método `abstract`, no se implementa el cuerpo del método, sólo su signatura. Las clases que se declaran como subclases de una clase `abstract` deben implementar los métodos `abstract`. Una clase `abstract` no puede ser instanciada, únicamente sirve para ser utilizada como superclase de otras clases.

²³  Ver el punto de 3.1.3.2 **Clases abstract** en página 81. En el mismo existe un ejemplo de declaración de métodos `abstract`.

3.4.1.3 Métodos *final*.

Los métodos de una clase que se declaran de tipo `final` no pueden ser redefinidos por las subclases. Esta opción puede adoptarse por razones de seguridad, para que nuestras clases no puedan ser extendidas por otros.



```
abstract class Animal {
    String nombre;
    int patas;
    public Animal(String n, int p) {
        nombre=n;
        patas=p;
    }
    public final int numPatas(){
        return patas;
    }
    abstract void habla();
}

class Perro extends Animal {
    String raza;
    public Perro(String n, int p, String r) {
        super(n,p);
        raza=r;
    }
    public void habla() {
        System.out.println("Me llamo "+nombre+": GUAU, GUAU");
        System.out.println("mi raza es "+raza);
        System.out.println("Tengo "+numPatas()+" patas.");
    }
}

class Gallo extends Animal {
    public Gallo(String n, int p) {
        super(n,p);
    }
    public void habla() {
        System.out.println("Soy un Gallo, Me llamo "+nombre);
        System.out.println("Kikirikiiii");
    }
}

class FinalAbstracta {
    public static void main(String argumentos[]) {
        Perro toby = new Perro("Toby",4,"San Bernardo");
        Gallo kiko = new Gallo("Kiko",2);
        kiko.habla();
    }
}
```

Las clases en Java.

```
        System.out.println();
        toby.habla();
    }
}
```

En este caso, la clase `Animal` es abstracta (no puede instanciarse), sólo puede utilizarse como superclase de otras (`Perro` y `Gallo`). Uno de los métodos de `Animal` es `final` y por lo tanto no puede redefinirse. Cualquier intento de declarar un método (`numPatas`) en cualquier subclase de `Animal` generaría un error del compilador.

3.4.1.4 *Métodos native.*

Los métodos `native`, son métodos que se encuentran escritos en otro lenguaje de programación distinto a Java, por ejemplo C. La integración de código escrito en otros lenguajes está en fase de desarrollo en la versión 1.0 del JDK, por lo tanto puede tener errores. Su estudio escapa del cometido de este libro que se centra en el lenguaje Java.

3.4.1.5 *Métodos synchronized.*

Los métodos `synchronized` son métodos especiales para cuando varios *threads* (subprocesos) pueden acceder concurrentemente a los mismos datos y se desea que una sección crítica se proteja (bloquee) para que los *threads* accedan en exclusión mutua a la misma. ☞ Se verá en el capítulo correspondiente 6 `Threads`, en la página 199.

3.4.1.6 *Modificadores de ámbito de los métodos.*

Los modificadores de ámbito de los métodos son exactamente iguales que los de los atributos²⁴, especifican la forma en que puede accederse a los mismos desde otras clases. Estos modificadores de ámbito son:

²⁴ ☞ Ver capítulo 3.3.5 `Modificadores de ámbito de atributos`. En Página 89

Las clases en Java.

- `private`.
- `public`.
- `protected`.
- **El ámbito por defecto.**

Modificador	Acceso desde:			
	misma clase	subclases	mismo paquete	todo el mundo
Private	SI	NO	NO	NO
Public	SI	SI	SI	SI
Protected	SI	SEGÚN	SI	NO
por defecto	SI	NO	SI	NO

3.4.1.7 Lista de excepciones potenciales.

Todos los métodos pueden generar excepciones (condiciones de error). Estas excepciones pueden ser “lanzadas” por el método para que sean tratadas por el método que realizó la llamada. Todas las posibles excepciones que pueda generar el método (y que no sean tratadas por el propio método) deben ser declaradas como una lista separada por comas en la declaración del mismo detrás de la declaración del identificador del método y la lista de parámetros:

TipoDevuelto NombreMétodo ([ListaParámetros]) [throws ListaExcepciones]

Las clases en Java.

Ejemplo:



```
class Nif {
    int dni;
    char letra;
    static char tabla[]={ 'T','R','W','A','G','M','Y','F','P','D',
                          'X','B','N','J','Z','S','Q','V','H','L',
                          'C','K','E' };
    public Nif(int ndni,char nletra) throws NifException{
        if (Character.toUpperCase(nletra)==tabla[ndni%23]) {
            dni=ndni;
            letra=Character.toUpperCase(nletra);
        } else throw new LetraNifException();
    }
    ...
}
```

En la declaración del método `Nif` (constructor) indicamos que puede “lanzar” la excepción `NifException`. Luego, en la cláusula `else` se procede a lanzar efectivamente la excepción cuando la letra calculada no coincide con la que hemos pasado como parámetro.

3.4.2 Cuerpo del método.

Sintaxis general de los métodos:

```
Declaración de método {  
    Cuerpo del método  
}
```

El cuerpo del método contiene la implementación del mismo y se codifica entre las llaves del método. Dentro del mismo pueden declararse variables locales al mismo²⁵.

²⁵ Ver 2.8.1 **Variables locales**. En página 40.

Las clases en Java.

El único caso en el que no se implementa el cuerpo del método es en la declaración de clases abstractas²⁶.

La forma de declarar la salida del método es mediante la palabra reservada `return`. Si el tipo de dato devuelto del método ha sido declarado `void`, entonces únicamente se sale del cuerpo del método mediante la instrucción `return`; Si en la declaración de la cabecera del método se ha declarado un tipo de dato devuelto por el mismo, entonces se devolverá el valor correspondiente al método mediante `return` seguido de una expresión cuyo resultado es del mismo tipo que el declarado.

REFERENCIA A ATRIBUTOS DE MÉTODO ANTE LA AMBIGÜEDAD:

Si existen en un método variables locales o parámetros con el mismo nombre que atributos de la clase, éstos últimos quedan ocultos dentro del cuerpo del método. Para hacer referencia a los atributos del método en lugar de las variables locales o parámetros, se les antepone la palabra reservada `this` seguida de un punto (`.`), que identifica al objeto actual.

Como puede verse en el ejemplo de la clase `Pila` (en la página siguiente):



```
void push(Object elemento) throws DesbordaPila{
    try {
        this.elemento[++cimaPila]=elemento;
    } catch (Exception ex) {
        throw new DesbordaPila();
    }
}
```

Existe un parámetro (`elemento`) que tiene el mismo nombre que un atributo de la clase, el vector `elemento[]`. Para acceder al atributo, se utiliza `this.elemento`, de otro modo se estaría accediendo al parámetro. Como puede verse, también en el ejemplo, al atributo (vector) se le asigna el valor del parámetro `elemento`.

²⁶ Ver 3.1.3.2 `abstract`. en página 81.

Las clases en Java.



```
class Pila {
    static final int PILAVACIA = -1;
    Object elemento[];
    int cimaPila = PILAVACIA;
    public Pila(int numElementos) {
        elemento = new Object[numElementos];
    }
    boolean pilaVacía() {
        if (cimaPila == PILAVACIA)
            return true;
        else
            return false;
    }
    void push(Object elemento) throws DesbordaPila{
        try {
            this.elemento[++cimaPila]=elemento;
        } catch (Exception ex) {
            throw new DesbordaPila();
        }
    }
    Object pop() {
        if (cimaPila == PILAVACIA)
            return null;
        else
            return elemento[cimaPila--];
    }
}

class DesbordaPila extends Exception {
}

class EjecutaPila {
    public static void main(String argumentos[])
        throws DesbordaPila {
        Pila p = new Pila(5);
        if (p.pilaVacía())
            System.out.println("Pila vacía");
        System.out.println("llenando pila ...");
        p.push("primer elemento");
        p.push("segundo elemento");
        p.push("tercer elemento");
        p.push("cuarto elemento");
        if (!p.pilaVacía())
            System.out.println("Pila no vacía");
        System.out.println("Vaciando pila ...");
        while (!p.pilaVacía())
            System.out.println(p.pop());
    }
}
```

Las clases en Java.

Salida del programa:

```
Pila vacía
llenando pila ...
Pila no vacía
Vaciando pila ...
cuarto elemento
tercer elemento
segundo elemento
primer elemento
```

REFERENCIA A ATRIBUTOS DE LA SUPERCLASE:

Para referenciar atributos o métodos de la superclase que han sido sobrescritos por la clase hija se utiliza el identificador del objeto padre `super`.



```
class Padre {
    String variable;
    void metodo() {
        variable = "Padre";
    }
}
class Hija extends Padre {
    String variable;
    void metodo() {
        variable = "Hija";
        super.metodo();
        System.out.println(variable);
        System.out.println(super.variable);
    }
}
```

Las clases en Java.

En primer lugar se declara una clase (`Padre`) con un atributo (`variable`) y un método (`metodo`). También se declara una subclase de `Padre` (`Hija`), que sobrescribe el atributo (`variable`) y el método (`metodo`).

El método `metodo`, lo único que hace es asignar un valor al atributo `variable` (en el caso de la clase `padre` “Padre” y en el caso de la clase `hija` “Hija”).

Cuando se ejecuta el método `metodo()` de la clase `Hija`, se asigna “Hija” al atributo (`variable`) de la clase `Hija`. Después se realiza una llamada al método `metodo()` de la superclase mediante la instrucción `super.metodo()`; Al ejecutarse este método, se asigna el valor “Padre” al atributo `variable` de la clase `Padre`.

`System.out.println(variable)` muestra el valor del atributo `variable` de la clase `Hija` **Hija**.

`System.out.println(super.variable)` muestra el valor del atributo `variable` de la clase `Padre` **Padre**

3.4.3 Constructores.

Un constructor es un método especial de las clases que sirve para inicializar los objetos que se instancian como miembros de una clase.

Para declarar un constructor basta con declarar un método con el mismo nombre que la clase. **No se declara el tipo devuelto por el constructor** (ni siquiera `void`), aunque sí que se pueden utilizar los modificadores de ámbito de los métodos: `public`, `protected`, `private`.

Los constructores tienen el mismo nombre que la clase y todas las clases tienen uno por defecto (que no es necesario declarar), aunque es posible sobrescribirlo e incluso declarar distintos constructores (sobrecarga de métodos) al igual que los demás métodos de una clase.



```
class Nif {
    int dni;
    char letra;
    static char tabla[]={ 'T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P',
```

Las clases en Java.

```
        'D','X','B','N','J','Z','S','Q','V',
        'H','L','C','K','E'};
public Nif(int ndni,char nletra) throws NifException{
    if (Character.toUpperCase(nletra)==tabla[ndni%23]) {
        dni=ndni;
        letra=Character.toUpperCase(nletra);
    }
    else
        throw new LetraNifException("Letra de NIF incorrecta");
}
public Nif(int ndni) {
    dni=ndni;
    letra=tabla[dni%23];
}
public Nif(String sNif) throws NifException,
        LetraNifException {
    char letraAux;
    StringBuffer sNumeros= new StringBuffer();
    int i,ndni;
    for (i=0;i<sNif.length();i++) {
        if ("1234567890".indexOf(sNif.charAt(i))!=-1) {
            sNumeros.append(sNif.charAt(i));
        }
    }
    try {
        dni=Integer.parseInt(sNumeros.toString());
        letraAux=Character.toUpperCase(sNif.charAt(
            sNif.length()-1));
    } catch (Exception ex) {
        throw new NifException("NIF incorrecto");
    }
    letra=tabla[dni%23];
    if ("ABCDEFGHIJKLMNPOQRSTUVWXYZ".indexOf(letraAux)!=-1) {
        if (letraAux!=letra) {
            throw new
                LetraNifException("Letra de NIF incorrecta");
        }
    } else letra=tabla[dni%23];
}
public char obtenerLetra() {
    return letra;
}
public int obtenerDni() {
    return dni;
}
public String toString() {
    return (String.valueOf(dni)+String.valueOf(letra));
}
public String toStringConFormato() {
    String sAux= String.valueOf(dni);
    StringBuffer s = new StringBuffer();
    int i;
```

Las clases en Java.

```
        for (i=sAux.length()-1;i>2;i-=3) {
            s.insert(0,sAux.substring(i-2,i+1));
            s.insert(0,".");
        }
        s.insert(0,sAux.substring(0,i+1));
        s.append('-');
        s.append(letra);
        return (s.toString());
    }
    static char letraNif(int ndni) {
        return tabla[ndni%23];
    }
    static char letraNif(String sDni) throws NifException {
        Nif j = new Nif(sDni);
        return j.obtenerLetra();
    }
}
class NifException extends Exception {
    public NifException() { super(); }
    public NifException(String s) { super(s); }
}
class LetraNifException extends NifException {
    public LetraNifException() { super(); }
    public LetraNifException(String s) { super(s); }
}
}
```

En el ejemplo anterior, la clase `Nif` tiene tres constructores:

- `public Nif(int ndni, char nletra) throws LetraNifException`
- `public Nif(int ndni)`
- `public Nif(String sNif) throws NifException, LetraNifException`

Para inicializar un objeto de una determinada clase se llama a su constructor después de la palabra reservada `new`.



```
class EjecutaNif {
    public static void main(String argumentos[]) {
        Nif n;
        int dni;
        if (argumentos.length!=1) {
            System.out.println("Uso: EjecutaNif dni");
            return;
        }
        else {
            dni = Integer.valueOf(argumentos[0]).intValue();
        }
    }
}
```

Las clases en Java.

```
        n = new Nif(dni);
        System.out.println("Nif: "+n.toStringConFormato());
    }
}
```

En este ejemplo, se está llamando al segundo constructor, aquel que acepta como parámetro un entero.

En él se acepta como argumento en la línea de comandos un DNI y muestra el NIF correspondiente:

```
java EjecutaNif 18957690
```

mostraría la siguiente salida por pantalla:

```
Nif: 18.957.690-D
```

Es bastante habitual cuando se sobreescribe el constructor o constructores de la superclase el realizar una llamada al constructor de la superclase y después realizar otras operaciones de inicialización de la clase hija. Esto, como ya se ha visto en el punto anterior, se realiza utilizando el identificador de clase `super`. Así, si se declarara una subclase de `Nif`, y se sobreescribiera alguno de sus constructores, se podría realizar en primer lugar, una llamada al constructor de `Nif`.



```
class HijaNif extends Nif {
    public static numNifs = 0;
    ...
    public Nif(int ndni) {
        super.Nif(ndni);
        numNifs++;
    }
    ...
}
```

En este caso se ha declarado una clase `HijaNif` que añade un atributo de clase²⁷ que sirve de contador del número de objetos que se instancian de la clase `HijaNif`. Se reescribe el constructor (constructores) de forma que se incrementa este contador por cada objeto declarado.

²⁷ Ver punto 3.3.1 **Atributos static**. En página 85.

3.4.4 Destructores.

Un destructor es un método de la clase que sirve para realizar una serie de operaciones cuando un objeto perteneciente a la clase deja de existir. Operaciones típicas en los objetos cuando desaparecen son la liberación de recursos del sistema que tuviera asignados el objeto: liberación de memoria que pudiera tener reservada el objeto, cierre de los ficheros y sockets que tuviera abiertos, etc..

En Java existe un *Thread* del sistema “*Garbage collector*” literalmente: Recolector de Basura, que se ejecuta regularmente para liberar la memoria asignada a objetos que ya no se necesitan²⁸. A pesar de ello, puede ser necesario realizar algunas operaciones adicionales. Para ello hay que declarar un método de la siguiente forma:

protected void finalize() throws throwable

Por ejemplo, en la clase `HijaNif`, se utiliza un contador para saber el número de objetos instanciados de la clase `HijaNif`. Para decrementar `numNifs`, habría que declarar el método `finalize()`:



```
protected void finalize() throws throwable {
    numNifs--;
    super.finalize();
}
```

Es conveniente llamar al método `super.finalize()`, el destructor de la superclase, para liberar recursos que pudiera tener asignados la clase heredados transparentemente de la clase padre y de los cuales no se tuviera conocimiento.

3.5 STRINGS.

Los *strings* (o cadenas de caracteres) en Java son objetos y no vectores de caracteres como ocurre en C.

²⁸ Se libera la memoria asignada a objetos cuando el contador de referencias al mismo es igual a cero.

Las clases en Java.

Existen dos clases para manipular strings: `String` y `StringBuffer`. `String` se utiliza cuando las cadenas de caracteres no cambian (son constantes) y `StringBuffer` cuando se quiere utilizar cadenas de caracteres dinámicas, que puedan variar en contenido o longitud.

3.5.1 La clase `String`.

Como ya se ha mencionado, se utilizarán objetos de la clase `String` para almacenar cadenas de caracteres constantes, que no varían en contenido o longitud.

Cuando el compilador encuentra una cadena de caracteres encerrada entre comillas dobles, crea automáticamente un objeto del tipo `String`.

Así la instrucción:

```
System.out.println( "Hola mundo");
```

Provoca que el compilador, al encontrar “Hola mundo”, cree un objeto de tipo `String` que inicializa con el *string* “Hola mundo” y este objeto es pasado como argumento al método `println()`.

Por la misma razón, también puede inicializarse una variable de tipo `String` de la siguiente forma:

```
String s;  
s = "Hola mundo";  
○  
String s = "Hola mundo";
```

Además, por ser `String` una clase, también puede inicializarse una variable de dicha clase mediante su constructor:

```
String s;
```

Las clases en Java.

```
s = new String("Hola mundo");  
○  
String s = new String("Hola mundo");
```

Se ha mencionado que la clase `String` sirve para almacenar cadenas de caracteres que no varían, sin embargo es posible hacer lo siguiente:

```
String s = "Hola mundo";  
s = "El mensaje es: " + s;
```

Esto no significa que la longitud de “s” pueda ser variada. Lo que ha ocurrido en realidad es que el compilador ha creado un objeto de la clase `String` al encontrar “El mensaje es: “, después ha realizado la operación de concatenación de strings entre ese objeto creado y `s`; y el resultado a sido asignado a un nuevo objeto creado por el compilador; este objeto es asignado a la variable `s` (el puntero de `s` apunta al nuevo objeto) y el objeto al que apuntaba anteriormente `s` ya no tiene ninguna referencia, por lo que el “recolector de basura” liberará la memoria utilizada por el mismo al detectar su inaccesibilidad.

El operador `+` está sobrecargado y puede concatenar `Strings` con enteros, `StringBuffers`, etc.

La siguiente expresión es correcta:

```
String s = "Quiero " + 2 + "cafés";
```

Puede concatenarse un `String` con cualquier objeto. El resultado es el `String` original concatenado al `String` que devuelve el método `toString()` que toda clase hereda de la clase raíz `Object` y que puede estar, o no, redefinido.

Pruébese a pasar como parámetro un objeto de cualquier clase al método `System.out.println()`. Por ejemplo:

```
class Clase {  
}  
  
class MuySimple {  
    public static void main(String arg[]) {  
        Clase c = new Clase();  
        System.out.println(c);  
    }  
}
```

Produce la salida:

```
Clase@1cc731
```

(Resultado de llamar al método `toString()` de la clase `Clase`.)

3.5.1.1 Constructores de la clase `String`.

- **`public String();`**

Construye un *string* vacío.

- **`public String(byte bytes[]);`**

Construye un *string* a partir de un vector de bytes codificados dependiendo de la plataforma (por ejemplo, ASCII), por lo tanto, la longitud del vector no tiene porqué coincidir siempre con la del *string*.

Ejemplo:

```
byte bytes[]={65,66};  
String s = new String(bytes);    "AB"
```

- **`public String(byte bytes[], int posición, int cantidad);`**

Construye un *string* a partir de un vector de bytes codificados dependiendo de la plataforma. Para ello toma el primer byte que se encuentra en la posición indicada como parámetro y tantos bytes del vector como se indica en el parámetro "*cantidad*".

Ejemplo:

```
byte bytes[]={65,66,67,68,69};  
String s = new String(bytes, 1 , 3);    BCD
```

- **`public String(char valor[])`**

Construye un *string* inicializado por un vector de caracteres (*valor*).

Ejemplo:

```
char c[] = {'H','o','l','a'};  
String s = new String( c );
```

- **`public String(char valor[], int posición, int cantidad);`**

Construye un *string* inicializado por un subvector de caracteres. El primer carácter del *string* será el indicado por el primer entero (*posición*),

Las clases en Java.

teniendo en cuenta que los vectores comienzan por el índice cero. La longitud del *string* será especificado por el segundo entero (**cantidad**).

Ejemplo:

```
char c[] = {'H','o','l','a'};
String s = new String( c , 1 , 2 );
```

 “ol”

- **public String(String valor);**

Construye un `String` que es copia del `String` especificado como parámetro.

Ejemplo:

```
String s = new String("Hola mundo");
```

- **public String(StringBuffer buffer);**

Construye un nuevo `String` a partir de un `StringBuffer`.

3.5.1.2 Métodos de la clase `String`.

Existen multitud de métodos para la manipulación del contenido de un objeto de la clase `String`. ↪ Véase el punto C Las clases `String` y `StringBuffer`. en la página 283.

3.5.2 La clase `StringBuffer`.

Si se quiere utilizar un *string* que pueda variar de tamaño o contenido, se deberá crear un objeto de la clase `StringBuffer`.

3.5.2.1 Constructores de la clase `StringBuffer`.

- **public StringBuffer();**

Crea un `StringBuffer` vacío.

- **public StringBuffer(int longitud);**

Crea un `StringBuffer` vacío de la longitud especificada por el parámetro.

- **public StringBuffer(String str);**

Las clases en Java.

Crea un `StringBuffer` con el valor inicial especificado por el `String`.

3.5.2.2 Métodos de la clase `StringBuffer`.

De forma análoga a lo que ocurre con la clase `String`, la clase `StringBuffer` proporciona una gran cantidad de métodos que se encuentran enumerados en  el punto C Las clases `String` y `StringBuffer`. en la página 283.

3.6 La línea de comandos.

Al igual que ocurre en C, a un programa escrito en Java también se le pueden pasar parámetros en la línea de comandos. Estos parámetros se especifican en el método `main()` del programa, donde siempre hay que declarar un vector de `Strings`.

```
class MiClase {
    ...
    public static void main( String argumentos[] ) {
        ...
    }
}
```

Recordemos que en Java, los vectores son objetos, con todas las ventajas que ello implica. Así, se puede acceder en tiempo de ejecución al número de elementos del vector (número de argumentos) mediante el atributo público `length`. Si se intenta acceder a un elemento del vector más allá del límite, se genera una excepción.



```
class Comandos {
    public static void main (String argumentos[] ) {
        int i;
        for (i=0 ; i<argumentos.length ; i++)
            System.out.println("Parámetro "+i+": "+
                                argumentos[i]);
    }
}
```

Las clases en Java.

Si se ejecuta el programa de la siguiente forma:

```
java Comandos primero segundo tercero
```

Se produce la siguiente salida por pantalla:

```
Parámetro 0: primero  
Parámetro 1: segundo  
Parámetro 2: tercero
```

Todos los parámetros son considerados `Strings`, por lo que si se desea pasar un parámetro de otro tipo, hay que hacer la conversión adecuada.

Los espacios en blanco son considerados como separadores de parámetros. Si se desea pasar como parámetro una cadena de caracteres que incluya espacios en blanco, habrá que hacerlo encerrándola entre comillas dobles.

Si se ejecuta el programa de la siguiente forma:

```
java Comandos "Esto es un parámetro" "esto es otro"
```

Se produce la siguiente salida por pantalla:

```
Parámetro 0: Esto es un parámetro  
Parámetro 1: esto es otro
```

3.7 Interfaces.

En Java no está permitida la herencia múltiple (más de una superclase). Una aproximación a este concepto es la utilización de una sola superclase y una o varias interfaces.

Una interface es un conjunto de constantes y métodos abstractos. Cuando una clase declara una lista de interfaces mediante la cláusula

`implements`²⁹ hereda todas las constantes definidas en el interface y se compromete a redefinir todos los métodos del interface ya que estos métodos son abstractos (no tienen implementación o cuerpo).

Mediante el uso de interfaces se consigue que distintas clases implementen métodos con el mismo nombre (todas aquellas que así lo manifiesten mediante la cláusula `implements` de su declaración). De esta forma se comprometen a implementarlas. Puede decirse, que los interfaces permiten al programador, definir un conjunto de funcionalidades sin tener “ni idea” de cómo serán implementadas estas funcionalidades.

Por ejemplo, el interface `java.lang Runnable` especifica que cualquier clase que implemente este interface deberá redefinir el método `run()`. Así, la máquina virtual Java puede realizar llamadas a este método (sabe que existe) aunque no sepa qué hace realmente.

Un interface se declara de forma similar a las clases.

Sintaxis:

```
Declaración de interface {  
    Cuerpo de interface  
}
```

3.7.1 Declaración de interface.

La sintaxis general es la siguiente:

```
[public] interface NombreInterface [ extends ListadeSuperInterfaces]
```

Utilizando el modificador de interface `public` se consigue que la interface sea accesible por cualquier clase en cualquier paquete³⁰, de lo contrario la interface sólo será accesible por las clases definidas en el mismo paquete que la interface.

²⁹ ↪ Véase el punto 3.1.2 **Lista de interfaces**, en la página 79 para repasar cómo declarar los interfaces que implementa una clase.

Las clases en Java.

A diferencia de las clases, que únicamente pueden heredar de una superclase, las interfaces pueden heredar de varias interfaces mediante la cláusula `extends`.

La lista de superinterfaces de las que hereda se enumera mediante los nombres de las mismas separados por comas y precedidos por la palabra reservada `extends`.

Por ejemplo:

```
public interface Operable extends
    OperableAritmético, OperableLógico
```

En este caso, la interface `Operable` heredaría todas las constantes y métodos de las interfaces `OperableAritmético` y `OperableLógico`, pudiendo redefinirse en la interface `Operable` si fuera necesario.

3.7.2 Cuerpo del interface.

En el cuerpo del interface se declaran los métodos y las constantes del interface.

Todos los métodos declarados en el cuerpo del interface son públicos y abstractos (`public` y `abstract`) sin que sea necesario declararlos de este modo explícitamente.

Todos los atributos declarados en el cuerpo del interface, si existen, son `final`, `public` y `static` implícitamente. Es decir, son constantes.



```
public interface OperableAritmetico {
    double PI=3.14;
    double suma();
    double resta();
}
```

³⁰  El punto 3.8 **Los paquetes**, se encuentra en la página 120.

Las clases en Java.

En este caso, cualquier clase que implemente la interface `OperableAritmetico`, se compromete a redefinir los métodos `suma()` y `resta()`; y hereda la constante `PI`.



```
class DosCirculos implements OperableAritmetico {
    private int radiol,radio2;
    public static void main (String args[]) {
        DosCirculos d = new DosCirculos(10,20);
        d.saluda();
        System.out.println("Suma de áreas: "+d.suma());
        System.out.println("Resta de áreas: "+d.resta());
    }
    public DosCirculos(int r1,int r2) {
        radiol=r1;
        radio2=r2;
    }
    public double suma() {
        return PI*(radiol*radiol+radio2*radio2);
    }
    public double resta() {
        return PI*(radiol*radiol-radio2*radio2);
    }
    public void saluda() {
        System.out.println("Hola");
    }
}
```

En este ejemplo, los métodos `suma()` y `resta()` deben declararse `public` ya que todos los métodos pertenecientes a una interface son `public`.

La clase que implementa una o varias interfaces debe redefinir todos los métodos de las interfaces. En el caso de que no redefina todos los métodos de las interfaces, deberá servir únicamente como superclase de otra u otras clases que los redefinan en su totalidad. Es decir, una clase podría redefinir sólo parte de los métodos declarados en las interfaces que implementa y una clase descendiente de ésta, el resto de los métodos.

Pueden declararse variables cuyo tipo sea un interface y asignarle a las mismas objetos pertenecientes a cualquier clase que implemente esa interface. En el ejemplo anterior, se podría sustituir

Las clases en Java.

```
DosCirculos d = new DosCirculos(10,20);
```

Por

```
OperableAritmetico d = new DosCirculos(10,20);
```

Pero en este caso, sólo podrían utilizarse los métodos pertenecientes a la interface `suma()` y `resta()` y no ningún otro método que pudiera declarar la clase `DosCirculos` como `saluda()`. En este caso, el compilador mostraría el error:

```
DosCirculos.java:7: Method saluda() not found in
interface OperableAritmetico.
```

```
    d.saluda();
```

```
        ^
```

```
1 error
```

3.8 Los paquetes.

Un paquete (*package*) es una librería de clases que se agrupan para evitar problemas de nomenclatura o identificadores repetidos y por razones de organización.

Los paquetes están organizados jerárquicamente y agrupan clases e interfaces que tienen algo que ver conceptualmente unas con otras.

3.8.1 Declaración de paquetes.

Para declarar un paquete al cual pertenecerán las clases e interfaces declaradas, la primera línea del fichero fuente debe ser esa declaración de paquete:

Las clases en Java.

Sintaxis:

```
package NombrePaquete;  
    interface Interface1 {  
        .  
    }  
    class Clase1 {  
        .  
    }  
    class Clase2 {  
        .  
    }
```

NombrePaquete: puede estar compuesto por varios identificadores separados por puntos. Por ejemplo:

victor.graf

victor.mate

Únicamente las clases e interfaces declaradas como `public` podrán ser accedidas desde otro paquete.

Pueden existir varios ficheros fuente (.java) con la misma declaración de paquete, de forma que todas las clases e interfaces declaradas en ficheros con el mismo nombre de paquete se incluirán en la misma librería (paquete).

Las clases e interfaces declaradas como miembros del paquete deberán estar en un directorio con el mismo nombre que el del paquete para poder ser utilizadas por otros paquetes, teniendo en cuenta que cada uno de los distintos identificadores separados por puntos representa un directorio diferente. Así, las clases declaradas en el paquete **victor.graf** deberán estar almacenadas en el directorio **victor\graf**. Además, para hacer uso de estas clases, el directorio en el cual está el subdirectorio **victor** tendrá que estar incluido en la variable CLASSPATH.

Las clases en Java.

Ejemplo:

(en autoexec.bat)

```
SET CLASSPATH = c:\www\java\java\lib; c:\programas\clases; .
```

Las clases de los paquetes `victor.graf` y `victor.mate` podrían estar en los directorios:

```
c:\programas\clases\victor\graf
c:\programas\clases\victor\mate
```

3.8.2 Cómo hacer uso de los paquetes existentes.

Para hacer uso de alguna clase perteneciente a un paquete existen dos fórmulas:

1) Importando la clase mediante la palabra reservada `import`:

Sintaxis:

```
import NombrePaquete.NombreClase;
class Clase1 {
    .
}
    .
```

Donde *NombreClase* es el nombre de la clase que va a utilizarse o un asterisco (*) para importar todas las clases pertenecientes al paquete.

Después de importar un paquete, puede accederse a las clases de tipo `public` y a sus métodos.

2) Anteponiendo el nombre del paquete a la clase:

Ejemplo:

```
objeto = new victor.graf.Clase1();
...
```

En cualquier caso, siempre que haya ambigüedad porque haya clases con el mismo nombre en distintos paquetes que se importen, habrá que distinguir las utilizando el nombre completo (paquete + clase).

3.8.3 Paquetes pertenecientes al Java.

En el JDK se suministran varios paquetes. Van a distinguirse los que acompañan a las versiones del JDK 1.0 de las del 1.1:

3.8.3.1 JDK 1.0

`java.lang`: Paquete que contiene las clases principales del lenguaje Java. Es el único paquete que se importa (`import`) automáticamente sin tener que hacerlo explícitamente.

`java.io`: Contiene clases para manejar *streams* de entrada y salida para leer y escribir datos en ficheros, *sockets*, etc..

`java.util`: Contiene varias clases para diversas utilidades, tales como: generación de número aleatorios, manipulación de Strings, fechas, propiedades del sistema, etc..

`java.net`: Paquete que contiene clases para la gestión de redes, *sockets*, IP, URLs, etc..

`java.awt`: (AWT = *Abstract Window Toolkit*) proporciona clases para el manejo de GUIs (*Graphic User Interface*) como ventanas, botones, menús, listas, texto, etiquetas, etc.

`java.awt.image`: Paquete perteneciente al paquete `awt` que proporciona clases para la manipulación de imágenes.

`java.awt.peer`: Es un paquete que conecta el AWT de Java con implementaciones específicas del sistema en el que se ejecuta. Estas clases son utilizadas por Java y el programador normal no debería tener que utilizarlas.

`java.applet`: Contiene clases para la creación de applets.

`sun.tools.debug`: Contiene clases para la depuración de programas.

Las clases en Java.

3.8.3.2 JDK 1.1

`java.lang`: Paquete que contiene las clases principales del lenguaje Java. Es el único paquete que se importa (`import`) automáticamente sin tener que hacerlo explícitamente.

`java.lang.reflect`: Proporciona una pequeña API que soporta la inspección de clases y objetos en la máquina virtual Java

`java.math`: Proporciona dos nuevas clases: `BigDecimal` y `BigInteger`.

`java.io`: Contiene clases para manejar *streams* de entrada y salida para leer y escribir datos en ficheros, *sockets*, etc..

`java.rmi`: Proporciona acceso a invocación de métodos remotos.

`java.rmi.dgc`:

`java.rmi.registry`:

`java.rmi.server`:

`java.security`: Proporciona algoritmos de encriptación y seguridad.

`java.security.acl`:

`java.security.interfaces`:

`java.sql`: Proporciona acceso a bases de datos mediante SQL (JDBC).

`java.text`: Proporciona internacionalización de textos.

`java.util`: Contiene varias clases diversas de utilidades, tales como: generación de número aleatorios, manipulación de Strings, fechas, propiedades del sistema, etc..

`java.util.zip`: Utilidades de compresión.

`java.net`: Paquete que contiene clases para la gestión de redes, *sockets*, IP, URLs, etc..

Las clases en Java.

`java.awt`: (AWT = *Abstract Window Toolkit*) proporciona clases para el manejo de GUIs (*Graphic User Interface*) como ventanas, botones, menús, listas, texto, etiquetas, etc.

`java.awt.datatransfer`: Proporciona clases para el tratamiento de transferencia de datos en el AWT, como por ejemplo, el portapapeles (*clipboard*).

`java.awt.image`: Paquete perteneciente al paquete `awt` que proporciona clases para la manipulación de imágenes.

`java.awt.peer`: Es un paquete que conecta el AWT de Java con implementaciones específicas del sistema en el que se ejecuta. Estas clases son utilizadas por Java y el programador normal no debería tener que utilizarlas.

`java.applet`: Contiene clases para la creación de applets.

`java.beans`: *beans* en Java.

`sun.tools.debug`: Contiene clases para la depuración de programas.

`sunw.io`: Paquete interno del JDK (no pertenece al paquete `java`)

`sunw.util`: Paquete interno del JDK (no pertenece al paquete `java`)

3.9 Ejemplo completo.

Seguidamente se presenta un ejemplo completamente desarrollado que emplea los conceptos estudiados hasta ahora y que supone la aplicación de muchos de ellos. Como siempre, un ejemplo vale más que mil palabras.

El ejemplo va a consistir en realizar un programa que rellene 8 bloques de la primitiva con números distintos. Este es un método bastante habitual. Consiste en utilizar un boleto auxiliar e ir tachando los números utilizados en los 8 bloques, de forma que se utilizarán $8 \times 6 = 48$ números (del 1 al 49) y se quedará un número sin utilizar.

Por ejemplo:

Las clases en Java.

Bloque 1: 35 31 36 15 26 23
Bloque 2: 37 19 44 9 38 1
Bloque 3: 5 29 10 46 48 2
Bloque 4: 17 20 27 33 47 32
Bloque 5: 25 4 18 21 11 45
Bloque 6: 24 7 8 3 39 49
Bloque 7: 43 41 12 34 28 13
Bloque 8: 42 30 40 22 14 6

Quedaría por utilizar el 16

En un primer momento podría pensarse que puede resolverse utilizando un vector auxiliar (a modo de boleto auxiliar para ir tachando) e ir introduciendo en él los números generados aleatoriamente (en el rango 1 ... 49). Por cada número generado aleatoriamente habría que comprobar que no está ya anotado (buscando en el vector auxiliar) antes de incluirlo en el resultado. A medida que se vaya avanzando en la generación de números aleatorios se irán produciendo más y más repeticiones y el proceso irá relentizándose. Este método no es bueno ya que no puede predecirse el tiempo que tardará en terminar.

Existe una solución más eficiente, que podemos denominar de “barajado” (en efecto funciona igual que el barajado de las cartas en los juegos de mesa) consistente en utilizar un vector de 49 elementos a modo de bombo. Se inicializa rellenando los componentes con su número de índice +1³¹:

0	1	2	3	4	5	6	...	43	44	45	46	47	48
1	2	3	4	5	6	7	...	44	45	46	47	48	49

En primer lugar se extraerá un número aleatorio comprendido entre 0 y 48 (que determinará el índice o posición de la bola a extraer). Supongamos que sale el 4. Se extrae el componente que se encuentra en esa posición el **5**, que se apuntará en el resultado. A continuación se extrae el contenido de la última posición (48) y se inserta en la posición 4:

0	1	2	3	4	5	6	...	43	44	45	46	47	48
1	2	3	4	49	6	7	...	44	45	46	47	48	49

De esta forma, no se repetirá el número 5, ya que no existe en el vector.

³¹ En Java, los vectores tienen como primer índice el cero y como último índice el número de elementos -1.

Las clases en Java.

A continuación se generará otro número aleatorio, esta vez comprendido entre 0 y 47. Supongamos que sale el 2. Se extrae la bola (3) y se sustituye el contenido de la posición 2 por el de la posición 47:

0	1	2	3	4	5	6	...	43	44	45	46	47	48
1	2	48	4	49	6	7	...	44	45	46	47	48	49

A continuación se vuelve a generar otro número aleatorio comprendido entre 0 y 46. Si sale el valor máximo no ocurre nada especial: Se extrae la bola en la posición 46 el 47. Se sustituye el contenido de la posición 46 por el de la posición máxima 46. Se vuelve a introducir el valor 47 en la posición 46, por lo que el vector no se modifica. Esto no significa ningún problema, ya que la próxima vez se generará un número aleatorio comprendido entre 0 y 45 y el 47 no podrá volver a aparecer en el resultado.

La situación final (cuando se extrae el último elemento) es la siguiente:

0	1	2	3	4	5	6	...	43	44	45	46	47	48
6	16	XX	XX	XX	XX	XX	...	XX	XX	XX	XX	XX	XX

Se extrae un número aleatorio comprendido entre 0 y 1. Supongamos que sale el 0. Se anota el 6 en el resultado. Se sustituye el contenido de la posición 0 por el de la posición máxima:

0	1	2	3	4	5	6	...	43	44	45	46	47	48
16	16	XX	XX	XX	XX	XX	...	XX	XX	XX	XX	XX	XX

Al final del proceso, en la primera componente del vector queda el número que no ha sido utilizado en los bloques rellenos.

La forma de anotar los resultados consistirá simplemente en ir rellendo una matriz de enteros de dos dimensiones (6 x 8).

3.9.1 Generar números aleatorios.

Para generar números aleatorios existe una clase en el paquete `java.util`. Esta clase es `Random`. Accediendo a su documentación puede comprobarse que no existe ningún método que nos permita obtener un número entero aleatorio

Las clases en Java.

comprendido entre 0 y 49, pero sí que puede calcularse a partir de un número entero aleatorio en el rango de los enteros (`int`).

El método `public int nextInt()` Devuelve un número entero aleatorio uniformemente distribuido. Puede calcularse un número aleatorio comprendido entre dos límites de la siguiente forma:

```
nuevo = inferior+(valorAbsoluto(i) % (superior-inferior+1))
```

También es necesario, como puede verse en la expresión anterior, un método que calcule el valor absoluto de un número. La clase `java.lang.Math` tiene el método `public static int abs(int n)` que lo calcula.

Este cálculo podría realizarse cada vez que fuera necesario o podría hacerse algo más interesante: crear una subclase de `Random` y añadirle un método que devuelva un entero entre dos límites. Va a hacerse esto último:



```
package java.util;
public class Aleatorio extends Random {
    public int nextInt(int inferior, int superior) {
        int i;
        i=nextInt();
        i=inferior+(Math.abs(i) % (superior-inferior+1));
        return(i);
    }
}
```

En este caso se está creando una nueva clase `Aleatorio` (subclase de `Random`) y se está añadiendo al paquete `java.util` (la primera línea del fichero). Esta clase `Aleatorio.class` deberá estar en un subdirectorio que se llame `java\util` (ya sea el del propio JDK como en uno creado a partir de un directorio perteneciente al CLASSPATH). Si el directorio actual (`.`) está incluido en el CLASSPATH, puede crearse en el directorio actual el directorio `java` y dentro de él el directorio `util`. En éste se copiará el fichero `Aleatorio.java` y `Aleatorio.class`.³²

³² Podría pensarse que es suficiente con mover el fichero `Aleatorio.class` del directorio actual al directorio `XXXX\java\util`. Es cierto pero, además, hay que hacer desaparecer el fichero `Aleatorio.java` del directorio actual en el que se van a crear el resto de clases. Si en otras clases se hace referencia a la clase `Aleatorio` y se detecta `Aleatorio.java` en el directorio actual, se intentará compilar, dando un mensaje de error porque se creará `Aleatorio.class` en el directorio actual y no en `XXXX\java\util`.

Las clases en Java.

Una vez creada la clase Aleatorio y añadida ésta al paquete `java.util`, podrá utilizarse en lugar de la clase `Random` desde cualquier aplicación que importe el paquete `java.util`. (Si el directorio `XXXX` en el que se encuentra `XXXX\java\util\Aleatorio.class` está en la variable de entorno `CLASSPATH`).

En la clase Aleatorio se ha creado un método: `public int nextInt(int inferior, int superior)` con el mismo nombre que otro existente, y heredado, de la clase `Random`. En él se llama al método `nextInt()` que devuelve un número aleatorio comprendido en el rango - 2.147.483.648 ... 2.147.483.647. También se llama al método `java.lang.Math.abs()`, que por ser estático no necesita instanciarse ningún objeto de la clase `Math`.

3.9.2 Creación de un bombo.

Además de este generador de números aleatorios será necesario un bombo de donde extraer la bolas:



```
import java.util.Aleatorio;
public class Bombo {
    // vector para almacenar las bolas
    private int bolas[];
    // número de bolas (por defecto) en el bombo
    private int numBolas=49;
    // Objeto para generar número aleatorios
    private Aleatorio alea;
    // Constructor por defecto
    public Bombo() {
        inicializa();
    }
    // Constructor que permite indicar un número
    // de bolas (distinto a 49)
    public Bombo(int nBolas) {
        numBolas=nBolas;
        inicializa();
    }
    // Método que devuelve el nº de bolas que quedan
    // en el bombo
    public int numBolas() {
        return(numBolas);
    }
    // Rellena el bombo e inicializa el
    // generador de números aleatorios
    private void inicializa() {
```

Las clases en Java.

```
        int i;
        bolas = new int[numBolas];
        for (i=1; i<=numBolas; i++) {
            bolas[i-1]=i;
        }
        alea = new Aleatorio();
    }
    // Extrae y devuelve una bola del bombo
    public int sacar() {
        int devuelve,i;
        i=alea.nextInt(0,numBolas-1);
        devuelve=bolas[i];
        bolas[i]=bolas[numBolas-1];
        numBolas--;
        return(devuelve);
    }
}
```

En este caso se han declarado dos constructores únicamente por cuestiones didácticas, ya que únicamente se necesita el constructor `Bombo()` para 49 bolas. Los constructores son los sitios adecuados para inicializar valores (también para realizar las operaciones que deban ejecutarse una sola vez antes de usar el objeto). En este caso, las operaciones de inicialización se han extraído a un método aparte `inicializa()` porque se deben ejecutar acciones comunes desde los dos constructores y de esta forma no se duplica código.

En esta clase se han declarado tres atributos. Una regla que suele funcionar para saber si debe utilizarse un atributo o una variable local en alguno de los métodos suele ser la siguiente:

Si un valor debe accederse desde más de un método o debe conservar su valor entre distintas llamadas al mismo método, debe ser un atributo.

Según la regla anterior, en este caso no es necesario que el generador de números aleatorios sea un atributo. En realidad podría ser perfectamente un objeto local al método `sacar()`, pero en ese caso debería llamarse a su constructor (inicializar el generador de números aleatorios) 48 veces (tantas como bolas se extraigan). Se ha optado por declararlo como atributo e inicializarlo llamando a su constructor una sola vez en el método `inicializa()` aunque, como ya se ha comentado, podría haberse declarado como variable local dentro del método `sacar()` de la forma: `Aleatorio alea = new Aleatorio();`

Las clases en Java.

El atributo **numBolas** indica en cada momento el número de bolas que quedan en el bombo. Cada vez que se llama al método `sacar()`, que devuelve una bola del bombo, se decrementa este atributo.

Se ha declarado también un método con el mismo identificador que el atributo anterior y que devuelve precisamente el valor del atributo **numBolas**. En este caso no es necesario, no se va a hacer ninguna llamada a éste método, pero si se piensa que la clase Bombo puede ser útil en otros programas y se va a reutilizar, es bastante lógico pensar que será necesario un método para averiguar la cantidad de bolas que quedan en el bombo.

Puede comprobarse que todos los atributos han sido declarados como `private`. Por esta razón no puede accederse directamente a los mismo, si no es a través de algún método.

Siempre que sea posible, es conveniente declarar los atributos de tipo `private` para evitar que puedan ser accedidos directamente desde otras clases y, quizás, de forma incorrecta.

Accediendo a ellos únicamente desde métodos de la propia clase se consigue evitar un uso inadecuado de los mismos, sólo se accede a ellos de la forma prevista.

3.9.3 La clase *Resultado*.

Ya se dispone del generador de números aleatorios y del bombo. Aparte de estos dos elementos es necesario un objeto donde almacenar los resultados. Puede abordarse la generación de resultados de muy diversas formas. En nuestro caso se va a crear un objeto **Resultado** un tanto independiente, ya que se encargará él mismo de rellenar sus valores (mediante su método `rellena()`) y de mostrarse en pantalla (mediante su método `muestra()`):



```
class Resultado {
// Clase que almacena el resultado
    private int r[][];
    protected Bombo bombo;
    public Resultado() {
        r = new int[6][8];
        bombo = new Bombo();
    }
}
```

Las clases en Java.

```
}  
public void rellena() {  
    // Rellena la matriz de resultados  
    int i,j;  
    for (i=0;i<6;i++) {  
        for (j=0;j<8;j++) {  
            r[i][j]= bombo.sacar();  
        }  
    }  
}  
// Muestra el resultado  
public void muestra() {  
    int i,j;  
    for (i=0;i<8;i++) {  
        for (j=0;j<6;j++) {  
            System.out.print(r[j][i]+" ");  
        }  
        System.out.println("\n");  
    }  
}  
}
```

3.9.4 El programa principal.

Con estos tres elementos únicamente falta el programa que los integre y que tenga un método `main()` para iniciar su ejecución:



```
class Primitiva {  
    public static void main (String arg[]) {  
        Resultado r = new Resultado();  
        r.rellena();  
        r.muestra();  
    }  
}
```

Las clases en Java.

Resultado de la ejecución:

```
14 2 25 24 26 30
21 15 6 37 19 16
43 8 38 3 7 45
41 18 22 5 36 49
42 12 23 10 34 29
17 13 20 35 44 48
31 47 1 32 9 28
33 11 27 39 40 4
```

¿ Por qué se ha creado un objeto de la clase `Bombo` como atributo de la clase `Resultado` y no como objeto local en el método `rellena()` de `Resultado`? Al fin y al cabo únicamente se utiliza el `Bombo` para rellenar el resultado. Es cierto. Si no se hubiera previsto una posible evolución y revisión del programa, ésta habría sido la forma lógica de operar.

3.9.5 Modificando la clase `Resultado`.

Imagínese ahora que, además de mostrar el resultado de los números utilizados se deseara mostrar también el único número de los 49 que no ha sido utilizado.

En el caso de haber utilizado un objeto `Bombo` local al método `rellena()`, después de ejecutar el método se pierde el contenido del objeto y, por tanto, no se puede acceder a la bola que queda por salir. Como hemos tenido la precaución de declararlo como atributo, puede modificarse la clase `Resultado` para mostrar además el número que falta por salir. Podría modificarse directamente la clase `Resultado`, pero vamos a hacerlo de una forma más elegante³³, creando una nueva clase `Resultado2` con `Resultado` como superclase:



```
class Resultado2 extends Resultado {
// Clase que redefinine el Resultado de forma
```

³³ No sólo es por cuestión de elegancia. Por ejemplo, es posible que un determinado programa haga uso de la clase `Resultado` y tenga el espacio de visualización del resultado muy ajustado. La modificación para mostrar también el número que falta por salir podría ser inconveniente. Puede crearse una nueva clase que mejore la anterior pero sin destruirla.

Las clases en Java.

```
// que muestre la bola que queda por sacar
public void muestra() {
    super.muestra();
    System.out.println("falta el "+bombo.sacar());
}
}
```

Se ha redefinido el método `muestra()`. Se llama al método de la superclase `super.muestra()` y, después se extrae la bola que queda en el bombo para ser mostrada `bombo.sacar()`. Puede accederse al atributo `bombo` por dos razones: porque es un atributo y porque éste es `protected`.

La clase principal con el método `main()` quedaría así:



```
class Primitiva2 {
    public static void main (String arg[]) {
        Resultado2 r = new Resultado2();
        r.rellena();
        r.muestra();
    }
}
```

Resultado³⁴:

```
6 10 36 2 7 35
45 14 48 40 22 3
44 31 49 47 23 33
15 4 17 13 9 1
42 5 20 11 26 46
41 34 16 30 39 25
43 37 38 27 24 12
18 28 8 19 21 29
falta el 32
```

³⁴ Los autores aceptarán gustosamente las contribuciones de cualquier lector que se haya convertido en supermillonario con la ayuda de este programa.

Las clases en Java.

Como curiosidad, puede decirse que se podría haber declarado `r` como `Resultado` en lugar de `Resultado2` y habría funcionado igual (↪ Véase el apéndice **B.3** Conversión por ampliación de tipos de datos referenciales. Página 277):

```
Resultado r = new Resultado2();
```

3.10 CUESTIONES

1. ¿Cómo declaramos la clase `hijo` para que descienda de la clase `padre`? En ese caso, qué métodos de la clase `padre` estarán disponibles para la clase `hijo`.
2. Las clases abstractas no pueden ser instanciadas. Entonces, cuál es su finalidad.
3. Una clase `final` no puede ser extendida, es decir, no podemos crear clases que desciendan de ella. En ese caso, qué interés tiene declarar una clase como `final`.
4. ¿Qué efecto tiene declarar un atributo de una clase como `static`? ¿Cómo podrías implementar un contador de instancias de la clase `coche` que te indicara cuántos objetos se han instanciado de esa clase?
5. ¿Por quienes pueden ser accedidos los atributos `protected`? ¿Hay alguna manera de poder acceder a esos atributos más allá de ese ámbito?
6. Para invocar a un método declarado como `static`. ¿Es necesario haber instanciado un objeto de esa clase?
7. Cómo podemos llamar a un método de una clase desde una clase derivada cuando lo hemos redefinido, es decir, cuando hemos escrito un nuevo método con el mismo nombre en la clase descendiente.
8. Cuando tenemos una variable local en un método de una clase y deseamos acceder un atributo con el mismo nombre que esa variable local. ¿Qué podemos hacer?
9. ¿Qué es un constructor y para qué sirve? (No nos referimos a las personas que construyen viviendas).
10. Las cadenas de caracteres (*strings*) en Java no son vectores de bytes, sino una clase específica. ¿Qué ventajas aporta este esquema?

4. TRATAMIENTO DE EXCEPCIONES.

Cuando un programa Java viola las restricciones semánticas del lenguaje (se produce un error), la máquina virtual Java comunica este hecho al programa mediante una **excepción**.

Muchas clases de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado hasta un disquete protegido contra escritura, un intento de dividir por cero o intentar acceder a un vector fuera de sus límites. Cuando esto ocurre, la máquina virtual Java crea un objeto de la clase `exception` o `error` y se notifica el hecho al sistema de ejecución. Se dice que se ha lanzado una excepción (“*Throwing Exception*”).

Un método se dice que es capaz de tratar una excepción (“*Catch Exception*”) si ha previsto el error que se ha producido y prevé también las operaciones a realizar para “recuperar” el programa de ese estado de error.

En el momento en que es lanzada una excepción, la máquina virtual Java recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual Java muestra un mensaje de error y el programa termina.

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción `throw`, lo que facilita la devolución de un “código de error” al método que invocó el método que causó el error.

Como primer encuentro con las excepciones, pruebe a ejecutar el siguiente programa:

Tratamiento de excepciones.



```
class Excepcion {
    public static void main(String argumentos[]) {
        int i=5, j=0;
        int k=i/j; // División por cero
    }
}
```

Produce la siguiente salida al ser ejecutado:

```
java.lang.ArithmeticException: / by zero
    at Excepcion.main(Excepcion.java:4)
```

Lo que ha ocurrido es que la máquina virtual Java ha detectado una condición de error y ha creado un objeto de la clase `java.lang.ArithmeticException`. Como el método donde se ha producido la excepción no es capaz de tratarla, se trata por la máquina virtual Java, que muestra el mensaje de error anterior y finaliza la ejecución del programa.

4.1 Lanzamiento de excepciones (*throw*).

Como se ha comentado anteriormente, un método también es capaz de lanzar excepciones. Por ejemplo, en el siguiente programa se genera una condición de error si el dividendo es menor que el divisor:



```
class LanzaExcepcion {
    public static void main(String argumentos[])
        throws ArithmeticException {
        int i=1, j=2;
        if (i/j < 1)
            throw new ArithmeticException();
        else
            System.out.println(i/j);
    }
}
```

Tratamiento de excepciones.

Genera el siguiente mensaje:

```
java.lang.ArithmeticException
    at
LanzaExcepcion.main(LanzaExcepcion.java:5)
```

En primer lugar, es necesario declarar todas las posibles excepciones que es posible generar en el método, utilizando la cláusula `throws` de la declaración de métodos³⁵.

Para lanzar la excepción es necesario crear un objeto de tipo `Exception` o alguna de sus subclase (por ejemplo: `ArithmeticException`) y lanzarlo mediante la instrucción `throw`.

4.2 Tratamiento de excepciones.

En Java, de forma similar a C++ se pueden tratar las excepciones previstas por el programador utilizando unos mecanismos, los manejadores de excepciones, que se estructuran en tres bloques:

- El bloque `try`.
- El bloque `catch`.
- El bloque `finally` (no existente en C++).

4.2.1 Manejador de excepciones.

Un manejador de excepciones es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar.

³⁵ Ver punto 3.4.1.7 Lista de excepciones potenciales. En página 101.

4.2.1.1 El bloque try.

Lo primero que hay que hacer para que un método sea capaz de tratar una excepción generada por la máquina virtual Java o por el propio programa mediante una instrucción `throw` es encerrar las instrucciones susceptibles de generarla en un bloque `try`.

```
try {  
    BloqueDeInstrucciones  
}
```

Cualquier excepción que se produzca dentro del bloque `try` será analizado por el bloque o bloques `catch` que se verá en el punto siguiente. En el momento en que se produzca la excepción, se abandona el bloque `try` y, por lo tanto, las instrucciones que sigan al punto donde se produjo la excepción no serán ejecutadas.

Cada bloque `try` debe tener asociado por lo menos un bloque `catch`.

4.2.1.2 El bloque catch.

```
try {  
    BloqueDeInstrucciones  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
}
```

Por cada bloque `try` pueden declararse uno o varios bloques `catch`, cada uno de ellos capaz de tratar un tipo u otro de excepción.

Para declarar el tipo de excepción que es capaz de tratar un bloque `catch`, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.

Tratamiento de excepciones.



```
class ExcepcionTratada {
    public static void main(String argumentos[]) {
        int i=5, j=0;
        try {
            int k=i/j;
            System.out.println("Esto no se va a ejecutar.");
        } catch (ArithmeticException ex) {
            System.out.println("Ha intentado dividir por cero");
        }
        System.out.println("Fin del programa");
    }
}
```

Salida:

```
Ha intentado dividir por cero
Fin del programa
```

Cuando se intenta dividir por cero, la máquina virtual Java genera un objeto de la clase `ArithmeticException`. Al producirse la excepción dentro de un bloque `try`, la ejecución del programa se pasa al primer bloque `catch`. Si la clase de la excepción se corresponde con la clase o alguna subclase de la clase declarada en el bloque `catch`, se ejecuta el bloque de instrucciones `catch` y a continuación se pasa el control del programa a la primera instrucción a partir de los bloques `try-catch`.

Como puede verse, también se podría haber utilizado en la declaración del bloque `catch`, una superclase de la clase `ArithmeticException`. Por ejemplo:

```
catch (RuntimeException ex)
    0
catch (Exception ex)
```

Sin embargo, es mejor utilizar excepciones lo más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error y si “se meten todas las condiciones en el mismo saco”, seguramente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

4.2.1.3 El bloque *finally*.

```
try {  
    BloqueDeInstrucciones  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
}  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
}  
finally {  
    BloqueFinally  
}
```

El bloque `finally` se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque `try` y en los bloques `catch`.

4.3 Jerarquía de excepciones.

Las excepciones son objetos pertenecientes a la clase `Throwable` o alguna de sus subclases.

Dependiendo del lugar donde se produzcan existen dos tipos de excepciones:

- 1) Las excepciones **síncronas** no son lanzadas en un punto arbitrario del programa sino que, en cierta forma, son previsibles en determinados puntos del programa como resultado de evaluar ciertas expresiones o la invocación de determinadas instrucciones o métodos.
- 2) Las excepciones **asíncronas** pueden producirse en cualquier parte del programa y no son tan “previsibles”. Pueden producirse excepciones asíncronas debido a dos razones:

Tratamiento de excepciones.

- La invocación del método `stop()` de la clase `Thread` que se está ejecutando. (☞ Véase el capítulo 6 **Threads**, en la página 199).

- Un error interno en la máquina virtual Java.

Dependiendo de si el compilador comprueba o no que se declare un manejador para tratar las excepciones, se pueden dividir en:

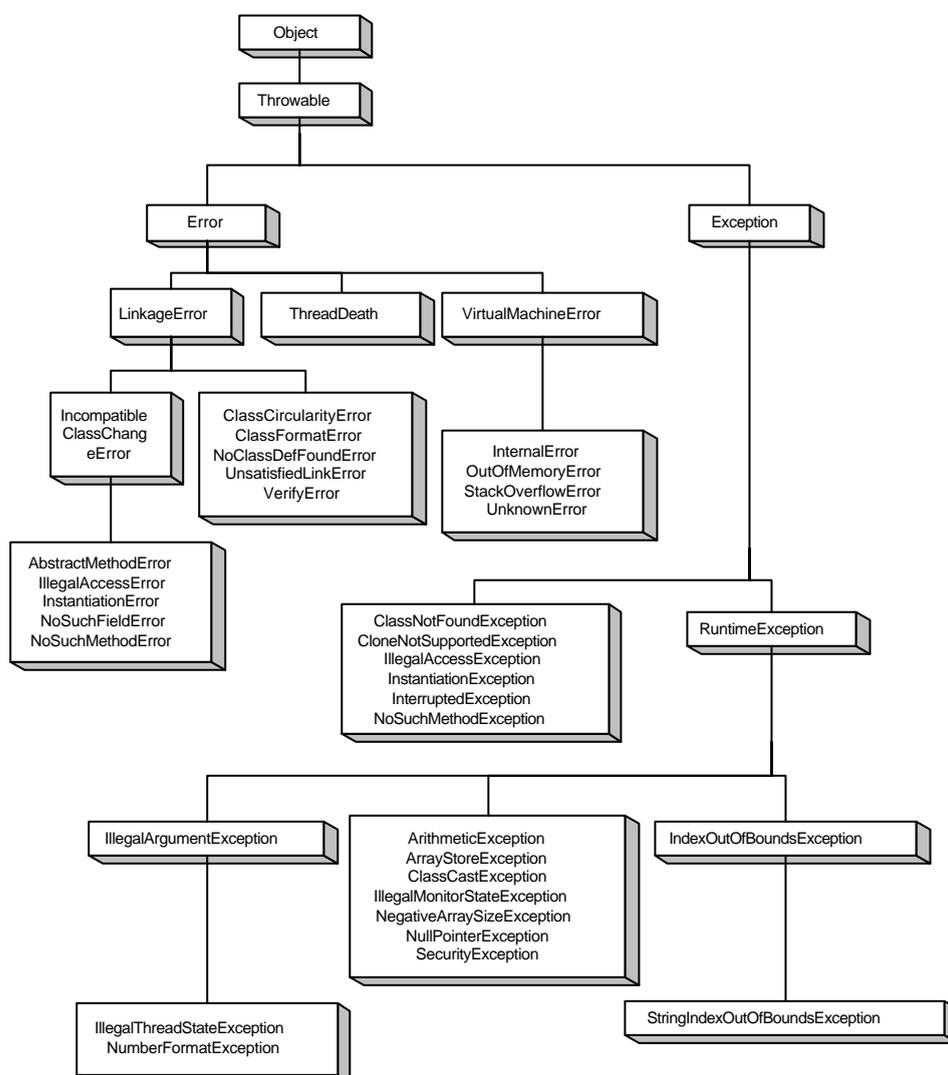
- 1) Las excepciones **comprobables** son repasadas por el compilador Java durante el proceso de compilación, de forma que si no existe un manejador que las trate, generará un mensaje de error.

- 2) Las excepciones **no comprobables** son la clase `RuntimeException` y sus subclases junto con la clase `Error` y sus subclases.

También pueden definirse por el programador subclases de las excepciones anteriores. Las más interesantes desde el punto de vista del programador son las subclases de la superclase `Exception` ya que éstas pueden ser comprobadas por el compilador.

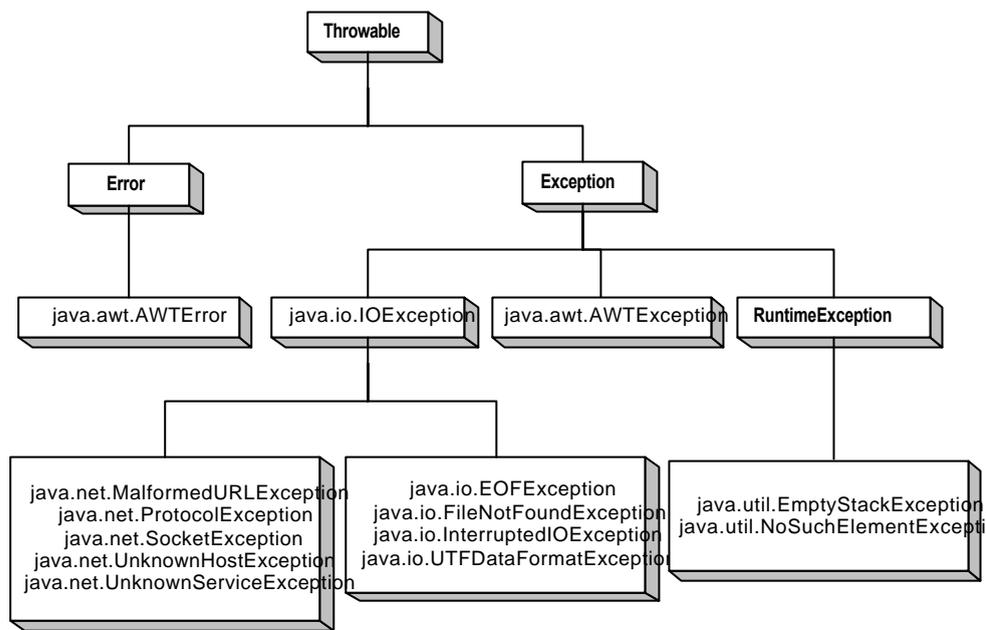
Tratamiento de excepciones.

La jerarquía de excepciones existentes en el paquete `java.lang` es la siguiente:



Tratamiento de excepciones.

Las excepciones en otros paquetes Java son:



```
class Nif {
    int dni;
    char letra;
    static final char tabla[]={'T','R','W','A','G','M','Y',
                               'F','P','D','X','B','N','J',
                               'Z','S','Q','V','H','L','C','K','E'};

    public Nif(int ndni,char nletra) throws NifException{
        // debido a la cláusula throws, este método es capaz
        // de generar excepciones de la clase NifException
        if (Character.toUpperCase(nletra)==tabla[ndni%23]) {
            // si la letra es correcta, almacenarla en el atributo
            dni=ndni;
            letra=Character.toUpperCase(nletra);
        }
        else
            // si la letra es incorrecta, generar una excepción
            throw new LetraNifException("Letra de NIF incorrecta");
    }
}
```

Tratamiento de excepciones.

```
}

public Nif(int ndni) {
    dni=ndni;
    letra=tabla[dni%23];
}

public Nif(String sNif) throws NifException,
    LetraNifException {
    // debido a la cláusula throws, este método es capaz
    // de generar excepciones de la clase NifException y
    // de la clase LetraNifException, aunque en realidad
    // sería suficiente con NifException, ya que la clase
    // LetraNifException es una subclase de ésta.
    char letraAux;
    StringBuffer sNumeros= new StringBuffer();
    int i,ndni;
    for (i=0;i<sNif.length();i++) {
        if ("1234567890".indexOf(sNif.charAt(i))!=-1) {
            sNumeros.append(sNif.charAt(i));
        }
    }
    try {
        dni=Integer.parseInt(sNumeros.toString());
        letraAux=Character.toUpperCase(sNif.charAt(
            sNif.length()-1));
    } catch (Exception ex) {
        // este bloque catch intercepta cualquier tipo de
        // excepción, incluso NumberFormatException
        throw new NifException("NIF incorrecto");
    }
    letra=tabla[dni%23];
    if ("ABCDEFGHIJKLMNPOQRSTUVWXYZ".indexOf(letraAux)!=-1) {
        // es una letra correcta
        if (letraAux!=letra) {
            // pero no la adecuada para el NIF
            throw new
                LetraNifException("Letra de NIF incorrecta");
        }
    } else letra=tabla[dni%23];
}

public char obtenerLetra() {
    return letra;
}

public int obtenerDni() {
    return dni;
}

public String toString() {
    // redefinición del método toString() para que devuelva
```

Tratamiento de excepciones.

```
// un valor más significativo.
    return (String.valueOf(dni)+String.valueOf(letra));
}

public String toStringConFormato() {
    String sAux= String.valueOf(dni);
    StringBuffer s = new StringBuffer();
    int i;
    for (i=sAux.length()-1;i>2;i-=3) {
        s.insert(0,sAux.substring(i-2,i+1));
        s.insert(0,".");
    }
    s.insert(0,sAux.substring(0,i+1));
    s.append('-');
    s.append(letra);
    return (s.toString());
}

static char letraNif(int ndni) {
    return tabla[ndni%23];
}

static char letraNif(String sDni) throws NifException {
    Nif j = new Nif(sDni);
    return j.obtenerLetra();
}
}

class NifException extends Exception {
    public NifException() { super(); }
    public NifException(String s) { super(s); }
}

class LetraNifException extends NifException {
    public LetraNifException() { super(); }
    public LetraNifException(String s) { super(s); }
}
}
```

En este programa se declara una clase **Nif** con dos atributos: dni, que almacena el número de DNI; y letra, que almacena la letra del NIF.

Se han declarado dos tipos de excepciones:

- La clase `NifException` como subclase de la clase `Exception`.
- La clase `LetraNifException` como subclase de la anterior.

Las excepciones tienen dos constructores, uno sin parámetros y otro que acepta un `String` (correspondiente al mensaje de error que se mostrará por la máquina

Tratamiento de excepciones.

virtual). En el ejemplo se han sobrescrito los constructores para introducir un mensaje de error en la excepción (“Letra de NIF incorrecta” y “NIF incorrecto” respectivamente). Si no se desea introducir ningún mensaje, no es necesario sobrescribir los constructores y la declaración de las Excepciones es más simple:

```
class NifException extends Exception {  
    }  
class LetraNifException extends NifException {  
    }
```

Por ser subclasses de la superclase `Exception`, ambos tipos de excepción son comprobables en tiempo de compilación.

El siguiente programa generaría errores de compilación:



```
class VerificaNif {  
    public static void main(String argumentos[]) {  
        Nif n;  
        if (argumentos.length!=1) {  
            System.out.println("Uso: VerificaNif NIF");  
            return;  
        }  
        else {  
            n = new Nif(argumentos[0]);  
            System.out.println("Nif: "+n.toStringConFormato());  
        }  
    }  
}
```

Los errores de compilación generados son:

Tratamiento de excepciones.

```
VerificaNif.java:9: Exception LetraNifException must be caught, or it must be declared in the throws clause of this method.
```

```
    n = new Nif(argumentos[0]);
```

```
    ^
```

```
VerificaNif.java:9: Exception NifException must be caught, or it must be declared in the throws clause of this method.
```

```
    n = new Nif(argumentos[0]);
```

```
    ^
```

2 errors

NOTA: Si la clase `NifException` se hubiera declarado como subclase de `RuntimeException` en lugar de `Exception`, el compilador no mostraría ningún mensaje de error, ya que la clase `RuntimeException` es NO comprobable en tiempo de compilación.

Si el método `main()` se hubiera declarado de esta forma:

```
public static void main(String argumentos[]) throws NifException {
```

Entonces no se habrían generado los errores de compilación, ya que mediante esta declaración se está indicando que las excepciones de la clase (o subclases) `NifException` no se tratan en este método sino que serán tratadas en el método que realice una llamada al mismo (en este caso no hay ninguno, únicamente la máquina virtual Java). En el caso en que se introduzca un `Nif` incorrecto de la siguiente forma:

```
java VerificaNif 18957690R
```

Se creará una excepción que es tratada por la máquina virtual, que lo que hace es mostrar un mensaje y finalizar el programa:

```
LetraNifException: Letra de NIF incorrecta
    at Nif.<init>(Nif.java:32)
    at
VerificaNif.main(VerificaNif.java:9)
```

Si lo que se pretende es que la excepción sea tratada por el propio método `main()`, hay que declarar un bloque `try` junto con los bloques `catch` adecuados:

Tratamiento de excepciones.



```
class VerificaNif {
    public static void main(String argumentos[]) {
        Nif n;
        if (argumentos.length!=1) {
            System.out.println("Uso: VerificaNif NIF");
            return;
        }
        else {
            try {
                n = new Nif(argumentos[0]);
                System.out.println("El NIF es correcto.");
            }
            catch (LetraNifException ex) {
                System.out.println(
                    "La letra del NIF es incorrecta");
            }
            catch (NifException ex) {
                System.out.println(
                    "Construcción de NIF incorrecta");
            }
            finally {
                System.out.println("Que tenga un buen día");
            }
        }
    }
}
```

En este ejemplo, el programa acepta un NIF en la línea de comandos. En caso de que el NIF sea correcto, no se produce ninguna excepción y se ejecuta la línea: `System.out.println("El NIF es correcto.");`

Si al crear el NIF (`new Nif (argumentos[0]);`) se produce un error, entonces esa excepción será comprobada por el primer bloque `catch`. Si la excepción se corresponde con la clase `LetraNifException`, se muestra el mensaje: La letra del NIF es incorrecta.

Si la excepción generada no pertenece a la clase `LetraNifException`, se comprueba si pertenece a la clase `NifException`, en cuyo caso, se mostraría el mensaje: `Construcción de NIF incorrecta`.

En el caso de que no hubiéramos querido diferenciar entre una letra de NIF incorrecta y un NIF construido incorrectamente, podríamos haber utilizado un

Tratamiento de excepciones.

solo bloque `catch` con la clase `NifException`, ya que esta clase es superclase de `LetraNifException` y por lo tanto abarca a ambas excepciones.

Por último, en cualquier caso, tanto si se produce una excepción como si no, o incluso si se produce una excepción no contemplada en los bloques `catch`, se mostrará el mensaje:

Que tenga un buen día.

4.4 Ventajas del tratamiento de excepciones.

Las ventajas de un mecanismo de tratamiento de excepciones como este son varias:

Separación del código “útil” del tratamiento de errores.

- Propagación de errores a través de la pila de métodos.
- Agrupación y diferenciación de errores.
- Claridad del código y obligación del tratamiento de errores.

4.4.1 Separación del código útil del tratamiento de errores.

Supóngase que se quiere realizar un bloque de instrucciones que realiza un procesamiento secuencial de un fichero:

```
{
  abrir_fichero("prueba");
  Mientras ! Fin_Fichero("prueba") {
    auxiliar = leer_registro("prueba");
    procesar(auxiliar);
  }
  cerrar_fichero("prueba");
}
```

Si se quiere tener en cuenta los posibles errores, debería hacerse algo parecido a esto:

```
{
  coderror = abrir_fichero("prueba");
  if (coderror ==0) {
    Mientras( ! Fin_Fichero("prueba")) y ( coderror==0 ) {
      auxiliar = leer_registro("prueba");
    }
  }
}
```

Tratamiento de excepciones.

```
if (auxiliar != ERROR) {
    coderror=procesar(auxiliar);
    if (coderror != 0)
        coderr = -3;
    };
else
    coderror = -2;
}
if(cerrar_fichero("prueba")!=0)
    coderror=-4;
};
else
    coderror = -1;
}
```

Por último faltaría tratar cada uno de los posibles errores, por ejemplo mediante una construcción del tipo switch. Nótese que las instrucciones “útiles” son las que se encuentran en **negrita**.

En Java, el código sería algo parecido a esto:

```
try {
    abrir_fichero("prueba");
    Mientras ! Fin_Fichero("prueba") {
        auxiliar = leer_registro("prueba");
        procesar(auxiliar);
    }
    cerrar_fichero("prueba");
} catch (ExceptionAbrirFichero) {
    Hacer algo;
} catch (ExceptionLeerRegistro) {
    Hacer algo;
} catch (ExceptionProcesar) {
    Hacer algo;
} catch (ExceptionCerrarFichero) {
    Hacer algo;
}
```

En este caso se ha incluido la estructura equivalente a la que falta en el caso anterior (switch).

En Java, el código útil (en negrita) se encuentra agrupado. De esta forma se consigue una mayor claridad en el código que realmente interesa.

4.4.2 Propagación de errores a través de la pila de métodos.

Supóngase que existen cuatro métodos y que el primero de ellos es el interesado en procesar una condición de error que se produce en el cuarto de ellos:

```
método1 ( ) {
    int error;
    error = método2 ( );
    if ( error != 0 )
        Procesar Error;
    else
        Hacer algo;
}
int método2 ( ) {
    int error;
    error = método3 ( );
    if ( error != 0 )
        return error;
    else
        Hacer algo;
}
int método3 ( ) {
    int error;
    error = método4 ( );
    if ( error != 0 )
        return error;
    else
        Hacer algo;
}
int método4 ( ) {
    int error;
    error = Proceso que puede generar un error ( );
    if ( error != 0 )
        return error;
    else
        Hacer algo;
}
```

Todos y cada uno de los métodos deben tener en cuenta el posible error y adaptar su código para tratarlo, así como devolver el código de error al método que lo invocó.

En Java:

```
método1 ( ) {
    try {
        método2 ( );
        Hacer algo;
    }
```

Tratamiento de excepciones.

```
        } catch (Exception ex)
            Procesar Error;
    }
}
int método2 ( ) throws Exception{
    método3 ( );
    Hacer algo;
}
int método3 ( ) throws Exception{
    método4 ( );
    Hacer algo;
}
int método4 ( ) throws Exception{
    Proceso que puede generar un error ( );
    Hacer algo;
}
```

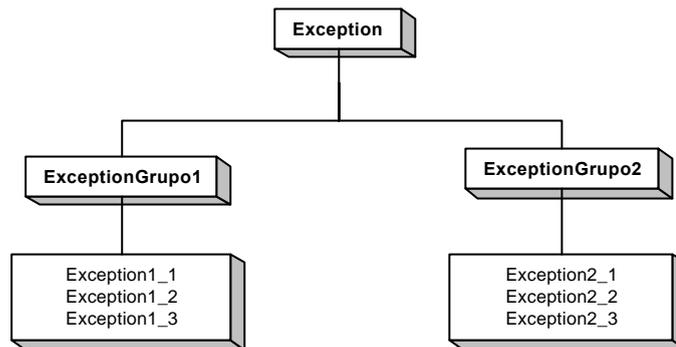
Cada uno de los métodos únicamente declara mediante la cláusula `throws` el /los error /errores que puede generar. Éstos se propagan automáticamente a través de la pila de llamadas sucesivas de métodos. El método interesado en tratar el error es el único que debe encargarse de “tratarlo”. Los demás métodos únicamente deben “ser conscientes” del error que puede producirse al llamar a otro método.

Si además el error se declara como una subclase de `Exception`³⁶, el compilador se encarga de mostrar un mensaje de error en caso de que el método no declare la cláusula **throws** adecuada (o trate la excepción), lo cual facilita la tarea del programador.

³⁶ Una excepción subclase de `Exception` es **comprobable** en tiempo de compilación. El compilador detecta que no se ha tenido en cuenta la excepción que puede generarse al llamar a un método.

4.4.3 Agrupación y diferenciación de errores.

Si se ha declarado una jerarquía de excepciones como la siguiente:



De esta forma:

```
class ExceptionGrupo1 extends Exception { }
class ExceptionGrupo2 extends Exception { }
class Exception1_1 extends ExceptionGrupo1 { }
class Exception1_2 extends ExceptionGrupo1 { }
class Exception1_3 extends ExceptionGrupo1 { }
class Exception2_1 extends ExceptionGrupo2 { }
class Exception2_2 extends ExceptionGrupo2 { }
class Exception2_3 extends ExceptionGrupo2 { }
```

Las excepciones pueden ser tratadas particularmente:

```
try {
    ...
} catch (Exception1_2 ex) {
    ...
}
```

O pueden tratarse como grupos:

```
try {
    ...
} catch (ExceptionGrupo1) {
    ...
}
```

Tratamiento de excepciones.

En este caso se detectarían las excepciones `Exception1_1`, `Exception1_2` y `Exception1_3` además de las excepciones de la clase `ExceptionGrupo1`.

También podrían detectarse todos los tipos de excepciones mediante: `catch (Exception exc)`.

4.4.4 Claridad del código y obligación del tratamiento de errores.

Como se ha visto en los puntos anteriores, el código resulta más claro ya que:

- Se separa el código “útil” del de tratamiento de errores.
- Se acerca el código de tratamiento de errores al método que realmente es el interesado en tratarlos.
- Cada tipo de error se trata en un bloque `catch` diferenciado.

Además, se obliga al programador a tratar los errores que puede generar un método, suponiendo que las excepciones declaradas correspondientes a los distintos tipos de error sean subclases de la clase `Exception`, que es comprobable por el compilador.

Quien programa un método declara las excepciones que puede generar mediante la cláusula `throws`. Si otro programador (o el mismo) desea utilizar ese método deberá utilizar una estructura `try-catch` para tratarlo o en caso contrario lanzarlo mediante la cláusula `throws` para que sea el método que invoque al mismo quien lo trate.

En cualquier caso, el programador que desea utilizar un método es consciente, porque el compilador se lo señala, de los tipos de errores (excepciones) que puede generar.

4.5 CUESTIONES

1. ¿Qué es una excepción dentro de un programa Java?
2. Explica cuál es la finalidad del bloque `try`. Pon un ejemplo de su uso.
3. ¿Cuándo se ejecuta el código que contiene un bloque `catch`?
4. ¿Qué instrucción empleamos para “lanzar” una excepción? ¿Para qué nos sirve lanzar una excepción?
5. El bloque `finally` nos permite especificar una acción a realizar después del bloque `try`, tanto si se produce una excepción como si no. ¿Qué ventajas aporta esto?
6. ¿Son todas las excepciones comprobables en tiempo de compilación? ¿Por qué?
7. Hay lenguajes que carecen de mecanismos de tratamiento de excepciones. ¿Qué ventajas crees que aporta este mecanismo en Java?
8. Cuando se produce una excepción en un método y no se ha declarado un tratamiento para la misma ¿qué sucede? ¿quién se encarga de su tratamiento?

5. LA ENTRADA/SALIDA EN JAVA.

El presente capítulo se van a presentan los conceptos básicos de la entrada y salida en Java. Existe una gran variedad de clases para realizar operaciones de entrada y salida de datos, por lo que únicamente se estudiaran algunas de ellas. Del capítulo se ha extraído un apéndice (↪ E Clases sobre streams y ficheros. Página 305) con los constructores y métodos de las clases implicadas para aligerarlo. Este apéndice puede irse consultando a medida que se lee este capítulo para ampliar algún conocimiento.

5.1 *Streams.*

Los *streams* son flujos secuenciales de bytes.

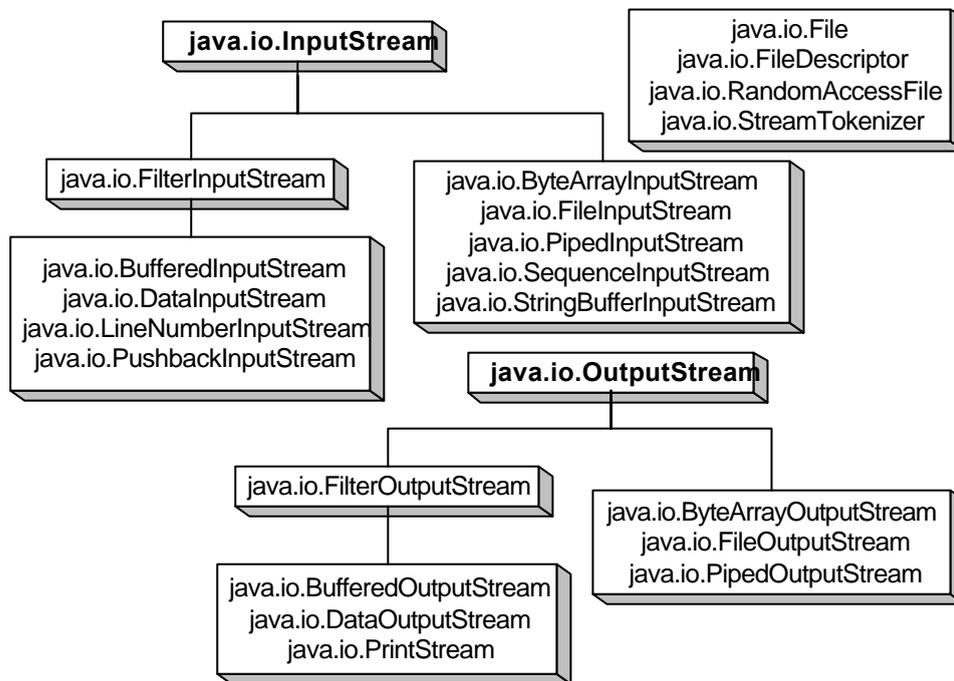
Para que un programa pueda leer datos de alguna fuente, debe crear un ***stream de entrada*** conectado a ésta; una fuente típica puede ser el teclado o un fichero. Y para escribir datos hacia un destino, debe crear un ***stream de salida*** conectado a éste; un destino típico puede ser la pantalla o un fichero.

Java proporciona distintas clases para el manejo de estos flujos de información, todas ellas contenidas en el paquete `java.io`.

5.2 *Jerarquía de clases en el paquete java.io.*

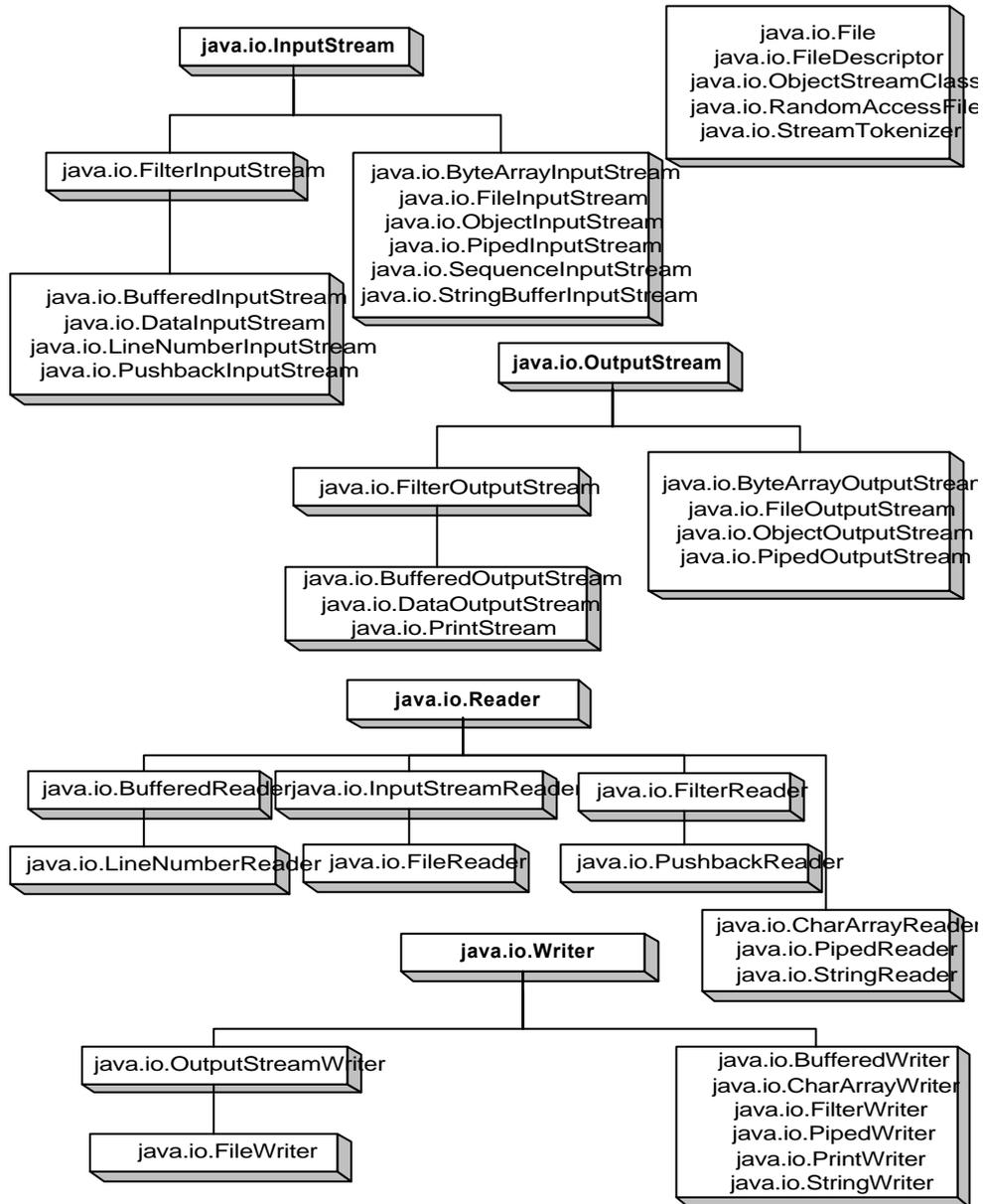
El JDK 1.1 añade ciertas clases en este paquete, con respecto al anterior 1.02, por lo que se mostrarán en dos esquemas distintos.

5.2.1 El paquete *java.io* en el JDK 1.02.



Las clases `InputStream` y `OutputStream` son clases abstractas, se utilizan como superclases.

5.2.2 El paquete java.io en el JDK 1.1.



El JDK 1.1 introduce una gran ventaja sobre la versión anterior: los *streams* de caracteres. La versión 1.02 únicamente trataba con *streams* de bytes, por lo cuál,

La Entrada/Salida en Java.

se realizaba una conversión de caracteres a bytes (con codificación dependiente de la plataforma). Mediante la introducción de las clases `Reader` y `Writer` junto con sus subclases, se proporcionan clases que escriben y leen caracteres codificados en Unicode (2 bytes) además de los *streams* que tratan con bytes.

Puede comprobarse en la tabla que existe una cantidad de clases como para abarcar por sí solas el contenido de un libro, por lo que sólo se estudiarán las más importantes para los objetivos de este libro.

Clases equivalentes entre *streams* de bytes y *streams* de caracteres.

	Clases de <i>streams</i> de caracteres	Clases de <i>streams</i> de bytes equivalentes
Entrada	<p><code>Reader</code> <code>BufferedReader</code> <code>LineNumberReader</code> <code>CharArrayReader</code> <code>InputStreamReader</code> <code>FileReader</code> <code>FilterReader</code> <code>PushbackReader</code> <code>PipedReader</code> <code>StringReader</code></p>	<p><code>InputStream</code> <code>BufferedInputStream</code> <code>LineNumberInputStream</code> <code>ByteArrayInputStream</code> (ninguna) <code>FileInputStream</code> <code>FilterInputStream</code> <code>PushbackInputStream</code> <code>PipedInputStream</code> <code>StringBufferInputStream</code></p>
Salida	<p><code>Writer</code> <code>BufferedWriter</code> <code>CharArrayWriter</code> <code>FilterWriter</code> <code>OutputStreamWriter</code> <code>FileWriter</code> <code>PrintWriter</code> <code>PipedWriter</code> <code>StringWriter</code></p>	<p><code>OutputStream</code> <code>BufferedOutputStream</code> <code>ByteArrayOutputStream</code> <code>FilterOutputStream</code> (ninguna) <code>FileOutputStream</code> <code>PrintStream</code> <code>PipedOutputStream</code> (ninguna)</p>

`Reader` / `InputStream` Clases abstractas para la entrada de caracteres / bytes.

`BufferedReader` / `BufferedInputStream` Clases para la entrada de caracteres / bytes con buffer en forma de líneas.

La Entrada/Salida en Java.

`LineNumberReader` / `LineNumberInputStream` Subclases de las anteriores. Éstas además tienen en cuenta los números de líneas.

`CharArrayReader` / `ByteArrayInputStream` Clases para la lectura desde una vector de caracteres.

`InputStreamReader` Transforma un *stream* de bytes en un *stream* de caracteres Unicode.

`FileReader` / `FileInputStream` Transforman un *stream* de bytes leído de un fichero en un *stream* de caracteres / bytes.

`FilterReader` / `FilterInputStream` Clases abstractas para la lectura filtrada de caracteres / bytes.

`PushbackReader` / `PushbackInputStream` Clases para lectura filtrada de caracteres / bytes, permitiéndose el retroceso.

`PipedReader` / `PipedInputStream` Clases para la lectura desde una tubería de escritura.

`StringReader` / `StringBufferInputStream` Clases para lectura desde un `String`.

`Writer` / `OutputStream` Clases abstractas para la escritura de caracteres / bytes.

`BufferedWriter` / `BufferedOutputStream` Clases para la escritura de caracteres / bytes con *buffer* y separadores de línea dependientes de la plataforma.

`CharArrayWriter` / `ByteArrayOutputStream` Clases para escritura a un vector de caracteres.

`FilterWriter` / `FilterOutputStream` Clases abstractas para la escritura filtrada de caracteres / bytes.

`OutputStreamWriter` Clase para transformar un *stream* de caracteres Unicode en un *stream* de bytes.

`FileWriter` / `FileOutputStream` Clases para escribir en un fichero a partir de un *stream* de caracteres / bytes.

La Entrada/Salida en Java.

`PrintWriter` / `PrintStream` Clases para escribir valores sobre *streams* de salida.

`PipedWriter` / `PipedOutputStream` Clases para escribir hacia un `PipedReader` / `PipedInputStream`.

`StringWriter` Clase para escribir hacia un `String`.

En este capítulo se estudiarán únicamente las más importantes. En primer lugar se verán los métodos de los que disponen las clases `InputStream`, `OutputStream` y `PrintStream`, ya que la entrada/salida estándar hace uso de ellos.

5.3 Clases importantes para la Entrada/Salida estándar.

A continuación se verá, de forma superficial, la composición de las clases más importantes para la Entrada/Salida estándar en Java `InputStream`, `OutputStream` y `PrintStream`; aunque hay que señalar que no son únicamente importantes para la E/S estándar sino en general. Sobre todo las dos primeras, que son las clases de la que parten todas las demás que tratan con *streams* de entrada y salida de bytes.

5.3.1 La clase `InputStream`

```
public abstract class java.io.InputStream extends java.lang.Object {
    // Constructor
    public InputStream(); // llamado por las subclases
    // Métodos
    public int available(); // nº de bytes que pueden ser leídos sin bloquearse
    public void close(); // cierra el stream
    public void mark(int readlimit); // Realiza una marca para volver a ésta
    public boolean markSupported(); // true si permite mark()
    public abstract int read(); // método a redefinir que lee un byte.
    // Debe devolver -1 si es final de fichero.
    public int read(byte b[]); // método que lee sobre un vector
    public int read(byte b[], int desplaza, int longitud); // igual que anterior
    public void reset(); // vuelve al último punto marcado con mark()
    public long skip(long n); // salta n bytes
}
```

```
}
```

Esta clase es abstracta (`abstract`). Es la superclase de todas las clases que implementan un stream de entrada de bytes, como puede comprobarse visualmente en el gráfico de la jerarquía de clases..

Las aplicaciones que necesiten definir una subclase de `InputStream` siempre deben proporcionar un método que devuelva el siguiente byte de entrada (deben redefinir el método abstracto: `public abstract int read()`).

Todos los métodos, a excepción del constructor y los métodos `mark()` y `markSupported()` son susceptibles de generar excepciones de la clase `java.io.IOException`.

5.3.2 La clase `OutputStream`

```
public abstract class java.io.OutputStream extends java.lang.Object {  
    // Constructor  
    public OutputStream();  
    // Métodos  
    public void close(); // cierra el stream de salida  
    public void flush(); // vacía el buffer sobre la salida  
    public abstract void write(int b); // método a redefinir que escribe un byte  
    public void write(byte b[]); // método que escribe un vector de bytes  
    public void write(byte b[], int desplaza, int longitud); // igual que anterior  
}
```

Esta clase es una clase abstracta que es superclase de todas las clases que implementan un *stream* de bytes de salida.

Las aplicaciones que necesiten definir una subclase de `OutputStream` siempre deben proporcionar al menos un método que escriba un byte de salida (`public abstract void write(int b)`).

Todos los métodos, a excepción del constructor pueden generar una excepción de la clase `java.io.IOException`.

5.3.3 La clase *PrintStream*.

```
public abstract class java.io.PrintStream
    extends java.io.FilterOutputStream {
    // Constructores
    public PrintStream(OutputStream out);
    public PrintStream(OutputStream out, boolean autoFlush);
    // Métodos
    public boolean checkError(); // vacía el buffer y comprueba si hay error
    public void close(); // cierra el stream
    public void flush(); // vacía el buffer
    public void print(boolean b);
    public void print(char c);
    public void print(char s[]);
    public void print(int i);
    public void print(long l);
    public void print(float f);
    public void print(double d);
    public void print(String s);
    public void print(Object obj);
    public void println();
    public void println(boolean b);
    public void println(char c);
    public void println(char s[]);
    public void println(int i);
    public void println(long l);
    public void println(float f);
    public void println(double d);
    public void println(String s);
    public void println(Object obj);
    protected void setError(); // establece a cierto el estado de error checkError()
    public void write(int b); // escribe un byte sin codificar como carácter.
    public void write(byte b[], int desplaza, int longitud);
}
```

Esta clase se encarga de escribir valores y objetos en un *stream* de salida en forma de caracteres. Para ello realiza una conversión de caracteres a la codificación en bytes dependiente de la plataforma en caso necesario. También proporciona *buffering*.

La Entrada/Salida en Java.

Esta clase no genera excepciones. Para ver si se ha producido algún error hay que llamar al método `checkError()`.

Es mejor utilizar la clase `PrintWriter` en lugar de esta, que se mantiene únicamente por cuestiones de compatibilidad con el JDK 1.02.

Los constructores necesitan de un objeto de la clase `OutputStream` sobre el cual enviar los bytes de salida una vez formateados por los distintos métodos de los que consta la clase `PrintStream`. En el caso del segundo constructor, si el parámetro `autoFlush` es `true`, después de escribir los datos mediante algún método `println()`, se vacía el buffer llamando al método `flush()`.

- `public void print(boolean b);`
- `public void print(char c);`
- `public void print(char s[]);`
- `public void print(int i);`
- `public void print(long l);`
- `public void print(float f);`
- `public void print(double d);`
- `public void print(String s);`
- `public void print(Object obj);`
- `public void println();`
- `public void println(boolean b);`
- `public void println(char c);`
- `public void println(char s[]);`
- `public void println(int i);`
- `public void println(long l);`
- `public void println(float f);`
- `public void println(double d);`
- `public void println(String s);`
- `public void println(Object obj);`

En todos los casos se escribe el resultado de llamar al método `toString()` del parámetro, o, en caso de ser tipos de datos simples, de su clase equivalente (por ejemplo `int Integer`), y convertir los caracteres a bytes codificados dependiendo de la plataforma subyacente.

El método `println()`, a diferencia del método `print()` escribe un separador de línea después. Este separador de línea no tiene que ser necesariamente el carácter `'\n'`. Depende de la propiedad del sistema `line.separator`.

El método `println()` sin parámetros únicamente escribe un separador de línea.

NOTA: La clase `PrintStream` ha sido adaptada en el JDK 1.1 de forma que utiliza un `OutputStreamWriter`, que proporciona los terminadores de línea propios de cada plataforma, de forma que no existan problemas de incompatibilidades con sistemas que no utilizan ASCII.

5.4 Entrada/Salida estándar.

Con anterioridad ya se ha hecho uso de los *streams* de salida (*OutputStream*) al utilizar el método `System.out.println()`. En la clase `System` hay definido un objeto `out` de la clase `java.io.PrintStream`, que gestiona la salida estándar y que contiene los métodos `print()`, `println()` y `write()`.

En la clase `System` existen tres atributos estáticos (atributos de clase):

- `out` (objeto de la clase `PrintStream`): Es un *stream* de salida que dirige el flujo de bytes hacia la salida estándar (la pantalla).
- `in` (objeto de la clase `InputStream`): Es un *stream* de entrada que acepta un flujo de bytes de la entrada estándar (el teclado).
- `err` (objeto de la clase `PrintStream`): Es un *stream* de salida que dirige el flujo de bytes hacia la salida de error (la pantalla).

Estos tres atributos son estáticos y, por lo tanto, no necesita instanciarse ningún objeto de la clase `System` para poder utilizarlos. Además, son *streams* que ya se encuentran abiertos y listos para ser utilizados.

El siguiente ejemplo lee los caracteres introducidos por el teclado y los va almacenando en un `StringBuffer` hasta que se introduce un cambio de línea. En ese momento, se muestra por la salida estándar.



```
import java.io.*;
class Standar1 {
    public static void main(String arg[]) {
        StringBuffer salida = new StringBuffer();
        char caracter;
        System.out.println("\nEscribe el texto: ");
        try {
            do {
                caracter=(char)System.in.read();
                // Es necesaria una conversión de int a char
                // véase el apéndice de conversión de tipos.
                salida.append(caracter); }
            while (caracter!='\n');
        }
        catch (Exception e) {
```

La Entrada/Salida en Java.

```
        System.err.println(e);
    }
    System.out.println("\nLa línea escrita es: ");
    System.out.println(salida);
}
}
```

La salida por pantalla es la siguiente:

```
Escribe el texto:
Este es el texto que escribo
```

```
La línea escrita es:
Este es el texto que escribo
```

Este ejemplo es equivalente al anterior, pero la lectura se realiza de una sola vez con el método `read()` que almacena los caracteres en un vector de bytes.



```
import java.io.*;
class Standar2 {
    public static void main(String arg[]) {
        byte buffer[] = new byte[255];
        System.out.println("\nEscribe el texto: ");
        try {
            System.in.read(buffer, 0, 255);
        }
        catch (Exception e) {
            System.err.println(e);
        }
        System.out.println("\nLa línea escrita es: ");
        System.out.println(new String(buffer));
    }
}
```

5.5 Streams de caracteres.

Al igual que la totalidad de las clases que tratan con *streams* de bytes tiene como raíz las clases `InputStream` y `OutputStream` para entrada y salida respectivamente. En el caso de los *streams* de caracteres se dispone de las clases `Reader` y `Writer`.

La Entrada/Salida en Java.

Como fue comentado en el punto [5.2.2](#) El paquete `java.io` en el JDK 1.1. en la página 161, estas dos clases y sus subclases (todas las que tienen como sufijo **-Reader** o **-Writer**) proporcionan clases que escriben y leen caracteres codificados en Unicode (2 bytes) en lugar de realizar una conversión a bytes dependiente de la plataforma.

Aparte de éstas, las clases más importantes para la entrada por teclado y salida por pantalla son `PrintWriter` y `BufferedReader`.

5.5.1 La clase `Reader`.

```
public class java.io.Reader {  
    //Atributos  
    protected Object lock; // Objeto sobre el que se realizan bloqueos  
    // Constructores  
    protected Reader();  
    protected Reader(Object lock);  
    // Métodos  
    public abstract void close() throws IOException; // cierra el stream  
    public void mark(int readlimit) throws IOException; // Realiza una marca  
                                                // para volver a ésta con reset()  
    public boolean markSupported(); // true si permite mark()  
    public int read() throws IOException; // método a redefinir que lee un byte  
    public int read(char b[]) throws IOException;  
    public int read(char b[], int desplaza, int longitud) throws IOException;  
    public boolean ready() throws IOException; // true si está listo para leer  
    public void reset() throws IOException; // vuelve al último punto marcado  
    public long skip(long n) throws IOException // salta n caracteres  
}
```

La clase `Reader` es análoga a la clase `InputStream`, pero mientras esta última trata con *streams* de entrada de bytes, la clase `Reader` trata con *streams* de caracteres.

5.5.2 La clase *Writer*.

```
public class java.io.Reader {  
    //Atributos  
    protected Object lock; // Objeto sobre el que se realizan bloqueos  
    // Constructores  
    protected Writer();  
    protected Writer(Object lock);  
    // Métodos  
    public abstract void close() throws IOException; // cierra el stream  
    public abstract void flush() throws IOException; // vacía el buffer  
    public void write(int c) throws IOException // escribe un carácter cuyo  
                                                // código Unicode es c  
  
    public void write(char b[]) throws IOException;  
    public void write(char b[], int desplaza, int longitud) throws IOException;  
    public void write(String s) throws IOException;  
    public void write(String s, int desplaza, int longitud) throws IOException;  
}
```

La clase `Writer` es análoga a la clase `OutputStream`, pero mientras ésta última trata con *streams* de salida de bytes, la clase `Writer` trata con *streams* de caracteres.

5.5.3 La clase *BufferedReader*.

```
public class java.io.BufferedReader extends java.io. Reader {  
    // Constructores  
    public BufferedReader(Reader in);  
    public BufferedReader(Reader in, int tamaño);  
    // Métodos  
    public void close() throws IOException;  
    public void mark(int readlimit) throws IOException;  
    public boolean markSupported();  
    public int read() throws IOException;  
    public int read(char b[], int desplaza, int longitud) throws IOException;  
    public String readLine() throws IOException; // lee una línea de texto  
    public boolean ready() throws IOException;
```

```
public void reset() throws IOException;  
public long skip(long n) throws IOException;  
}
```

Un `BufferedReader` se construye a partir de un objeto de la clase `Reader`, pudiéndose indicar el tamaño del buffer de caracteres de entrada.

Únicamente añade un método con respecto a la clase `Reader`. Este método es `readLine()` y sirve para leer una línea de texto completa (hasta que se encuentra el carácter '\n', '\r' o ambos consecutivamente).

5.5.4 La clase `PrintWriter`.

```
public class java.io.PrintWriter extends java.io.Writer {  
    // Constructores  
    public PrintWriter(OutputStream out);  
    public PrintWriter(OutputStream out, boolean autoFlush);  
    public PrintWriter(Writer outW);  
    public PrintWriter(Writer outW, boolean autoFlush);  
    // Métodos  
    public boolean checkError();  
    public void close();  
    public void flush();  
    public void print(boolean b);  
    public void print(char c);  
    public void print(char s[]);  
    public void print(int i);  
    public void print(long l);  
    public void print(float f);  
    public void print(double d);  
    public void print(String s);  
    public void print(Object obj);  
    public void println();  
    public void println(boolean b);  
    public void println(char c);  
    public void println(char s[]);  
    public void println(int i);  
    public void println(long l);  
}
```

```
public void println(float f);
public void println(double d);
public void println(String s);
public void println(Object obj);
protected void setError();
public void write(char b[]);
public void write(char b[], int desplaza, int longitud);
public void write(int c);
public void write(String s);
public void write(String s, int desplaza, int longitud);
}
```

Para construir un objeto de esta clase es necesario realizarlo a partir de un objeto subyacente de la clase `OutputStream` o `Writer`. En el constructor puede indicarse, además, un parámetro de tipo `boolean`. Si su valor es `true`, indica que se vacíe el buffer (*flush*) al llamar a alguno de los métodos `println()`. Si su valor es `false`, no se vaciará el *buffer* hasta que se haga explícitamente mediante el método `flush()`.

Los cometidos de los métodos son los mismos que los declarados en la clase `PrintStream`, pero en este caso se trabaja con *streams* de caracteres.

5.6 Entrada por teclado y salida por pantalla.

Para la salida por pantalla debería utilizarse la clase `PrintWriter` y para la entrada por teclado, `BufferedReader` (aunque a veces, por razones de comodidad y prisa, se utilicen directamente `System.out` y `System.in`).



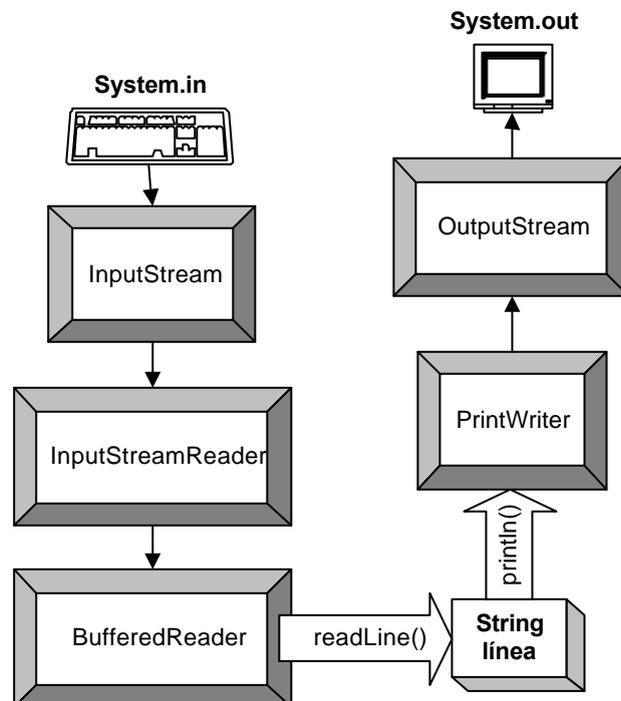
```
import java.io.*;
class E_S_1 {
    public static void main(String arg[]) {
        String línea=null;
        BufferedReader entrada = new BufferedReader(
            new InputStreamReader(System.in));
        PrintWriter salida = new PrintWriter(System.out,
true);
        salida.println("\nEscribe el texto: ");
```

La Entrada/Salida en Java.

```
try {
    línea = entrada.readLine();
}
catch (Exception e) {
    System.err.println(e);
}
salida.println("\nLa línea escrita es: ");
salida.println(línea);
}
}
```

Atención: es importante llamar al constructor de la clase `PrintWriter` con el segundo parámetro a `true` (*autoflush*), para que se visualice en pantalla la línea al realizar la llamada al método `println()`. Si no se pusiera este parámetro a `true`, no se visualizaría la línea hasta llamar al método `flush()` que realiza un volcado del *buffer*.

El esquema de declaración de objetos sería el siguiente:



5.7 *Ficheros.*

La mayoría de las aplicaciones de una cierta entidad hacen uso de ficheros para almacenar valores.

Antes de empezar a ver cómo utilizar las clases que crean los *streams* de entrada o salida sobre ficheros es necesario conocer algunas clases que permiten acceder a información referente a los mismos y que no intervienen en los flujos de datos hacia o desde ellos.

Para ello existen las clases `File` y `FileDescriptor`:

5.7.1 *La clase File.*

La clase `File` sirve para representar ficheros o directorios en el sistema de ficheros de la plataforma específica. Mediante esta clase pueden abstraerse las particularidades de cada sistema de ficheros y proporcionar los métodos necesarios para obtener información sobre los mismos.

```
public class java.io.File implements Serializable {  
    // Atributos  
    public static final String pathSeparator; // separador de paths  
    public static final char pathSeparatorChar; // separador de paths  
    public static final String separator; // separador de directorios en un path  
    public static final char separatorChar; // separador de directorios en un path  
    // Constructores  
    public File(String pathYNombre);  
    public File(String path, String nombre);  
    public File(File path, String nombre);  
    // Métodos  
    public boolean canRead(); // true si permite la lectura  
    public boolean canWrite(); // true si permite la escritura  
    public boolean delete(); // borrar el fichero  
    public boolean exists(); // true si existe el fichero  
    public String getAbsolutePath(); // obtiene el path absoluto  
    public String getCanonicalPath() throws IOException; // path canónico  
    public String getName(); // obtiene el nombre del fichero o directorio
```

La Entrada/Salida en Java.

```
public String getParent(); // obtiene el path del directorio padre  
public String getPath(); // obtiene el path del fichero  
public int hashCode(); // devuelve un código hash para el fichero  
public native boolean isAbsolute(); // true si el path es absoluto  
public boolean isDirectory(); // true si es un directorio y no un fichero  
public boolean isFile(); // true si es un fichero  
public long lastModified(); // Devuelve el momento de la última modificación  
public long length(); // Devuelve el tamaño en bytes del fichero  
public String[] list(); // Devuelve los ficheros de un directorio  
public String[] list(FilenameFilter filtro); // igual que anterior con filtro  
public boolean mkdir(); // Crea el directorio indicado  
public boolean mkdirs(); // Crea todos los directorios del path necesarios  
public boolean renameTo(File destino); // Cambia el nombre por destino  
}
```



```
import java.io.*;  
class DatosDeFile {  
    public static void main(String arg[]) {  
        File f = new File("sub\\prueba.txt");  
        System.out.println("pathSeparator: "+  
            File.pathSeparator);  
        System.out.println("pathSeparatorChar: "+  
            File.pathSeparatorChar);  
        System.out.println("separator: "+File.separator);  
        System.out.println("separatorChar: "+  
            File.separatorChar);  
        try {  
            System.out.println("canRead():"+f.canRead());  
            System.out.println("canWrite():"+f.canWrite());  
            System.out.println("exists():"+f.exists());  
            System.out.println("getAbsolutePath():"+  
                f.getAbsolutePath());  
            System.out.println("getCanonicalPath():"+  
                f.getCanonicalPath());  
            System.out.println("getName():"+f.getName());  
            System.out.println("getParent():"+f.getParent());  
            System.out.println("getPath():"+f.getPath());  
            System.out.println("hashCode():"+f.hashCode());  
            System.out.println("isAbsolute():"+  
                f.isAbsolute());  
            System.out.println("isDirectory():"+  
                f.isDirectory());  
        }  
    }  
}
```

La Entrada/Salida en Java.

```
        System.out.println("isFile():"+f.isFile());
        System.out.println("lastModified():"+
            f.lastModified());
        System.out.println("length():"+f.length());
    } catch (IOException e) {
        System.out.println(e); }
    }
}
```

Salida por pantalla:

```
pathSeparator: ;
pathSeparatorChar: ;
separator: \
separatorChar: \
canRead():true
canWrite():true
canRead():true
exists():true
getAbsolutePath():C:\datos\Victor\Documentos\java
\io\file\sub\prueba.txt
getCanonicalPath():C:\datos\Victor\Documentos\jav
a\io\file\sub\prueba.txt
getName():prueba.txt
getParent():sub
getPath():sub\prueba.txt
hashCode():-2131944982
isAbsolute():false
isDirectory():false
isFile():true
lastModified():868560320000
length():29
```

5.7.2 La clase *FileDescriptor*.

```
public class java.io.FileDescriptor {
    // Atributos
    public static final FileDescriptor in; // FD de la entrada estándar
    public static final FileDescriptor out; // FD de la salida estándar
    public static final FileDescriptor err; // FD de la salida de error
    // Constructor
```

La Entrada/Salida en Java.

```
public FileDescriptor();  
// Métodos  
public native void sync() throws SyncFailedException; // Este método no  
// termina hasta que se vacíen todos los buffers intermedios.  
public native boolean valid(); // true si el FD es válido  
}
```

Esta clase no debería utilizarse para instanciar objetos de la misma, sino que se suele utilizar algún método que devuelve un objeto de la clase `FileDescriptor`.

5.7.3 Acceso a ficheros secuenciales.

Para acceder a datos almacenados en ficheros sin demasiadas pretensiones de formateo, y con acceso secuencial, puede utilizarse la clase `FileInputStream` que implementa los métodos de la clase `InputStream` sin mayores funcionalidades.

Para almacenar datos en un fichero, puede crearse un *stream* de salida mediante la clase más simple que implementa los métodos de la clase `OutputStream`: `FileOutputStream`

Los constructores respectivos son:

Fichero de entrada

`FileInputStream(String nombre);`

`FileInputStream(File fichero);`

`FileInputStream(FileDescriptor fd);`

Fichero de salida

`FileOutputStream(String nombre);`

`FileOutputStream(File fichero);`

`FileOutputStream(FileDescriptor fd, boolean añadir);`

Pueden crearse *streams* de entrada o salida sobre ficheros utilizando como parámetro en el constructor:

- Un `String`: con el nombre del fichero junto con el path.

La Entrada/Salida en Java.

- Un objeto de la clase `File`: con el fichero a utilizar.
- Un objeto de la clase `FileDescriptor`: con el descriptor de fichero.

Métodos de lectura y escritura de bytes:

`FileInputStream`

`public native int read();`

`public int read(byte b[])`

`public int read(byte b[], int d, int l);`

`FileOutputStream`

`public native void write(int b);`

`public void write(byte b[])`

`public void write(byte b[], int d, int l);`

El puntero del fichero avanza según el número de bytes leídos en cada operación `read()`.

En ficheros de entrada, puede utilizarse el método `public native long skip(long n)` para saltar `n` bytes adelante.

Ambas clases permiten obtener el Descriptor de Fichero (FD) asociado mediante el método `public final FileDescriptor getFD()`, y cerrarlo mediante `public native void close()`.

El siguiente ejemplo toma un fichero de entrada y convierte las mayúsculas a minúsculas (siempre que esté en código ascii y únicamente los códigos del 65 al 90):



```
import java.io.*;
class Ficherol{
    public static void main(String arg[]) {
        FileInputStream f;
        int letra;
        try {
            f = new FileInputStream("prueba.txt");
```

La Entrada/Salida en Java.

```
while ((letra=f.read())!=-1) {
    if ((letra>64) && (letra<91))
        letra+=32;
    System.out.print((char)letra);
    f.close();
}
} catch(FileNotFoundException e) {
    System.err.println("Fichero no encontrado");
}
} catch(IOException e) {
    System.err.println("ERROR");
}
}
```

Si se pretende tratar con *streams* de caracteres en lugar de bytes, pueden utilizarse las nuevas clases equivalentes a `FileInputStream` y `FileOutputStream`: `FileReader` y `FileWriter` respectivamente. Cuyos constructores son:

Fichero de entrada

FileReader (String nombre);

FileReader(File fichero);

FileReader(FileDescriptor fd);

Fichero de salida

FileWriter(String nombre);

FileWriter(File fichero);

FileWriter(FileDescriptor fd);

FileWriter(FileDescriptor fd, boolean añadir);

Los métodos para leer y escribir caracteres son heredados de las superclases `InputStreamReader` y `OutputStreamWriter` y son:

FileReader

public int read();

public int read(char b[])

public int read(char b[], int d, int l);

FileWriter

public void write(int b);

public void write(char b[]);

public void write(char b[], int d, int l);

public void write(String s);

public void write(String s, int d, int l);

El siguiente ejemplo lee un fichero de caracteres y lo copia, tal cual, sobre un fichero de salida:



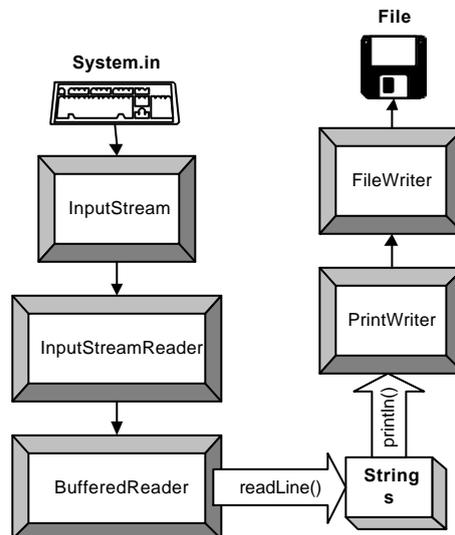
```
import java.io.*;
class Fichero2 {
    public static void main(String[] args) {
        try {
            File fEntrada = new File("prueba.entrada");
            File fSalida = new File("prueba.salida");
            FileReader frEntrada = new FileReader(fEntrada);
            FileWriter fwSalida = new FileWriter(fSalida);
            int c;
            while ((c = frEntrada.read()) != -1) {
                fwSalida.write(c);
            }
            frEntrada.close();
            fwSalida.close();
        } catch (FileNotFoundException e) {
            System.err.println("Fichero no encontrado");
        }
        catch (IOException e) {
            System.err.println("Error de E/S");
        }
    }
}
```

El siguiente ejemplo lee líneas de texto escritas desde el teclado y va escribiéndolas en el fichero de salida “prueba.salida” hasta que se encuentra la marca de final de fichero ^Z.

La Entrada/Salida en Java.



```
import java.io.*;
class Fichero3 {
    public static void main(String[] args) {
        BufferedReader teclado;
        InputStreamReader inStream;
        inStream=new InputStreamReader(System.in);
        teclado = new BufferedReader(inStream);
        File fSalida = new File("prueba.salida");
        try {
            String s;
            FileWriter fwSalida = new FileWriter(fSalida);
            PrintWriter w = new PrintWriter(fwSalida);
            while ((s = teclado.readLine())!=null ) {
                w.println(s);
            }
            fwSalida.close();
            inStream.close();
        } catch (FileNotFoundException e) {
            System.err.println("Fichero no encontrado");
        }
        catch (IOException e) {
            System.err.println("Error de E/S");
        }
    }
}
```



En este caso, se ha utilizado la clase `BufferedReader` envolviendo a la clase `InputStreamReader` porque añade el método `readLine()`, muy conveniente para los propósitos del programa. También se utiliza la clase `PrintWriter` envolviendo a la clase `FileWriter` para utilizar un método de escritura de línea completa como `println()`.

5.7.4 La clase `RandomAccessFile`.

```
public class java.io.RandomAccessFile implements DataOutput, DataInput {
    // Constructores
    public RandomAccessFile(File fichero, String modo) throws IOException;
    public RandomAccessFile(String nomF, String modo) throws IOException
    // El modo del fichero puede ser "r" (lectura) o "rw" (lectura/escritura).
    // Métodos
    public native void close() throws IOException; // Cierra el fichero y recursos
    public final FileDescriptor getFD() throws IOException; // Devuelve el FD
    public native long getFilePointer() throws IOException; // Devuelve la
        // posición del puntero del fichero (distancia en bytes desde el principio)
    public native long length() throws IOException; // n° de bytes del fichero
    public native int read() throws IOException; // Lee un byte del fichero
    public int read(byte b[]) throws IOException;
    public int read(byte b[], int desplaza, int longitud) throws IOException;
    public final boolean readBoolean() throws IOException;
    public final byte readByte() throws IOException;
    public final char readChar() throws IOException;
    public final double readDouble() throws IOException;
    public final float readFloat() throws IOException;
    public final int readInt() throws IOException;
    public final String readLine() throws IOException;
    public final long readLong() throws IOException;
    public final short readShort() throws IOException;
    public final int readUnsignedByte() throws IOException;
    public final int readUnsignedShort() throws IOException;
    public final void readFully(byte b[]) throws IOException;
    public final void readFully (byte b[], int desplaza, int longitud)
        throws IOException;
    public final String readUTF() throws IOException;
    public native void seek(long posición) throws IOException; // posiciona el
        // puntero del fichero directamente (en bytes)
    public int skipBytes(int n) throws IOException // Salta n bytes
}
```

La Entrada/Salida en Java.

```
public native void write(int b) throws IOException; // Escribe un byte
public void write(byte b[]) throws IOException;
public void write(byte b[], int desplaza, int longitud) throws IOException;
public final void WriteBoolean(boolean x) throws IOException;
public final void WriteByte(int x) throws IOException;
public final void WriteBytes(String x) throws IOException;
public final void WriteChar(char x) throws IOException;
public final void writeChars(String s) throws IOException
public final void WriteDouble(double x) throws IOException
public final void WriteFloat(float x) throws IOException
public final void WriteInt(int x) throws IOException
public final void WriteLong(long x) throws IOException
public final void WriteShort(int x) throws IOException
public final void writeUTF(String str) throws IOException;
}
```

Esta clase sirve para crear un objeto de acceso a un fichero de acceso directo a partir de un objeto de la clase `File` o, directamente, a partir de un `String` que contenga el nombre del fichero. Los constructores aceptan un segundo parámetro que indica el modo de apertura, pudiendo ser uno de los dos siguientes valores:

- “r” Apertura en modo lectura.
- “rw” Apertura en modo lectura/escritura.

Esta clase tiene dos métodos para mover el puntero del fichero:

seek(long posición) posicionamiento absoluto en el byte indicado por el parámetro.

skipBytes(int n) posicionamiento relativo saltando n bytes.

La posición actual (puntero del fichero) puede ser obtenida mediante el método:

getFilePointer().

5.7.5 Acceso a ficheros aleatorios.

Los ficheros de acceso aleatorio o directo permiten situar el puntero del fichero en una posición determinada sin tener que recorrer previamente el contenido del fichero como ocurre con los ficheros de acceso secuencial vistos anteriormente.

Para trabajar con ficheros aleatorios, únicamente es necesaria la clase `RandomAccessFile`, que proporciona directamente (no heredados) todos los métodos para leer y escribir datos del fichero (los indicados en las interfaces `DataInput` y `DataOutput`).

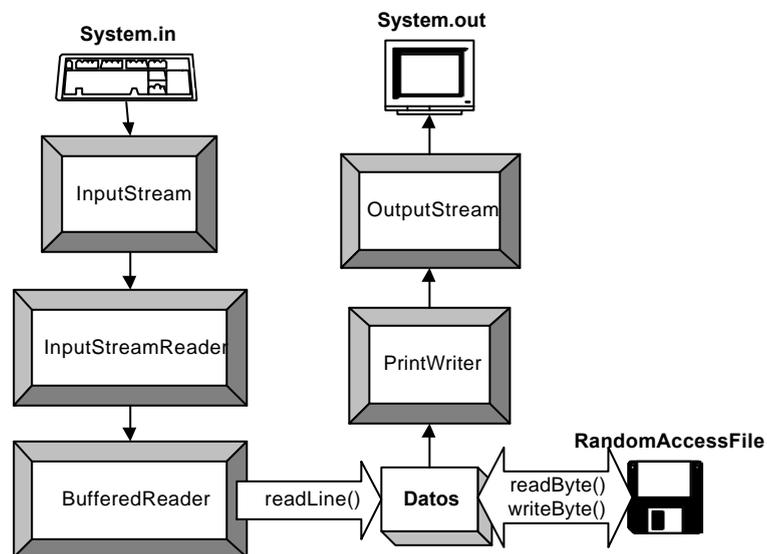
El siguiente programa solicita que le sea proporcionado el nombre de un fichero. Después pregunta el byte a modificar, muestra su valor y solicita el nuevo valor. Este proceso se repite hasta pulsar ^Z (final de fichero).



```
import java.io.*;
class Fichero4 {
    public static void main(String[] args) {
        BufferedReader teclado;
        InputStreamReader inStream;
        PrintWriter pantalla;
        String nombreFichero, posición, b;
        RandomAccessFile fichero;
        inStream=new InputStreamReader(System.in);
        teclado = new BufferedReader(inStream);
        File f;
        long punteroF;
        try {
            // PrintWriter con autoflush a true
            pantalla = new PrintWriter(System.out,true);
            pantalla.println("Nombre de fichero:");
            nombreFichero=teclado.readLine();
            f = new File(nombreFichero);
            // apertura del fichero en modo lectura/escrit.
            fichero = new RandomAccessFile(f,"rw");
            pantalla.println("^Z para terminar");
            pantalla.println("byte a examinar:");
            while ((posición = teclado.readLine())!=null ) {
                try {
                    // buscar la posición en el fichero.
                    // Integer.parseInt() obtiene int a partir
                    // de un String
                    fichero.seek(Integer.parseInt(posición));
                    // obtener el puntero del fichero
                    punteroF=fichero.getFilePointer();
```

La Entrada/Salida en Java.

```
// leer y mostrar el byte en la posición
pantalla.println("Valor del byte = "+
    fichero.readByte());
pantalla.println("Nuevo valor del byte:");
b=teclado.readLine();
// posicionar el puntero del fichero en
// el lugar guardado anteriormente
fichero.seek(punteroF);
// escribir el nuevo valor del byte
fichero.writeByte(Integer.parseInt(b));
} catch(IOException e) {
    System.err.println("No existe ese byte.");
}
catch(NumberFormatException e) {
    System.err.println("error num.");
}
pantalla.println("byte a examinar:");
}
teclado.close();
pantalla.close();
fichero.close();
} catch (IOException e) {
    System.err.println("Error de E/S");
}
}
```



A la hora de modificar un byte en el fichero de acceso directo, hay que tener en cuenta que primero se lee el byte mediante `readByte()`. Este método, además de

La Entrada/Salida en Java.

realizar la lectura, avanza el puntero de fichero un byte adelante, por lo que antes de realizar la escritura con `writeByte()`, habrá que retrocederlo 1 byte. También puede realizarse este proceso de la forma en que se ha hecho en el ejemplo; guardando la posición del puntero del fichero mediante `getFilePointer()` antes de realizar la lectura y restaurándolo mediante `seek()` antes de escribir.

5.8 Serialización de objetos.

De todos es conocida la importancia de poder salvar en memoria secundaria y permanente determinados estados o valores de una aplicación en un momento determinado para ser restablecidos en algún momento posterior.

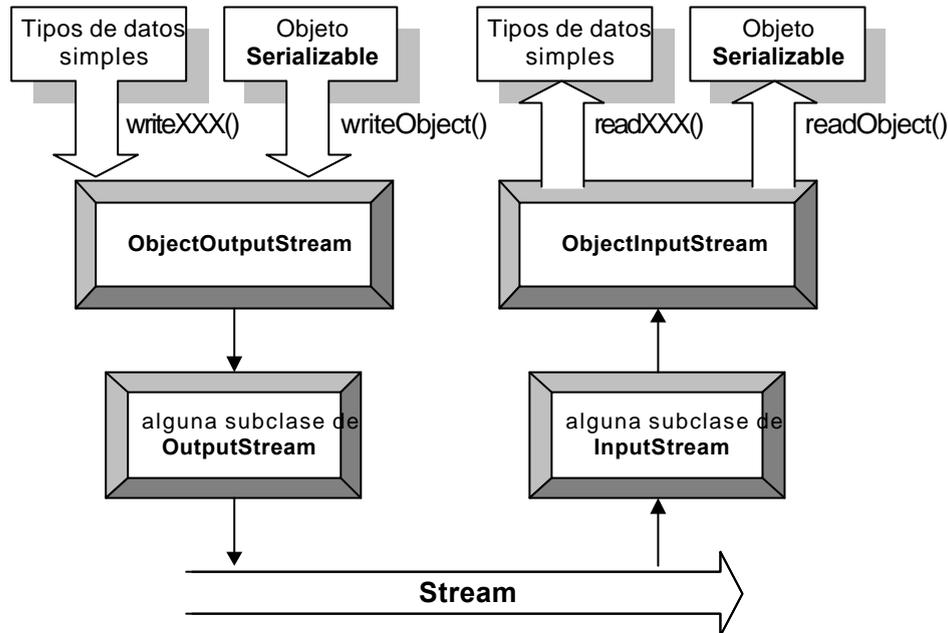
Si un programador desea guardar permanentemente el estado de un objeto puede utilizar las clases vistas hasta el momento para ir almacenando todos los valores de los atributos como valores char, int, byte, etc. Pero esto puede ser muy molesto y complicado, ya que un objeto puede tener atributos que sean otros objetos, que a su vez tengan atributos que hagan referencia a otras clases y así sucesivamente.

Pero existe una forma más cómoda de enviar objetos a través de un *stream* como una secuencia de bytes para ser almacenados en disco, y también para reconstruir objetos a partir de *streams* de entrada de bytes. Esto puede conseguirse mediante la “**serialización**” de objetos.

La **serialización** consiste en la transformación de un objeto Java en una secuencia de bytes para ser enviados a un *stream*.

Mediante este mecanismo pueden almacenarse objetos en ficheros o incluso en bases de datos como BLOBs (*Binary Large Object*), o se pueden enviar a través de *sockets*, en aplicaciones cliente/servidor de un equipo a otro, por ejemplo. Pero todo esto no es nuevo, ya podía hacerse antes de la serialización de objetos, lo que ocurre es que con este instrumento se consigue de una forma mucho más cómoda y sencilla.

5.8.1 Esquema de funcionamiento de la serialización.



Para enviar y recibir objetos serializados a través de un *stream* se utilizan las clases `java.io.ObjectOutputStream` para la salida y `java.io.ObjectInputStream` para la entrada.

La clase `ObjectOutputStream` necesita como parámetro en su constructor un objeto de la clase `OutputStream` (alguna de sus subclases, ya que `OutputStream` es abstracta):

```
public ObjectOutputStream(OutputStream out) throws IOException;
```

Si, por ejemplo, se utiliza como parámetro un objeto de la clase `FileOutputStream`, los objetos serializados a través de este *stream* serán dirigidos a un fichero.

La clase `ObjectInputStream` necesita como parámetro en su constructor un objeto de la clase `InputStream` (alguna de sus subclases, ya que `InputStream` es abstracta):

```
public ObjectInputStream(InputStream in)  
    throws IOException, StreamCorruptedException;
```

Si, por ejemplo, se utiliza como parámetro un objeto de la clase `FileInputStream`, los objetos se deserializan después de obtenerlos a través de este *stream* que, a su vez, obtiene los bytes de un fichero.

5.8.1.1 Objetos serializables.

Sólo los objetos de clases que implementen la interface `java.io.Serializable` o `java.io.Externalizable` o aquellos que pertenezcan a subclases de clases serializables pueden ser serializados.

En este capítulo únicamente van a estudiarse los objetos serializables mediante la interface `Serializable`, que es más sencilla porque automatiza el proceso. Los objetos que implementan la interface `Externalizable` deben encargarse de crear el formato en el que se almacenarán los bytes.

La interface `Serializable` no posee ningún método. Sólo sirve para marcar las clases que pueden ser serializadas.

Cuando un objeto es serializado, también lo son todos los objetos alcanzables desde éste (los atributos que son objetos), ignorándose todos los atributos `static`, `transient` y los no serializables.

No se almacenan los valores de los atributos `static` porque éstos pertenecen a la clase (no al objeto), y son compartidos por todos los objetos implementados a partir de ésta. Tampoco se almacenan los atributos `transient` porque esta palabra reservada se utiliza precisamente para eso, para no formar parte de las características permanentes de los objetos.



```
public class Persona implements java.io.Serializable{  
    // Esta clase debe ser Serializable para  
    // poder ser escrita en un stream de objetos  
    private String nombre;  
    private int edad;
```

La Entrada/Salida en Java.

```
public Persona(String s, int i) {
    nombre=s;
    edad=i;
}
public String toString() {
    return nombre+" "+edad;
}
}
```

Como puede comprobarse en el ejemplo anterior, en el fichero `Persona.java`, se ha declarado la clase `Persona` de forma que implemente la interface `Serializable` para poder ser serializada.

5.8.1.2 Escritura.

Para serializar objetos (también vectores y Strings) y escribirlos a través del *stream* de salida se llama al método `writeObject()` del objeto `ObjectOutputStream` creado:

```
public final void writeObject(Object objeto) throws IOException;
```

También pueden escribirse, además de objetos, valores de tipos de datos simples mediante cualquiera de los métodos de la interface `DataOutput`, que implementa la clase `ObjectOutputStream`³⁷:

write(int b);	writeChars(String x);
write(byte b[]);	writeDouble(double x);
write(byte b[], int desplaza, int longitud);	writeFloat(float x);
writeBoolean(boolean x);	writeInt(int x);
writeByte(int x);	writeLong(long x);
writeBytes(String x);	writeShort(int x);
writeChar(int x);	writeUTF(String x);

Todos estos métodos generan excepciones de la clase `IOException`.

³⁷ En realidad, la clase `ObjectOutputStream` no implementa la interface `DataOutput`, sino la interface `ObjectOutput`, que es una subinterface de `DataOutput`, por lo que hereda sus métodos.

La Entrada/Salida en Java.

Veamos un ejemplo en el que se crea un *stream* de salida, en el cual se escribirán: un objeto de la clase *Persona* (visto en el punto anterior), un objeto de la clase fecha (*Date*) y un entero:



```
import java.io.*;
import java.util.Date;
class SerialEscribe {
    public static void main(String arg[]) {
        try {
            // Un OutputStream subyacente sobre el que
            // escribir los bytes
            FileOutputStream f=new
                FileOutputStream("prueba.dat");
            // El objeto serializador
            ObjectOutputStream ost = new ObjectOutputStream(f);
            Persona persona=new Persona("Victor",30);
            Date fecha = new Date();
            ost.writeObject(persona);
            ost.writeObject(fecha);
            ost.writeInt(13);
            ost.flush(); // vaciar el buffer
            ost.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

En el caso de los objetos *persona* y *fecha* se utiliza el método `writeObject()`, y pueden ser serializados porque tanto la clase *Persona* como la clase *Date* implementan la interface *Serializable*; y en el caso del literal entero 13, se utiliza el método `writeInt()`.

5.8.1.3 Lectura.

Para deserializar objetos después de leerlos a través del stream de entrada se llama al método `readObject()` del objeto *ObjectInputStream* creado:

```
public final Object readObject() throws OptionalDataException,
    ClassNotFoundException, IOException;
```

La Entrada/Salida en Java.

También pueden leerse, además de objetos, valores de tipos de datos simples mediante cualquiera de los métodos de la interface `DataInput`, que implementa la clase `ObjectInputStream`³⁸:

Boolean readBoolean();	int readInt();
byte readByte();	String readLine();
char readChar();	long readLong();
double readDouble();	short readShort();
float readFloat();	int readUnsignedByte();
void readFully(byte b[]);	int readUnsignedShort();
void readFully(byte b[], int desplaza,	String readUTF();
int longitud);	int skipBytes(int n)

Todos estos métodos generan excepciones de la clase `IOException` y `EOFException`.

Para leer objetos y valores de tipos de datos simples, de un stream, evidentemente, deben ser recuperados en el mismo orden en que se introducen. Si se introducen objetos de las clases `C1`, `C2`, `C3`, `C2`, ... deben ser recuperados en ese mismo orden `C1`, `C2`, `C3`, `C2`, ...

El siguiente ejemplo lee y muestra los objetos y el entero guardado anteriormente en el fichero `prueba.dat`:



```
import java.io.*;
import java.util.Date;
class SerialLee {
    public static void main(String arg[]) {
        try {
            // Un InputStream subyacente del cuál
            // leer los bytes
            FileInputStream f=new
                FileInputStream("prueba.dat");
            // El objeto deserializador
            ObjectInputStream ost = new ObjectInputStream(f);
            Persona persona=(Persona)ost.readObject();
            Date fecha=(Date)ost.readObject();
            int entero=ost.readInt();
        }
    }
}
```

³⁸ En realidad, la clase `ObjectInputStream` no implementa la interface `DataInput`, sino la interface `ObjectInput`, que es una subinterface de `DataInput`, por lo que hereda sus métodos.

La Entrada/Salida en Java.

```
        System.out.println(persona);
        System.out.println(fecha);
        System.out.println(entero);
        ost.close();
    } catch (IOException e) {
        System.err.println(e);
    }
    catch (ClassNotFoundException e) {
        System.err.println(e);
    }
}
}
```

En los casos de lectura de objetos, es necesario realizar una conversión de tipo referencial, ya que el método `readObject()` devuelve un objeto de la clase `Object`. Para ello hay que anteponer el nombre de la clase entre paréntesis a la expresión a convertir.

La salida del programa según los datos almacenados en el ejemplo del punto anterior sería la siguiente:

```
Victor:30
Tue Jul 15 18:10:00 GMT+01:00 1997
13
```

5.8.2 Personalización en la serialización.

Una clase puede controlar por sí misma cómo será serializada, alterando qué datos serán almacenados o leídos, definiendo los métodos `writeObject()` y `readObject()` con, exactamente, las siguientes firmas:

- **private void readObject(java.io.ObjectInputStream in)
throws IOException, ClassNotFoundException;**
- **private void writeObject(java.io.ObjectOutputStream out)
throws IOException**

La Entrada/Salida en Java.

Estos métodos declaran los parámetros `ObjectInputStream` y `ObjectOutputStream`, por lo que puede accederse a sus métodos. Entre ellos se encuentran:

- **`public final void defaultWriteObject() throws IOException;`**

Este método escribe la información del objeto por defecto: todos sus atributos que no sean `static` o `transient` o no sean serializables. Puede utilizarse para realizar la escritura por defecto y, después, agregar al *stream* información adicional sobre el objeto.

- **`public final void defaultReadObject() throws IOException, ClassNotFoundException, NotActiveException;`**

Este método lee la información del objeto por defecto: todos sus atributos que no sean `static` o `transient` o no sean serializables. Puede utilizarse para realizar la lectura por defecto y, después, leer del *stream* información adicional sobre el objeto.

Véase el siguiente ejemplo que modifica la clase `Persona` de forma que serialice, además de los atributos propios de `Persona2` (`nombre` y `edad`), el momento(`Date`) en que se escribe:



```
import java.util.Date;
import java.io.*;
public class Persona2 implements java.io.Serializable{
    private String nombre;
    private int edad;
    public Persona2(String s, int i) {
        nombre=s;
        edad=i;
    }
    public String toString() {
        return nombre+":"+edad;
    }
    private void writeObject(java.io.ObjectOutputStream out)
        throws IOException {
        // Escribir los atributos por defecto
        out.defaultWriteObject();
        // Añadir la fecha de escritura
        out.writeObject(new Date());
    }
    private void readObject(java.io.ObjectInputStream in)
```

La Entrada/Salida en Java.

```
throws IOException, ClassNotFoundException {
// Leer los atributos por defecto
in.defaultReadObject();
// Leer la fecha guardada
Date fecha = (Date)in.readObject();
System.out.println("Objeto grabado en fecha: "+
    fecha);
}
}
```

Los programas para la escritura y para la lectura no necesitan de ninguna modificación especial:



```
import java.io.*;
class SerialEscribe2 {
    public static void main(String arg[]) {
        try {
            FileOutputStream f=new
                FileOutputStream("prueba.dat");
            ObjectOutputStream ost = new ObjectOutputStream(f);
            Persona2 persona=new Persona2("Victor",30);
            ost.writeObject(persona);
            ost.flush(); // vaciar el buffer
            ost.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```



```
import java.io.*;
class SerialLee2 {
    public static void main(String arg[]) {
        try {
            FileInputStream f=new
                FileInputStream("prueba.dat");
            ObjectInputStream ost = new ObjectInputStream(f);
            Persona2 persona=(Persona2)ost.readObject();
            System.out.println(persona);
            ost.close();
        } catch (IOException e) {
```

La Entrada/Salida en Java.

```
        System.err.println(e);
    }
    catch (ClassNotFoundException e) {
        System.err.println(e);
    }
}
}
```

5.9 CUESTIONES

1. ¿Qué cambio fundamental se ha introducido en la entrada-salida de la versión 1.02 a la 1.1? ¿Por qué podemos considerar esto como una mejora?
2. ¿Qué clase emplearías para escribir en un fichero un vector de caracteres? ¿Y si quieres escribir en la pantalla?
3. ¿Qué escribe un *stream* de caracteres? ¿Qué codificación emplea?
4. El método `println()` utiliza como separador de línea una constante, dependiente de plataforma. ¿Dónde se encuentra este valor?
5. ¿Qué sucede cuando leemos de un *stream* que está vacío, es decir, que no tiene caracteres pendientes? ¿Qué método podemos emplear para conocer cuántos caracteres hay pendientes de leer?
6. ¿Qué operación realiza el método `flush()` en un `OutputStream`? ¿Cuándo lo aplicarías?
7. ¿Qué clases deberían emplearse para realizar la entrada por teclado y la salida por pantalla en vez de `System.in` y `System.out`?
8. ¿Por qué deberemos colocar a `true` el segundo parámetro al instanciar la clase `PrintWriter`? ¿Qué sucedería si no lo hacemos?
9. ¿Cómo podemos averiguar si tenemos permiso de escritura en un archivo? (Por ejemplo en el archivo `C:\nopuedo.txt`).
10. ¿Cómo podemos obtener la lista de archivos que hay en un directorio del disco?

6. THREADS.

6.1 ¿Qué son los threads?

Todos los programadores conocen lo que es un proceso, la mayoría diría que es un programa en ejecución: tiene un principio, una secuencia de instrucciones y tiene un final.

Un *thread*³⁹ es un flujo simple de ejecución dentro de un programa. Hasta el momento, todos los programas creados contenían un único *thread*, pero un programa (o proceso) puede iniciar la ejecución de varios de ellos concurrentemente. Puede parecer que es lo mismo que la ejecución de varios procesos concurrentemente a modo del *fork()* en UNIX, pero existe una diferencia. Mientras en los procesos concurrentes cada uno de ellos dispone de su propia memoria y contexto, en los *threads* lanzados desde un mismo programa, la memoria se comparte, utilizando el mismo contexto y recursos asignados al programa (también disponen de variables y atributos locales al *thread*).

Un *thread* no puede existir independientemente de un programa, sino que se ejecuta dentro de un programa o proceso.

6.1.1 Un ejemplo, mejor que mil palabras.

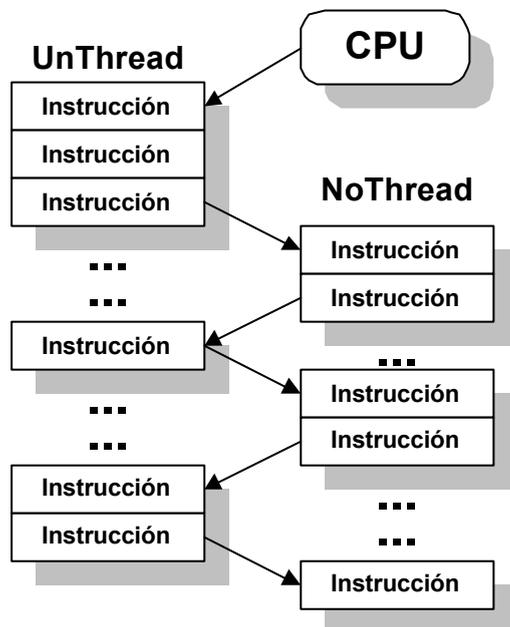


```
class UnThread {
    public static void main(String args[] ) {
        int i;
        NoThread t = new NoThread();
        t.start();
        for (i=1; i<=20; i++)
            System.out.print("SI ");
    }
}
```

³⁹ La palabra **thread** podría traducirse al español por hilo o hebra. En términos de programación también se conocen como “contextos de ejecución” o “procesos ligeros”.

Threads.

```
}  
  
class NoThread extends Thread {  
    public void run() {  
        int i;  
        for (i=1;i<=20; i++)  
            System.out.print("NO ");  
    }  
}
```



La salida de este programa es una serie alternativa de Síes y Noes:

```
SI NO NO NO SI SI SI  
NO NO NO NO NO NO NO  
NO NO NO NO NO NO NO  
NO NO NO SI SI SI SI  
SI SI SI SI SI SI SI  
SI SI SI SI SI
```

En el ejemplo se declara una clase principal (`UnThread`) que inicia su ejecución como un proceso con un único *thread* mediante su método `main()`, como ocurría en todos los programas vistos hasta ahora. En este proceso, se declara y se crea un *thread* (`NoThread t = new`

`NoThread();`). Después se inicia su ejecución mediante la llamada al método de la clase `Thread start()` (`t.start()`), con lo cuál comienza a ejecutarse el método `run()` redefinido en la clase `NoThread` (el método `start()` llama al método `run()`). Tenemos dos *threads* ejecutándose.

Una vez se inicia la ejecución del *thread*, el tiempo de la CPU se reparte entre todos los procesos y *threads* del sistema, con lo cuál, se intercalan instrucciones del método `main()` con instrucciones del método `run()` entre otras correspondientes a otros procesos (del sistema operativo y otros procesos de usuario que pudieran estar ejecutándose).

Threads.

6.1.2 Otro ejemplo



```
class DosThreads {
    public static void main(String args[] ) {
        NoThread n = new NoThread();
        SiThread s = new SiThread();
        n.start();
        s.start();
    }
}

class NoThread extends Thread {
    public void run() {
        int i;
        for (i=1;i<=20; i++)
            System.out.print("NO ");
    }
}

class SiThread extends Thread {
    public void run() {
        int i;
        for (i=1;i<=20; i++)
            System.out.print("SI ");
    }
}
```

La salida del programa, al igual que en el ejemplo anterior, es:

```
NO NO NO NO NO NO SI SI
SI SI SI SI SI NO NO SI SI NO NO SI NO NO NO NO NO
NO NO NO NO
```

En este caso se instancian dos *threads* y se llama a su ejecución mediante los métodos `start()`.

Estos dos *threads* se reparten el tiempo de la CPU y se ejecutan concurrentemente. Una vez que finalizan su ejecución, el programa termina.

Threads.

6.1.3 Y otro ejemplo más.



```
class UnThreadDosInstancias {
    public static void main(String args[] ) {
        SiNoThread s = new SiNoThread("SI");
        SiNoThread n = new SiNoThread("NO");
        s.start();
        n.start();
    }
}

class SiNoThread extends Thread {
    private String SiNo;
    static int Contador=0;
    public SiNoThread(String s) {
        super();
        SiNo=s;
    }
    public void run() {
        int i;
        for (i=1;i<=20; i++)
            System.out.print(++Contador+": "+SiNo+" ");
    }
}
```

Produce la siguiente salida:

```
1:SI 2:SI 3:SI 5:NO 6:NO 4:SI 8:SI 9:SI 10:SI 11:SI
12:SI 13:SI 14:SI 15:SI 7:NO 17:NO 16:SI 19:SI 20:SI
21:SI 22:SI 23:SI 24:SI 25:SI 18:NO 26:NO 27:NO 28:NO
29:NO 30:NO 31:NO 32:NO 33:NO 34:NO 35:NO 36:NO 37:NO
38:NO 39:NO 40:NO
```

En este caso se declaran dos instancias de una misma clase (`SiNoThread`) y se ejecutan concurrentemente. Cada una de ellas con sus propios atributos de objeto (`String SiNo`), pero comparten los atributos de clase⁴⁰ (`int Contador`).

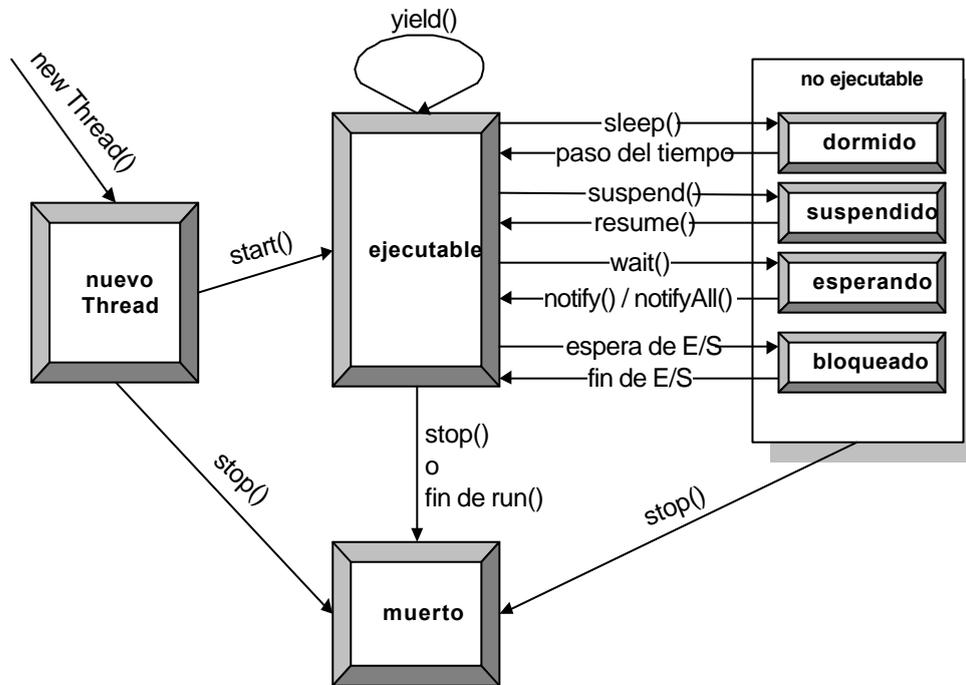
⁴⁰ Los atributos de clase son los declarados `static`.

Threads.

Puede comprobarse que el contador no ha funcionado todo lo correctamente que pudiera esperarse: 1,2,3,5,6,4,8, ... Esto es debido a que se ha accedido concurrentemente a una misma zona de memoria sin que se produzca exclusión mutua. Se verá cómo solucionar este problema en el punto 6.5 Threads y sincronismo. en la página 212.

6.2 Estado de un thread.

El ciclo de vida de un *thread* puede pasar por varios estados ilustrados en la siguiente figura:



Cuando se instancia un *thread*, se inicializa sin asignarle recursos. Está en el estado **nuevo Thread**. Un *thread* en este estado únicamente acepta las llamadas a los métodos `start()` o `stop()`.

La llamada al método `start()` asigna los recursos necesarios al objeto, lo sitúa en el estado **ejecutable** y llama al método `run()` del objeto. Esto no significa que el *thread* esté ejecutándose (existen multitud de sistemas que poseen una sola

Threads.

CPU que debe ser compartida por todos los *threads* y procesos) sino que está en disposición de ejecutarse en cuanto la CPU le conceda su tiempo.

Un *thread* en estado **ejecutable** puede pasar al estado **no ejecutable** por alguna de las siguientes razones: que sean invocados alguno de sus métodos `sleep()` o `suspend()`, que el *thread* haga uso de su método `wait()` o que el *thread* esté bloqueado esperando una operación de entrada/salida o que se le asigne algún recurso.

Un *thread* puede pasar al estado **muerto** por dos motivos: que finalice normalmente su método `run()` o que se llame a su método `stop()` desde cualquiera de sus posibles estados (nuevo *thread*, ejecutable o no ejecutable).

Un *thread* pasa del estado **no ejecutable** a **ejecutable** por alguna de las siguientes razones:

- **dormido:** que pase el tiempo de espera indicado por su método `sleep()`, momento en el cual, el *thread* pasará al estado ejecutable y, si se le asigna la CPU, proseguirá su ejecución.
- **suspendido:** que, después de haber sido suspendido mediante el método `suspend()`, sea continuado mediante la llamada a su método `resume()`.
- **esperando:** que después de una llamada a `wait()` se continúe su ejecución con `notify()` o `notifyAll()`.
- **bloqueado:** una vez finalizada una espera sobre una operación de E/S o sobre algún recurso.

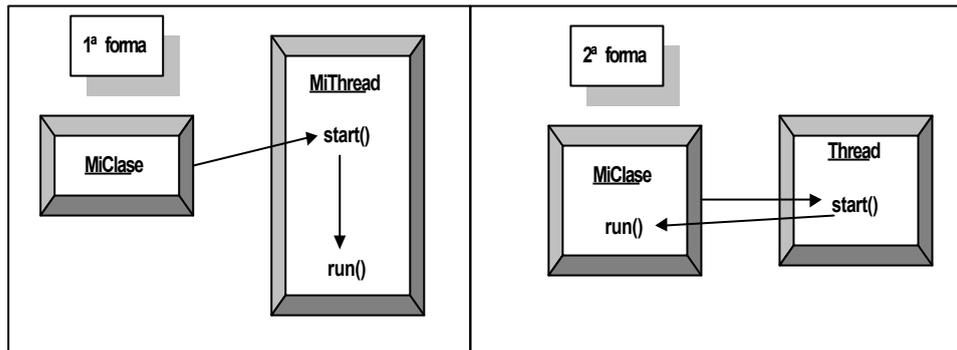
6.3 Creación de threads.

Pueden crearse *threads* de dos formas distintas: declarando una subclase de la clase `Thread` o declarando una clase que implemente la interface `Runnable` y redefiniendo el método `run()` y `start()` de la interface.

Se utilizará la primera forma, más evidente y sencilla, cuando la clase declarada no tenga que ser subclase de ninguna otra superclase.

Threads.

Se utilizará la segunda forma cuando la clase declarada tenga que ser subclase de una superclase que no es subclase de `Thread` o implemente el interface `Runnable`.



La forma en que se crean los *threads* como subclase de `Thread` ya se ha visto en puntos anteriores.

Creación de *threads* utilizando la interface `Runnable`:



```
class PrimerThread implements Runnable {
    Thread t;
    public void start() {
        t = new Thread(this);
        t.start();
    }
    public void run() {
        int i;
        for (i=1;i<=50;i++)
            System.out.println(i);
    }
}

class SegundoThread extends PrimerThread {
}

class ThreadRunnable {
    public static void main(String args[]) {
        SegundoThread s = new SegundoThread();
    }
}
```

Threads.

```
        SegundoThread ss = new SegundoThread();
        s.start();
        ss.start();
    }
}
```

En este caso se utiliza el constructor de la clase `Thread`:

```
public Thread(Runnable destino);
```

Metodología para la creación del *thread*:

- 1) La clase creada debe implementar el interface `Runnable`:

```
class PrimerThread implements Runnable {
```

- 2) La clase ha de crear un atributo (o variable local en el método `start()`) de la clase `Thread`:

```
Thread t;
```

- 3) Hay que redefinir el método `start()`:

- Instanciar el atributo de la clase `Thread` llamando a su constructor pasándole como parámetro la propia instancia de clase (`this`):

```
t = new Thread(this);
```

- Iniciar el *thread*:

```
t.start();
```

- 4) Redefinir el método `run()` tal y como se hace en la otra alternativa de creación de *threads* (mediante subclases de `Thread`).

Una vez declarada la clase que implementa la interface `Runnable`, ya puede ser instanciada e iniciada como cualquier *thread*. Es más, cualquier subclase descendiente de esta clase poseerá también las características propias de los *threads*, como ocurre en el ejemplo: `SegundoThread` es subclase de `PrimerThread`, y por lo tanto, ambas son `Thread`.

Threads.

Ambas formas de crear *threads* admiten un parámetro de tipo `String` que identifica al *thread* por su nombre:

```
public Thread(String nombre);
```

```
public Thread(Runnable destino, String nombre);
```

Para poder después averiguarlo mediante el método:

```
public final String getName();
```

Si no se ha asignado nombre a los *threads*, la clase `Thread` se los asigna por defecto como Thread-1, Thread-2, ...

6.4 Operaciones sobre threads.

6.4.1 `currentThread()`.

```
public static Thread currentThread();
```

Devuelve el *thread* que se está ejecutando acualmente. ↪ Ver ejemplo en el punto siguiente.

6.4.2 `isAlive()`.

```
public final boolean isAlive();
```

Devuelve `false` si el *thread* está en el estado **nuevo thread** o **muerto** y `true` en caso contrario.



```
class MiThread extends Thread {  
    public void run() {  
        try {  
            sleep(2000);  
        } catch (InterruptedException e) { }  
    }  
}
```

Threads.

```
class Vivo {
    public static void main(String args[]) {
        MiThread t = new MiThread();
        System.out.println("isAlive() antes de iniciar: "+
            t.isAlive());

        t.start();
        System.out.println("isAlive() en ejecución: "+
            t.isAlive());

        try {
            // Hace "dormir" al thread actual
            Thread.currentThread().sleep(3000);
            // de esta forma, le da tiempo a terminar al
            // thread t
        } catch (InterruptedException e) { }
        System.out.println("isAlive() después de ejecución: "+
            t.isAlive());
    }
}
```

La salida correspondiente al programa es:

```
Estado antes de iniciar: false
Estado en ejecución: true
Estado después de ejecución: false
```

6.4.3 *sleep()*.

- 1) **public static void sleep(long milisegundos) throws InterruptedException;**
 - 2) **public static void sleep(long milisegundos, int nanosegundos) throws InterruptedException;**
- 1) Hace que el *thread* actual pase del estado **ejecutable** a **dormido** y permanezca en dicho estado durante los milisegundos especificados como parámetro. Una vez que se ha cumplido el tiempo, el *thread* “despierta” y pasa automáticamente al estado de **ejecutable**.
 - 2) Acepta un parámetro más: un número entero de nanosegundos (0 ... 999999) que se sumaran a los milisegundos de espera.

En ambos casos se lanzará la excepción `InterruptedException` si se llama al método `interrupt()` del *thread*.

6.4.4 *suspend()*.

```
public final void suspend();
```

Llamando al método `suspend()` de un *thread*, el estado del mismo pasa de **ejecutable** a **suspendido** inmediatamente y sólo puede ser reactivado (pasado al estado **ejecutable**) llamando a su método `resume()`.

La llamada al método `resume()` de un *thread* que está dormido no tiene ningún efecto.

6.4.5 *wait()*.

El método `wait()` es heredado de la superclase `Object`, de la cuál heredan todos sus métodos todas las clases creadas en Java.

- 1) **public final void wait() throws InterruptedException;**
- 2) **public final void wait(long milisegundos) throws InterruptedException;**
- 3) **public final void wait(long milisegundos, int nanosegundos) throws InterruptedException**

- 1) Causa el paso al estado “**esperando**” del *thread* indefinidamente hasta que el *thread* sea notificado mediante `notify()` o `notifyAll()`.
- 2) El *thread* estará “**esperando**” hasta que sea notificado o expire el timeout (**milisegundos**).
- 3) El *thread* estará “**esperando**” hasta que sea notificado o expire el timeout (**milisegundos** + **nanosegundos**).

6.4.6 *yield()*.

```
public static void yield();
```

Threads.

Transfiere el control al siguiente **Thread** en el scheduler⁴¹ (con la misma prioridad que el actual) que se encuentre en estado **ejecutable**.



```
class UnThreadDosInstanciasAmables {
    public static void main(String args[] ) {
        SiNoThreadAmable s = new SiNoThreadAmable("SI");
        SiNoThreadAmable n = new SiNoThreadAmable("NO");
        s.start();
        n.start();
    }
}

class SiNoThreadAmable extends Thread {
    private String SiNo;
    static int Contador=0;
    public SiNoThreadAmable(String s) {
        super();
        SiNo=s;
    }
    public void run() {
        int i;
        for (i=1;i<=20; i++) {
            System.out.print(++Contador+": "+SiNo+" ");
            yield();
        }
    }
}
```

En este caso hay dos *threads* que se ejecutan concurrentemente. Para cada salida por pantalla, (`System.out.println(++Contador+": "SiNo+" ");`) el *thread* cede la CPU al otro *thread* mediante la llamada a `yield()`. Esto no significa que el otro *thread* continúe su ejecución hasta llegar a su propio `yield()`, ya que puede perder la CPU antes y por consiguiente no tienen por qué salir los Síes y Noes alternativamente, pero sí que está más repartida la CPU:

```
1:NO 2:SI 3:SI 4:NO 5:SI 6:NO 7:NO 8:SI 9:NO 10:SI
11:NO 12:NO 14:NO 13:SI 15:NO 16:SI 17:SI 18:NO 19:SI
20:NO 21:SI 22:NO 23:SI 24:SI 25:NO 26:SI 27:SI 28:NO
```

⁴¹ El *scheduler* es el proceso que determina en cada momento qué proceso o thread adquiere la CPU para su ejecución.

Threads.

29:NO 30:NO 31:SI 32:NO 33:SI 34:SI 35:SI 36:NO 37:SI
38:NO 39:SI 40:NO

6.4.7 `join()`.

public final void join() throws InterruptedException;

Hace que el *thread* que se está ejecutando actualmente pase al estado “esperando” indefinidamente hasta que muera el *thread* sobre el que se realiza el `join()`.



```
class MiThread extends Thread {
    public void run() {
        int i;
        for (i=1;i<=100; i++)
            System.out.print(i+" ");
    }
}

class Join1 {
    public static void main (String args[]) throws
    InterruptedException {
        MiThread t = new MiThread();
        t.start();
        t.join();
        System.out.println("El thread ha terminado");
    }
}
```

El *thread* de la clase `Join1` espera indefinidamente hasta que finalice el *thread* `t`.
La salida por pantalla es:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 96 97 98 99 100 El thread ha terminado
```

Threads.

Si no se hubiera realizado el `t.join()`, el *thread* de la clase `Join1` no habría esperado la finalización de `t` y la salida por pantalla habría sido otra:

```
1 2 3 El thread ha terminado
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100
```

Existen dos métodos `join()` más:

- **public final synchronized void join(long miliseg)**
throws InterruptedException;
- **public final synchronized void join(long miliseg, int nanoseg)**
throws InterruptedException;

En estos dos casos, el *thread* actual no espera indefinidamente sino que reinicia su ejecución en el instante en que se finalice el *thread* sobre el que se hace el `join()` o pase el tiempo especificado por los parámetros **miliseg** (milisegundos) y opcionalmente **nanoseg** (nanosegundos).

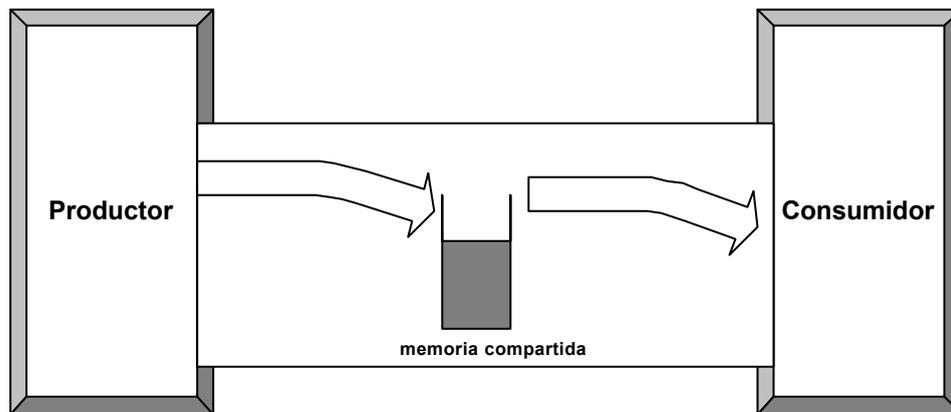
6.5 Threads y sincronismo.

Hasta el momento se ha estado tratando con *threads* asíncronos, cada uno de ellos se ejecutaba independientemente de los demás, que se ejecutaban concurrentemente. Existen, sin embargo, situaciones en las que es imprescindible sincronizar en cierta forma la ejecución de distintos *threads* que buscan un objetivo común, cooperando entre ellos para su correcto funcionamiento.

Threads.

6.5.1 El paradigma productor/consumidor.

El paradigma productor/consumidor consiste en la ejecución de un *thread* que produce un flujo de datos que deben ser consumidos por otro *thread*. Este flujo de datos puede ser almacenado en una zona de memoria compartida por ambos *threads*. En este caso, la sincronización es vital, ya que de lo contrario, si el productor genera los datos más deprisa de lo que el consumidor es capaz de tratarlos, puede producirse un desbordamiento de la memoria asignada al efecto; por otra parte, si el consumidor es más rápido que el productor, puede encontrarse con que no hay datos que consumir o que se consuman más de una vez (si no elimina los datos consumidos, sino que los elimina el productor).

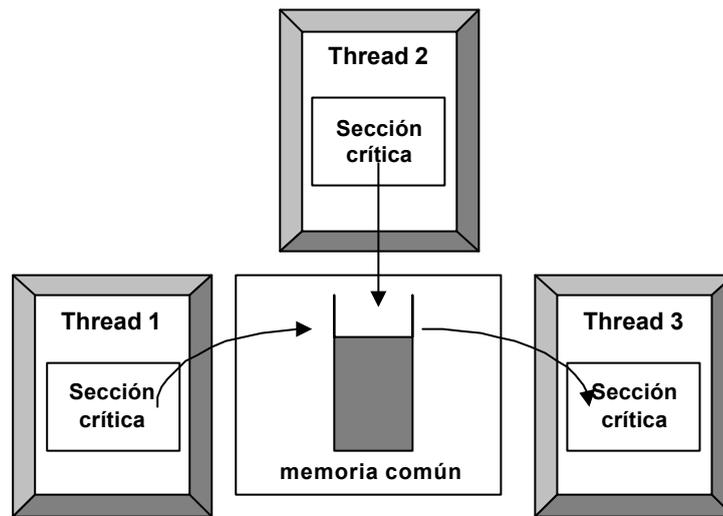


En este caso, debe existir un mecanismo que “frene” al productor o al consumidor en los casos necesarios.

Threads.

6.5.2 Sección crítica.

Se llama **sección crítica** a los segmentos de código dentro de un programa que acceden a zonas de memoria comunes desde distintos *threads* que se ejecutan concurrentemente



En Java, las secciones críticas se marcan con la palabra reservada `synchronized`. Aunque está permitido marcar bloques de código más pequeños que un método como `synchronized`, para seguir una buena metodología de programación, es preferible hacerlo a nivel de método.



```
class Contador {
    private long valor=0;
    public void incrementa() {
        long aux;
        aux=valor;
        aux++;
        valor=aux;
    }
    public long getValor() {
        return valor;
    }
}
```

Threads.

```
class Contable extends Thread {
    Contador contador;
    public Contable (Contador c) {
        contador=c;
    }
    public void run () {
        int i;
        long aux;
        for (i=1;i<=100000; i++) {
            contador.incrementa();
        }
        System.out.println("Contado hasta ahora: "+
            contador.getValor());
    }
}

class ThreadSinSync {
    public static void main(String arg[]) {
        Contable c1 , c2 ;
        Contador c = new Contador();
        c1 = new Contable(c);
        c2 = new Contable(c);
        c1.start();
        c2.start();
    }
}
```

Para realizar el incremento en el método `incrementa()`, a efectos didácticos, se ha empleado una variable auxiliar intermedia (`aux`), aunque el efecto de haber codificado `valor++` habría sido similar.

En este caso, la sección crítica es la línea: `contador.incrementa();` ya que desde esta instrucción se accede a un objeto (`contador`) que es compartido por ambos *threads* (`c1` y `c2`).

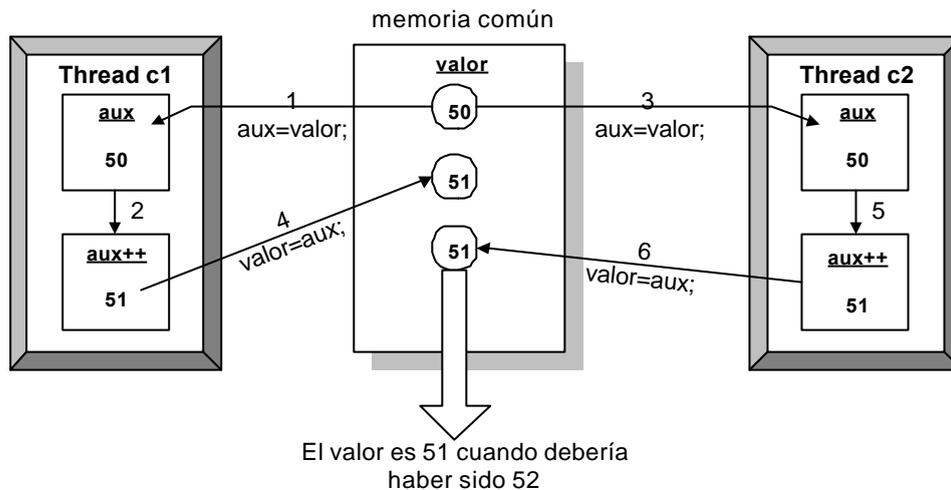
La salida por pantalla es:

```
Contado hasta ahora: 124739
Contado hasta ahora: 158049
```

cuando debería haber sido: en la primera línea un número comprendido entre 100000 y 200000; y en la segunda línea el valor 200000.

Threads.

Lo que ha ocurrido en realidad para que se produzca este resultado equivocado es lo siguiente:

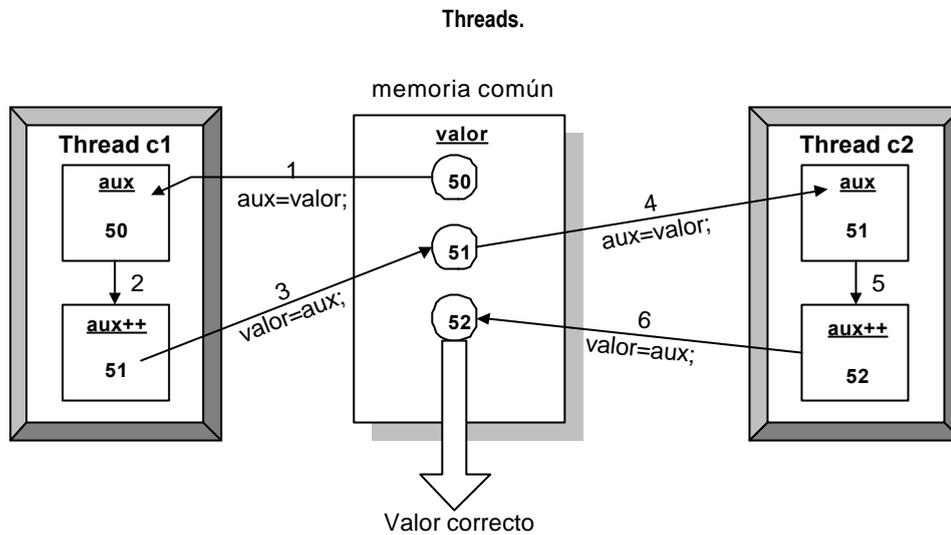


Evidentemente, esto mismo ha ocurrido un gran número de veces durante la ejecución de los *threads*.

Simplemente cambiando la línea de código:

```
public void incrementa() {  
    por  
    public synchronized void incrementa() {
```

se habría conseguido bloquear el método `incrementa`, de forma que no pudieran ejecutarlo simultáneamente dos *threads* sobre el mismo objeto (contador).



La salida del programa habría sido (correcta):

```
Contado hasta ahora: 186837
Contado hasta ahora: 200000
```

6.5.3 Monitores.

En Java, cada objeto que posee un método `synchronized` posee un único monitor para ese objeto.

Cuando un *thread* está ejecutando un método `synchronized` de un objeto, se convierte en el propietario del monitor, evitando que cualquier otro *thread* ejecute ningún otro método `synchronized` sobre ese mismo objeto. Esto no impide que el propietario del monitor ejecute otro método `synchronized` de ese mismo objeto, adquiriendo de nuevo el monitor (llamada a un método `synchronized` desde otro método `synchronized`).



```
class Caja {
    private int valor;
    public synchronized void meter(int nv) {
        valor = nv;
        System.out.println("metido el valor: "+valor);
    }
    public synchronized void sacar() {
        System.out.println("sacado el valor: "+valor);
    }
}
```

Threads.

```
        valor=0;
    }
}
class Productor extends Thread {
    Caja c;
    public Productor(Caja nc) {
        c = nc;
    }
    public void run() {
        int i;
        for (i=1; i<=10; i++)
            c.meter(i);
    }
}

class Consumidor extends Thread {
    Caja c;
    public Consumidor(Caja nc) {
        c = nc;
    }
    public void run() {
        int i;
        for (i=1; i<=10; i++)
            c.sacar();
    }
}

class ProdCons {
    public static void main(String argum[]) {
        Caja cj = new Caja();
        Productor p = new Productor(cj);
        Consumidor c = new Consumidor(cj);
        p.start();
        c.start();
    }
}
```

En este ejemplo existen dos *threads* que se comportan siguiendo el paradigma productor/consumidor.

Existe un objeto de la clase Caja que es capaz de almacenar un único número entero. Sobre esta caja se pueden realizar dos tipos de operaciones:

- **meter(int)**: Almacena el entero que se le pasa como parámetro y muestra el valor en pantalla.
- **sacar()**: Muestra en pantalla el valor almacenado y pone ese valor a cero.

Threads.

Tanto el *thread* productor como el consumidor comparten el mismo objeto de la clase *Caja*. Para asegurar la exclusión mutua en la zona crítica (la que accede a *valor*), se declaran los métodos *meter* y *sacar* como *synchronized*. Pero esto no es suficiente para que el programa funcione correctamente, ya que el productor puede almacenar varios valores antes de que el consumidor extraiga el valor o, también, que el consumidor intente sacar varios valores consecutivamente, por lo que la salida por pantalla podría ser la siguiente:

```
metido el valor: 1
sacado el valor: 1
sacado el valor: 0
sacado el valor: 0
metido el valor: 2
metido el valor: 3
sacado el valor: 3
sacado el valor: 0
sacado el valor: 0
metido el valor: 4
metido el valor: 5
sacado el valor: 5
sacado el valor: 0
metido el valor: 6
metido el valor: 7
metido el valor: 8
sacado el valor: 8
sacado el valor: 0
metido el valor: 9
metido el valor: 10
```

De alguna forma habría que asegurar que no se meta ningún valor si la caja está llena y que no se saque ningún valor si la caja está vacía.



```
class Caja {
    private int valor;
    private boolean disponible=false;
    public synchronized void meter(int nv) {
        if (!disponible) {
            valor = nv;
        }
    }
}
```

Threads.

```
        disponible=true;
        System.out.println("metido el valor: "+valor);
    }
    public synchronized void sacar() {
        if (disponible) {
            System.out.println("sacado el valor: "+valor);
            valor=0;
            disponible=false;
        }
    }
}
class Productor extends Thread {
    Caja c;
    public Productor(Caja nc) {
        c = nc;
    }
    public void run() {
        int i;
        for (i=1; i<=10; i++)
            c.meter(i);
    }
}

class Consumidor extends Thread {
    Caja c;
    public Consumidor(Caja nc) {
        c = nc;
    }
    public void run() {
        int i;
        for (i=1; i<=10; i++)
            c.sacar();
    }
}

class ProdCons2 {
    public static void main(String argum[]) {
        Caja cj = new Caja();
        Productor p = new Productor(cj);
        Consumidor c = new Consumidor(cj);
        p.start();
        c.start();
    }
}
```

Salida por pantalla:

Threads.

```
metido el valor: 1
sacado el valor: 1
metido el valor: 2
```

Después de sacar el valor 1, el consumidor ha seguido su ejecución en el bucle y como no se mete ningún valor por el consumidor, termina su ejecución, el productor introduce el siguiente valor (el 2), y sigue su ejecución hasta terminar el bucle; como el consumidor ya ha terminado su ejecución no se pueden introducir más valores y el programa termina.

Lo que hace falta es un mecanismo que produzca la espera del productor si la caja tiene algún valor (disponible) y otro que frene al consumidor si la caja está vacía (no disponible):



```
class Caja {
    private int valor;
    private boolean disponible=false;
    public synchronized void meter(int nv) {
        if (disponible) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        valor = nv;
        disponible=true;
        System.out.println("metido el valor: "+valor);
        notify();
    }
    public synchronized void sacar() {
        if (!disponible) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        System.out.println("sacado el valor: "+valor);
        valor=0;
        disponible=false;
        notify();
    }
}
class Productor extends Thread {
    Caja c;
    public Productor(Caja nc) {
        c = nc;
    }
}
```

Threads.

```
    }  
    public void run() {  
        int i;  
        for (i=1; i<=10; i++)  
            c.meter(i);  
    }  
}  
  
class Consumidor extends Thread {  
    Caja c;  
    public Consumidor(Caja nc) {  
        c = nc;  
    }  
    public void run() {  
        int i;  
        for (i=1; i<=10; i++)  
            c.sacar();  
    }  
}  
  
class ProdCons3 {  
    public static void main(String argum[]) {  
        Caja cj = new Caja();  
        Productor p = new Productor(cj);  
        Consumidor c = new Consumidor(cj);  
        p.start();  
        c.start();  
    }  
}
```

Produce el resultado deseado:

```
metido el valor: 1  
sacado el valor: 1  
metido el valor: 2  
sacado el valor: 2  
metido el valor: 3  
sacado el valor: 3  
metido el valor: 4  
sacado el valor: 4  
metido el valor: 5  
sacado el valor: 5  
metido el valor: 6  
sacado el valor: 6  
metido el valor: 7  
sacado el valor: 7
```

Threads.

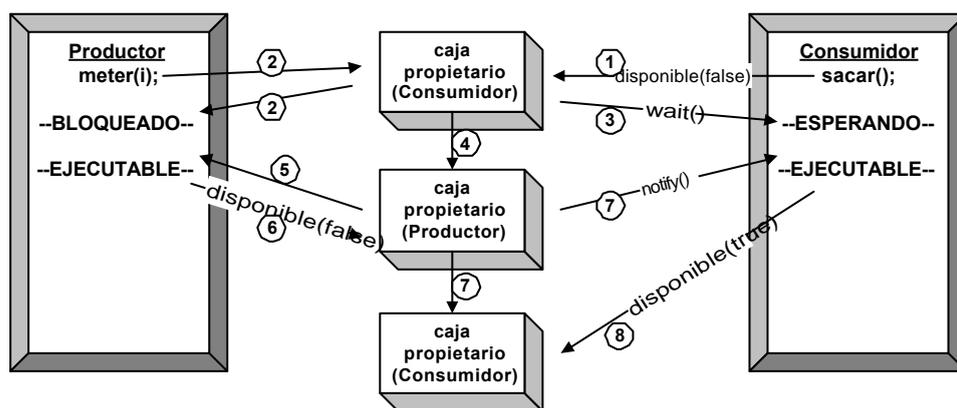
```
metido el valor: 8
sacado el valor: 8
metido el valor: 9
sacado el valor: 9
metido el valor: 10
sacado el valor: 10
```

- Si el primer *thread* en adquirir el monitor del objeto Caja es el Productor al ejecutar el método `meter(i)`:

la variable `disponible` contendrá el valor `false` y no se ejecutará `wait()`. Realiza la operación correspondiente y ejecuta el método `notify()` que no tiene ningún efecto.

- Si el primer *thread* en adquirir el monitor del objeto `Caja` es el `Consumidor` al ejecutar el método `sacar()`:

La variable `disponible` contendrá el valor `false` y por lo tanto, se ejecuta el método `wait()` el método `wait` hace que el *thread* que actualmente posee el monitor lo abandone el monitor es tomado por el otro *thread* al llamar a `meter(i)` como `disponible` sigue siendo `false`, no se ejecuta el `wait()`, se completa el método, se cambia el valor de `disponible` a `true` y se llama al método `notify()` el método `notify()` hace que el *thread* que está en estado de espera pase a ejecutable y tome el monitor del objeto.



6.6 *Prioridad de un thread.*

Hasta ahora se ha supuesto que todos los *threads* se repartían el tiempo de la CPU por igual, pero esto no es generalmente así. Se repartirán la CPU por igual sólo si la prioridad de los distintos *thread* es la misma.

En todos los ejemplos que se han visto anteriormente los *threads* tenían la misma prioridad porque cuando se crea un *thread*, éste hereda la prioridad del *thread* que lo creó, sin embargo esta prioridad puede ser modificada en cualquier momento llamando al método `setPriority()`. Para conocer la prioridad de un *thread* se utiliza el método `getPriority()`.

public final static int MAX_PRIORITY = 10; prioridad máxima de un *thread*

public final static int MIN_PRIORITY = 1; prioridad mínima de un *thread*

public final static int NORM_PRIORITY = 5; prioridad normal (por defecto) de un *thread*

public final void setPriority(int newPriority);

public final int getPriority();

El algoritmo de planificación (*scheduling*) es el siguiente: la CPU se asigna al *thread* en estado ejecutable de mayor prioridad y si hay varios, se reparten el tiempo mediante *round-robin*. Además este algoritmo es exclusivo, es decir, si se está ejecutando un *thread* y en ese momento pasa a estado ejecutable un *thread* de mayor prioridad, se le asigna directamente la CPU. Este algoritmo tiene un peligro: que un *thread* nunca se ejecute porque siempre existan *threads* de mayor prioridad. Java evita esto asignando la CPU también a *threads* de menor prioridad aunque cada mucho menos tiempo que a los *threads* de mayor prioridad. De todas formas, no debe confiarse la “no inanición” a la bondad del Java.



```
class ThreadConPrioridad extends Thread {
    String nombre;
    public ThreadConPrioridad(String n , int pri) {
        super();
    }
}
```

Threads.

```
        nombre=n;
        setPriority(pri);
    }
    public void run() {
        int i;
        for(i=1;i<100;i++) {
            System.out.print(nombre);
            yield();
        }
    }
}

class Prioridad1 {
    public static void main(String args[]) {
        ThreadConPrioridad t1 = new ThreadConPrioridad("1",5);
        ThreadConPrioridad t2 = new ThreadConPrioridad("2",5);
        t1.start();
        t2.start();
    }
}
```

Salida por pantalla:

```
221211122122212121212121112221212121212121212121212121
212121212121212121212121221212121212121121212121221212
1212121212121212112121212121212121212121221212211212
121212121121212121212121212121212121221111
```

En este caso, los *threads* *t1* y *t2* tienen la misma prioridad (5), por lo que al llamar al método `yield()` se cede la ejecución al otro *thread*.



Si las prioridades hubieran sido otras. Por ejemplo:

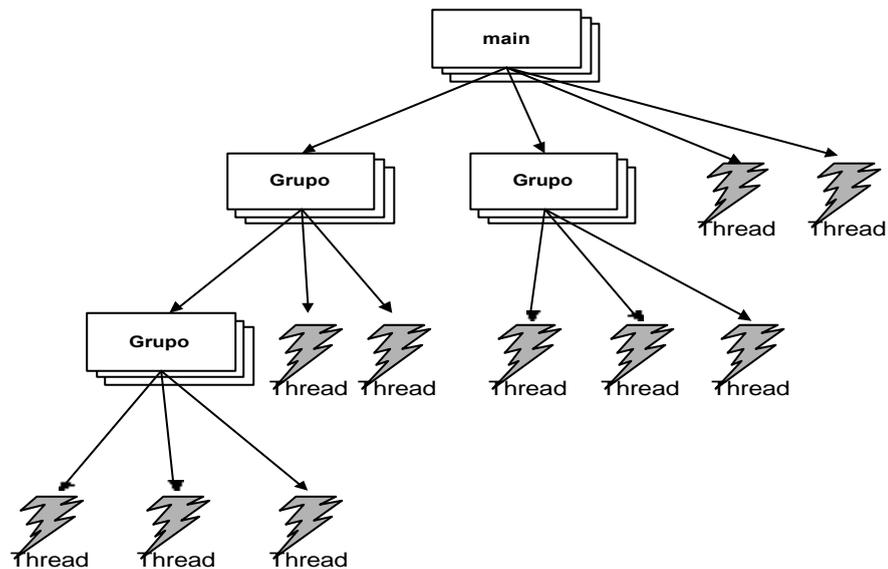
```
ThreadConPrioridad t1 = new ThreadConPrioridad("1",5);
ThreadConPrioridad t2 = new ThreadConPrioridad("2",4);
```


6.8 Grupos de threads.

Todo *thread* creado en Java pertenece a un Grupo de *threads*. Los grupos de *threads* sirven para manejar como una unidad a un conjunto de *threads*; por ejemplo: iniciar o detener varios *threads* mediante una sola operación. Hay que señalar que los grupos de *threads* no añaden ningún tipo de funcionalidad, únicamente sirven para que sea más cómodo el manejo muchos *threads* en un programa. Para ello se utiliza la clase `ThreadGroup` que pertenece al paquete `java.lang`.

Cuando se inicia la ejecución de un programa, Java crea un grupo de *threads* llamado `main`. A no ser que se especifique lo contrario, cuando se crea un *thread*, se añade al grupo al que pertenece el *thread* que lo creó. Esa es la razón por la cual todos los *threads* creados hasta el momento pertenecen al grupo `main`.

Un grupo de *threads* puede contener *threads*, pero también grupos, y todos ellos pertenecen a un grupo raíz: `main`.



6.8.1 Creación de grupos de threads.

Mediante llamadas al constructor:

Threads.

1) **public ThreadGroup(String nombre);**

2) **public ThreadGroup(ThreadGroup padre, String nombre);**

- 1) Crea un grupo cuyo nombre se pasa como parámetro. Este grupo pertenecerá (subgrupo) al grupo actual (al que pertenece el *thread* que realiza la llamada al constructor).

Ejemplo: `ThreadGroup tg = new ThreadGroup("MiGrupo");`

- 2) Crea un grupo cuyo nombre se pasa como parámetro. Este grupo pertenecerá al grupo que se pasa como parámetro (padre).

Ejemplo: `ThreadGroup tg2 = new ThreadGroup(tg ,
"MiGrupo2");`

Es posible obtener el grupo al que pertenece un *thread* mediante la llamada al método `getThreadGroup()` de la clase `Thread`:



```
class Grupo1 {  
    public static void main(String args[]) {  
        ThreadGroup tg =  
            Thread.currentThread().getThreadGroup();  
        System.out.println(tg.getName());  
    }  
}
```

El método `getName()` devuelve un `String` que contiene el nombre del grupo.

La salida por pantalla, evidentemente es:

```
main
```

Threads.

6.8.2 Operaciones sobre threads agrupados.

6.8.2.1 Operaciones sobre threads.

6.8.2.1.1 Asignación de un thread a un grupo.

Como se ha visto, los *threads* se insertan por defecto en el mismo grupo que el *thread* actual (el *thread* que lo crea) a no ser que se indique explícitamente a qué grupo debe pertenecer. Esto se realiza en la llamada al constructor del *thread*:

- `public Thread(ThreadGroup grupo, String nombreDelThread);`
- `public Thread(ThreadGroup grupo, Runnable destino, String nombreDelThread);`

6.8.2.1.2 Operaciones sobre un conjunto de threads.

`public final void suspend();` Pasan al estado de “suspendido” todos los *threads* descendientes de este grupo (todos los *threads* pertenecientes a este grupo) y se llama al método `suspend()` de todos los grupos pertenecientes a este grupo (valga la redundancia).

`public final void resume();` Pasan al estado de “ejecutable” todos los *threads* descendientes de este grupo que estaban en estado “suspendido”.

`public final void stop();` Pasan al estado “muerto” todos los *threads* descendientes de este grupo.

6.8.2.2 Operaciones sobre grupos.

`public final void setDaemon(boolean daemon);` Convierte al grupo en *daemon* si el parámetro es `true`⁴³. Un grupo *daemon*

⁴³ Atención: la llamada al método `setDaemon()` de un grupo de threads no significa que los threads descendientes del grupo se conviertan en daemons.

Threads.

es destruido automáticamente cuando no contiene ningún *thread* o grupo.

public final boolean isDaemon(); Devuelve `true` si el grupo es un *daemon* y `false` en caso contrario.

public final void setMaxPriority(int pri); Establece la máxima prioridad que puede tener un *thread* descendiente de este grupo, aunque los *threads* descendientes que ya pertenecían al grupo antes de llamar al método `setMaxPriority()` pueden conservar su prioridad mayor que el parámetro (`pri`).

public final void destroy() throws IllegalStateException; Destruye el grupo (pero no los *threads* descendientes). Este grupo debe estar vacío: no debe contener *threads* activos, de lo contrario se genera una excepción de la clase `IllegalThreadStateException`. Si el grupo no está vacío, hay que llamar antes al método `stop()` del grupo⁴⁴ para que todos los *threads* descendientes pasen al estado “muerto”.

public final boolean parentOf(ThreadGroup g); Devuelve `true` si este grupo es padre (o es el mismo) que el grupo que se pasa como parámetro (`g`) y `false` en caso contrario.

6.8.2.3 Obtención del contenido de un grupo.

1) **public int enumerate(Thread lista[]);**

2) **public int enumerate(Thread lista[], boolean recurr);**

3) **public int enumerate(ThreadGroup lista[]);**

4) **public int enumerate(ThreadGroup lista[], boolean recurr);**

⁴⁴ Desconozco la razón pero no es lo mismo llamar al método `stop()` del grupo que a cada método `stop()` de cada uno de los *threads* descendientes. En teoría debería poderse llamar al método `destroy()` del grupo, pero en el segundo supuesto no funciona, se genera una `IllegalThreadStateException`.

Threads.

El método `enumerate` puede obtener un vector con los *threads* pertenecientes al grupo (los casos 1 y 2) o un vector con los grupos pertenecientes a un grupo (casos 3 y 4) se almacena en el parámetro `lista[]`⁴⁵.

Por defecto, se obtienen listas de *threads* (y grupos) recursivas, añadiendo los *threads* o grupos del grupo sobre el que se realiza la llamada y de los grupos descendientes del mismo. Para obtener únicamente los *threads* pertenecientes al grupo en cuestión es necesario pasar un nuevo parámetro de tipo `boolean` (`recur`) con el valor `false`.

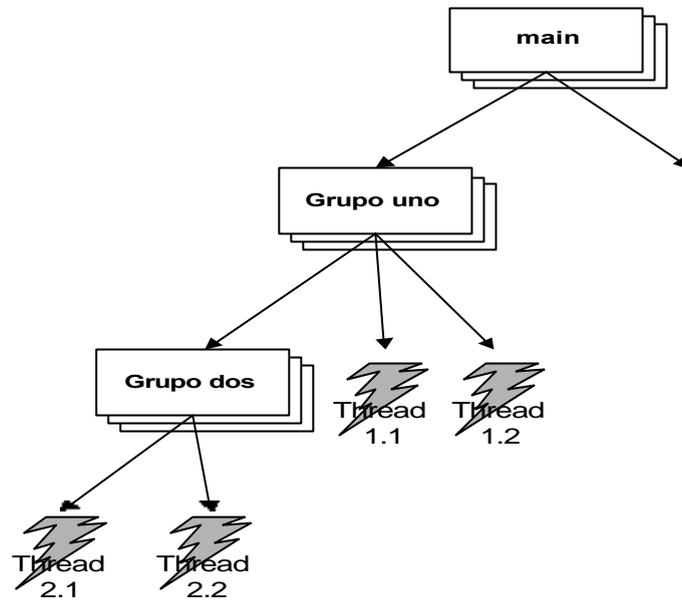


```
class MiThread extends Thread {
    public MiThread(ThreadGroup tg, String nom) {
        super(tg,nom);
    }
    public void run() {
        System.out.println("Ejecutado "+getName());
    }
}

class Grupo2 {
    public static void main(String args[])
        throws InterruptedException {
        ThreadGroup tg = new ThreadGroup("grupo uno");
        ThreadGroup tg2 = new ThreadGroup(tg,"grupo dos");
        MiThread t1 = new MiThread(tg,"Thread 1.1");
        MiThread t12 = new MiThread(tg,"Thread 1.2");
        MiThread t21 = new MiThread(tg2,"Thread 2.1");
        MiThread t22 = new MiThread(tg2,"Thread 2.2");
        Thread t[] = new Thread[4];
        int i;
        tg.enumerate(t);
        for (i=0;i<t.length;i++)
            System.out.println(t[i]);
    }
}
```

⁴⁵ Recordemos que un vector en Java es un objeto y que todos los objetos pasados como parámetro a un método lo hacen por referencia, es decir, que lo que en realidad se pasa es el puntero al objeto, por lo que puede alterarse su contenido en el cuerpo del método.

Threads.



En este caso, la salida por pantalla es la siguiente:

```
Thread[Thread 1.1,5,grupo uno]
Thread[Thread 1.2,5,grupo uno]
Thread[Thread 2.1,5,grupo dos]
Thread[Thread 2.2,5,grupo dos]
```

Si en lugar de `enumerate(t)` se hubiera utilizado la fórmula `enumerate(t, false)`, la salida por pantalla habría sido:

```
Thread[Thread 1.1,5,grupo uno]
Thread[Thread 1.2,5,grupo uno]
null
null
```

6.9 CUESTIONES

1. Un programa que tiene varios *threads* ¿disponen los diferentes *threads* de acceso a la misma imagen de memoria, o cada uno de ellos tiene un contexto privado?
2. ¿Qué método se debe redefinir en una clase para que incluya las tareas de una nueva clase que extiende la clase `Thread`?
3. Si se dispone de un programa con un método `main()` y un *thread* adicional en ejecución. ¿Se ejecuta el programa principal mientras está en ejecución el *thread* o espera el programa principal a que finalice ese *thread* antes de continuar su ejecución?
4. ¿Qué método inicia la ejecución de un *thread*?
5. ¿Qué condiciones hacen que finalice un *thread*?
6. ¿Cómo podemos averiguar si un determinado *thread* está aún activo?
7. ¿Qué diferencia hay entre los métodos `sleep()` y `suspend()`?
8. ¿Puede un *thread* modificar la política de planificación? ¿De qué manera?
9. ¿Cómo asegurar que un método de un *thread* tenga acceso exclusivo a un atributo de la clase a la que pertenece ese *thread*?
10. ¿Cómo se puede sincronizar una acción en un *thread* con la finalización de otro?

7. COMUNICACIONES TCP/IP

7.1 Introducción.

Las comunicaciones entre diferentes sistemas se realizan típicamente a través de una red, que usualmente denominamos local cuando la distancia es pequeña y extensa cuando se trata de equipos muy alejados (se mantiene intencionadamente la ambigüedad en los conceptos de distancia dada la gran variedad de redes existente).

Tanto si usamos un tipo de red como otro existen multitud de protocolos para el intercambio de información. Java incorpora un soporte nativo para la familia de protocolos que se conoce con el nombre genérico de TCP/IP, que son los protocolos empleados en Internet.

Como se verá más adelante esta denominación hace referencia a dos de los protocolos utilizados, pero en realidad hay más protocolos disponibles para el programador Java.

Básicamente se va a disponer de dos tipos de servicio claramente diferenciados:

- Un servicio que denominamos **sin conexión**, que se parece al funcionamiento de un servicio de correo o de mensajería, en el que origen y destino intercambian uno o más bloques de información sin realizar una conexión previa y donde no se dispone de un control de secuencia (el orden de entrega puede no corresponder al de envío) ni tampoco control de error (algunos de estos bloques de información pueden perderse sin que se reciba aviso alguno).
- Otro servicio **orientado a la conexión**, en el que disponemos de un *stream* que nos asegura la entrega de información ordenada y fiable, sin que se pueda perder información. En este servicio, antes de enviar información se realiza un proceso de conexión, similar al funcionamiento de un teléfono (donde, antes de hablar, tenemos que completar el proceso de llamada).

Comunicaciones TCP/IP

Cada uno de estos tipos de servicio será de aplicación para tareas diferentes y posee un conjunto de características que los hace aptos para diferentes usos.

7.2 *Arquitectura Cliente / Servidor.*

Mediante la arquitectura cliente-servidor se definen servidores pasivos y clientes activos que crean conexiones hacia los servidores. Se realiza un diseño de aplicaciones asimétricas entre cliente y servidor. Es decir, son distintas las aplicaciones que se ejecutan en el servidor a las que se ejecutan en el cliente.

La comunicación entre un cliente y un servidor se puede realizar de dos formas distintas mediante conexiones (servicio orientado a conexión) o datagramas (servicio sin conexión).

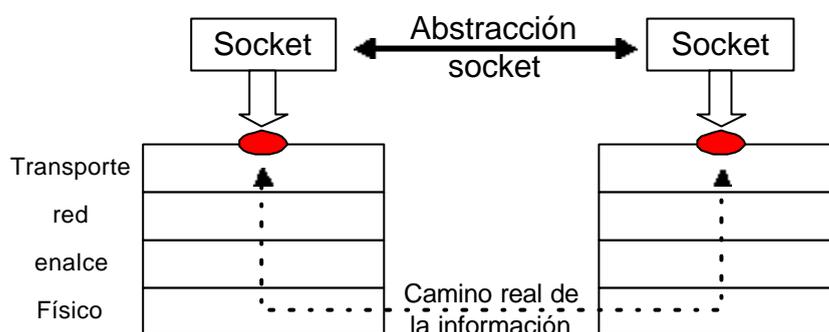
- Protocolo orientado a conexión: se basa en una conexión establecida en la que queda fijado el destino de la conexión desde el momento de abrirla. Esta conexión permanecerá hasta que se cierre, como ocurre con una llamada telefónica. Este protocolo en TCP/IP se llama TCP (*Transmission Control Protocol*).
- Protocolo sin conexión (datagramas): por el contrario envía mensajes individuales con el destino grabado en cada uno de ellos, como en una comunicación a través de correo postal, donde no son necesarias las fases de establecimiento y liberación de conexión alguna. Este protocolo en TCP/IP se llama UDP (*User Datagram Protocol*).

En este modelo de aplicaciones, el servidor suele tener un papel pasivo, respondiendo únicamente a las peticiones de los clientes, mientras que estos últimos son los que interactúan con los usuarios (y disponen de *interface* de usuario).

La comunicación entre cliente y servidor se efectúa través de la red que los une, empleando, además de un protocolo de transporte (TCP o UDP), un protocolo de aplicación que varía de unas aplicaciones a otras y que, como indica su nombre, está codificado en la propia aplicación (mientras que el protocolo de transporte está incorporado en el sistema operativo y es accedido desde los programas Java a través de las clases que se estudiarán en este capítulo).

7.3 La abstracción socket.

Un *socket*⁴⁶ es el extremo de un enlace de comunicación bidireccional entre dos programas que se comunican a través de la red.



En la arquitectura TCP/IP, los ordenadores están identificados por una dirección de red (IP), de la misma forma que en el sistema telefónico cada terminal dispone de un identificador (número de teléfono) único. Para comunicarnos con otro ordenador necesitamos conocer (o averiguar) su dirección IP (de 32 bits).

Sin embargo, este esquema de direccionamiento no es suficiente si deseamos poder tener varias comunicaciones independientes desde cada ordenador⁴⁷. Por este motivo, los protocolos de transporte extienden este esquema con un nuevo identificador, denominado puerto (*port*). Este identificador (de 16 bits) permite incrementar las posibilidades de comunicación y mantener varias conexiones simultáneamente.

Una dirección del nivel de transporte (TCP o UDP) viene especificada por una dirección del nivel de red (IP) más un número de puerto.

Dirección IP (4 números entre 0 y 255 separados por puntos—8 bits)

⁴⁶ Se mantendrá la denominación inglesa *socket* a lo largo del texto.

⁴⁷ De todos es conocido que un teléfono permite una comunicación nada más. Si el teléfono está ocupado, obtendremos una señal de comunicando. Si es nuestro teléfono el que está en uso por otra persona, tendremos que esperar a que termine antes de poder iniciar otra conexión.

Comunicaciones TCP/IP

Puerto (número entre 0 y 65.535)

Ejemplo: 158.42.53.1:1270 (IP=158.42.53.1, puerto=1270)

El paquete **java.net** proporciona tres clases: `Socket`, `ServerSocket` y `DatagramSocket`.

`Socket` Implementa un extremo de la conexión (TCP) a través de la cual se realiza la comunicación.

`ServerSocket` Implementa el extremo Servidor de la conexión (TCP) en la cual se esperan las conexiones de clientes.

`DatagramSocket` Implementa tanto el extremo servidor como el cliente de UDP.

El sistema operativo creará un punto de conexión en el protocolo de nivel 4 (nivel de transporte) correspondiente, en el caso de internet (arquitectura TCP/IP) las conexiones serán soportadas por el protocolo TCP (si el tipo es `Socket` fiable y los datagramas (`DatagramSocket`) por el protocolo UDP no fiable.

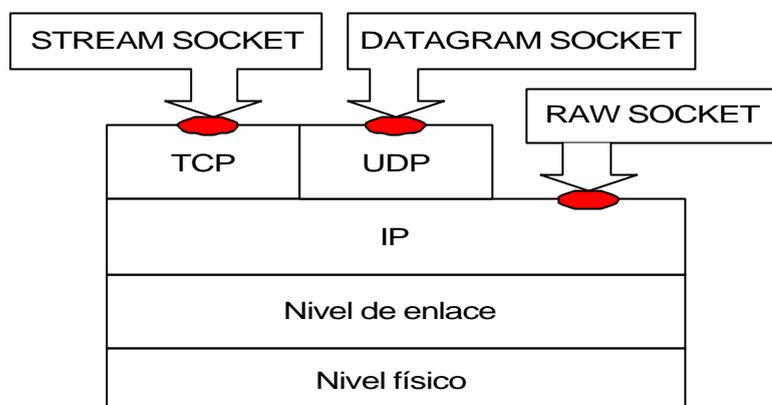
Concepto de fiabilidad:

1. No se perderán paquetes.
2. No llegarán paquetes duplicados.
3. El orden de recepción será el mismo que el de emisión.

Si se emplea un protocolo fiable, el programa de aplicación quedará liberado de la responsabilidad de gestionar reordenaciones, pérdidas y duplicados. A cambio, el software de protocolos interno necesitará un mayor tiempo de proceso.

Existe otro tipo de *socket*, el *socket* crudo **RAW SOCKET** que se salta el nivel de transporte y accede directamente al nivel 3 (nivel de red): En el caso de Internet IP. Este tipo de *socket* se reserva a usuarios experimentados ya que suele ser utilizado por programas con privilegios para la definición de protocolos de bajo nivel.

Comunicaciones TCP/IP



7.4 Servicio sin conexión (*Datagram Socket*).

Se trata del mecanismo más simple, puesto que el servicio sin conexión tan sólo nos ofrece un mero envío (y se verá si con suerte llega) de datos.

Puesto que no existe aquí la conexión no hay proceso previo alguno antes de enviar información. Para poder comunicar con otro proceso lo único que hay que hacer es crear el *socket* y utilizar sus métodos para el envío y recepción de información.

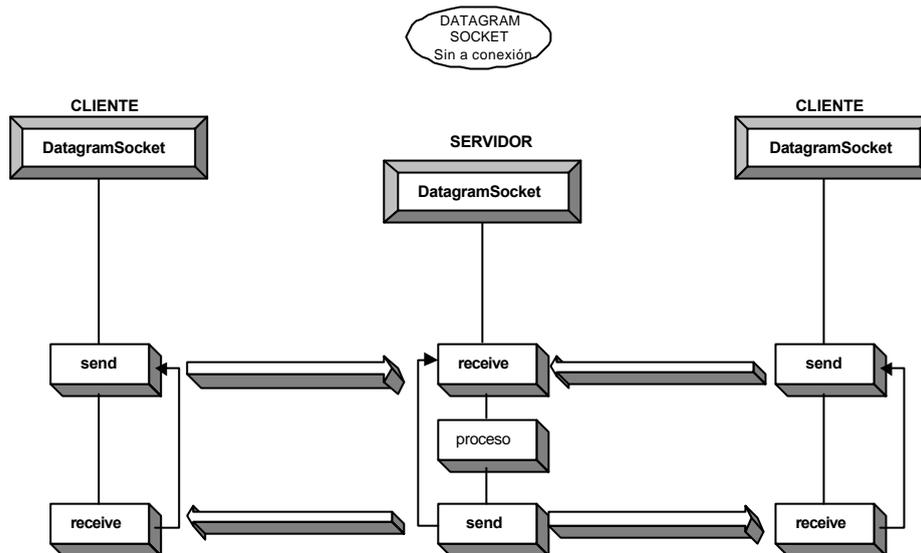
7.4.1 Creación de un *DatagramSocket*.

Los constructores disponibles (desde la versión 1.0 del JDK) son:

- `DatagramSocket()` Construye un *socket* para datagramas y lo “conecta” al primer puerto disponible.
- `DatagramSocket(int)` Igual que el anterior, pero además permite especificar el número de puerto asociado.
- `DatagramSocket(int, InetAddress)` Este constructor nos permite especificar además del puerto, la dirección a la que vamos a conectar.

Comunicaciones TCP/IP

La siguiente figura ilustra cómo se produce la comunicación entre clientes y servidor empleando un `DatagramSocket`.



En el siguiente ejemplo se han combinado muchos de los elementos de temas anteriores, incluyendo *threads*, *strings* y entrada-salida, junto con los servicios básicos sin conexión que utilizan el protocolo UDP.

Básicamente se trata de un programa que incluye un servidor de eco UDP que se ejecuta en una *thread* independiente y un cliente de eco UDP que se ejecuta en la *thread* del procedimiento principal `main()` de la clase `EcoUdp`.

La finalidad de utilizar *threads* en este ejemplo es permitir verificar la operación empleando un solo ordenador. Sin embargo, puesto que se trata de *threads* que se ejecutan concurrentemente, el servidor también responderá a peticiones de eco UDP solicitadas desde otros ordenadores. Mientras que el cliente implementado sólo envía peticiones locales, esto es, a la dirección local, que se obtiene con el método `getLocalHost()` de la clase `InetAddress`.

Es interesante destacar el uso de la clase `DatagramPacket` para albergar los paquetes que se intercambian entre cliente y servidor. El constructor de esta clase permite inicializar todos los parámetros como la dirección y puerto de destino, además de los datos a enviar. También proporciona los métodos necesarios para obtener esa información cuando se recibe un paquete (dirección y puerto del

Comunicaciones TCP/IP

origen, así como los datos y su longitud) por medio de los métodos `getData()`, `getLength()`, `getAddress()` y `getPort()`.

Atención: Este ejemplo funcionará sólo en Windows 95, puesto que Windows NT, Solaris o Linux emplean el puerto 7 UDP para el servidor de eco UDP estándar en estos sistemas operativos, por lo que para usar el presente ejemplo habrá que cambiar a un puerto disponible, es decir, alguno por encima del 1023.



```
import java.net.*;
import java.io.*;
class ServidorEcoUdp extends Thread {
// el servidor correra en un thread
// para poder lanzarlo con el cliente
public void run(){
//aquí es donde se hace el trabajo (sólo una instancia)
try{ // pueden producirse excepciones
byte[] datos,buffer=new byte[256];
DatagramSocket s=new DatagramSocket(7); // puerto de eco
DatagramPacket p;
int puerto,longitud;
InetAddress dir;
String mensaje;
System.out.println(
"Comienza la ejecucion del servidor ...");
while (true) {
p=new DatagramPacket(buffer,256);
s.receive(p); //espero un datagrama
datos=p.getData(); //obtengo datos
puerto=p.getPort(); //obtengo puerto origen
dir=p.getAddress(); //obtengo dir IP
longitud=p.getLength(); //longitud del mensaje
mensaje=new String(buffer,0,longitud); //texto del
//mensaje
System.out.println("Eco recibido:"+dir+":"+puerto+
" > "+mensaje);
// construyo contestacion
p=new DatagramPacket(buffer,longitud,dir,puerto);
s.send(p); //devuelvo contestacion
}
}
catch(SocketException e) {System.out.println(e.toString());}
catch(IOException e) {System.out.println(e.toString());}
}
}

class EcoUdp {
public static void clienteEcoUdp() throws SocketException,
IOException, UnknownHostException {
```

Comunicaciones TCP/IP

```
//en este método hemos optado por no tratar las excepciones
String mensaje;
InputStreamReader isr = new
    InputStreamReader(System.in);
BufferedReader teclado = new BufferedReader(isr);
// puerto origen: el que encuentre libre
DatagramSocket s=new DatagramSocket();
while ((mensaje=teclado.readLine())!=null) {
    // lo convierte a vector de bytes
    byte[] buffer= mensaje.getBytes();
    //ahora construyo el paquete, especifico destino
    // local y puerto 7
    DatagramPacket p=new DatagramPacket( buffer,
        mensaje.length(),
        InetAddress.getLocalHost(),7);
    s.send(p); //envio datagrama
}
}
public static void main(String param[]) throws
    SocketException, IOException {
    // se instancia el thread
    ServidorEcoUdp servidor=new ServidorEcoUdp();
    // lo lanzo
    servidor.start();
    // cliente de eco UDP
    clienteEcoUdp();
}
}
```

7.5 Servicio orientado a conexión (*Stream Socket*).

El uso de este tipo de *sockets* permite a las aplicaciones cliente y servidor disponer de un *stream* que facilita una comunicación libre de errores. Esto va a ser muy útil siempre que se desee fiabilidad en la comunicación.

El comportamiento para usar este tipo de *socket* es diferente en el cliente y el servidor. Cada uno de ellos utilizará unos métodos distintos. El esquema básico pasa por suponer que el servidor adoptará un papel pasivo y procederá a esperar conexiones de los posibles clientes. Mientras que los clientes serán los encargados de solicitar conexiones a los servidores de forma activa.

Se puede plantear un símil con el teléfono. Si dos usuarios desean comunicar por teléfono entre sí, no es una buena estrategia que ambos intenten llamar al otro, pues al estar los dos teléfonos descolgados siempre obtendrán señal de comunicando. Es mejor que acuerden que uno de ellos (el servidor) espere la

llamada del otro (cliente). De este modo no habrá problema para establecer la comunicación.

7.5.1 Operaciones en el servidor.

7.5.1.1 Creación del ServerSocket.

En Java existen dos constructores para crear un ServerSocket:

```
public ServerSocket(int port);
```

```
public ServerSocket(int port, int count);
```

En el primer caso se crea un *socket* local al que se enlaza el puerto especificado y acepta hasta 50 peticiones en cola (pendientes) de conexión por parte de los clientes.

En el segundo caso se puede especificar el número máximo de peticiones de conexión que se pueden mantener en cola.

En cualquiera de los dos casos, un puerto 0 indica que se utilice cualquier puerto disponible.

En la mayoría de los casos no suele importar en qué puerto se halle el servidor, pero es fundamental que el puerto escogido sea conocido por el cliente, ya que, de no ser así no se podrá establecer la conexión.

En otras ocasiones se trata de protocolos normalizados que tendrán que atender en un puerto local específico. Así, por ejemplo, el servidor de SMTP (*Simple Mail Protocol*) para correo electrónico “escucha siempre en el puerto 25 o el de HTTP (*HyperText Transfer Protocol*), el del WWW, “escucha” siempre en el 80. Los primeros 1024 puertos (del 0 al 1.023) son de uso reservado para el sistema.

7.5.2 Espera de conexiones de clientes.

```
public Socket accept();
```

Comunicaciones TCP/IP

Sobre un `ServerSocket` se puede realizar una espera de conexión por parte del cliente mediante el método `accept()`. Este método es bloqueante, el proceso espera a que se realice una conexión por parte del cliente para seguir su ejecución.

Una vez que se establece una conexión por el cliente, el método `accept()` devuelve un objeto de tipo `Socket`, a través del cual se establecerá la comunicación con el cliente.

Los objetos de tipo `ServerSocket` únicamente sirven para aceptar llamadas de clientes, no para establecer comunicaciones con los clientes. Es una especie de encargado de recibir llamadas (peticiones de conexión) que no es capaz de completarlas, tan sólo nos avisa cuando se han producido y nos proporciona la información necesaria para que podamos completarlas mediante la creación de un objeto `Socket`.

7.5.3 Operaciones en el cliente.

Como ya se ha indicado, es el cliente el que iniciará activamente el proceso de conexión. Para poder conectar con algún servidor, el cliente necesita varias cosas:

1. Conocer la dirección IP dónde reside el servidor⁴⁸.
2. Conocer el puerto en el que está esperando conexiones el servidor.

7.5.3.1 Creación del Socket.

Desde el punto de vista del cliente, será necesario realizar las peticiones a una dirección destino determinada en la que se encontrará esperando el servidor

⁴⁸ Se recuerda que la palabra servidor hace referencia a un proceso en un determinado ordenador. Así cuando se habla del servidor se está haciendo referencia a **ese** proceso servidor en **ese** ordenador.

⁴⁸ El verificador es un mecanismo de la máquina virtual Java (JVM) que comprueba que no se viole ninguna restricción del lenguaje Java. Se utiliza para evitar ataques de virus o programas (o applets) malintencionados.

⁴⁸ Esto es por razones de seguridad. Piénsese qué podría pasar si se pudiera depurar y cambiar contenidos de atributos en una programa Java en ejecución.

⁴⁸ 127.0.0.1 es la dirección IP de *loopback*. Es una dirección para pruebas. Algunos sistemas aceptan también la dirección 127.0.0.0

⁴⁸ En realidad se utiliza la interface `Hablador` (`Hablador h1,h2,h3,h4;`). Se realiza una conversión de tipos. Véase: B. Conversión de tipos. En la página 273.

⁴⁸ El valor de separador de paths están en la propiedad del sistema `path.separator` y el valor de separador de directorios en un path está en la propiedad del sistema `file.separator`. Estos valores pueden obtenerse con `System.getProperty()`.

Comunicaciones TCP/IP

(como se indicó anteriormente, esto supone especificar la dirección IP más el número de puerto).

Existen cuatro constructores en Java para implementar un *socket*:

```
public Socket(InetAddress address, int port);
```

```
public Socket(InetAddress address, int port, boolean stream);
```

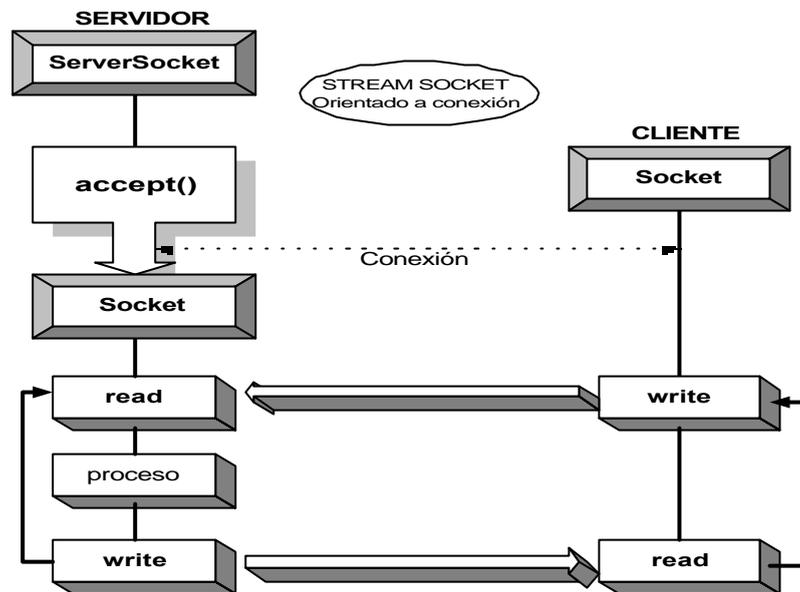
```
public Socket(String host, int port);
```

```
public Socket(String host, int port, boolean stream);
```

Para la creación de un Socket, hay que proporcionar la dirección (o nombre) del host y el puerto, del host remoto.

Servidor: 130.1.1.20	Cliente: 130.1.1.14
<code>ServerSocket(1500)</code> <code>accept()</code>	<code>Socket("130.1.1.20",1500)</code>
<code>Socket</code> "130.1.1.14",1024,1500	<code>Socket</code> "130.1.1.20",1500,1024

7.5.3.2 Envío y recepción de datos a través de sockets.



Como se representa en la figura, el servidor crea un *socket* (`ServerSocket`), le asigna una dirección y un puerto y acepta llamadas (`accept`). Tras el `accept`, el proceso queda bloqueado a la espera de recibir una llamada. Una vez se recibe una llamada (`Socket` cliente con la dirección y puerto del servidor), el `accept` crea un nuevo *socket*, por lo que todo servidor orientado a conexión requerirá, al menos, dos *sockets*, uno para recibir conexiones y otro para procesarlas. Cuando un cliente desea comunicarse, crea su *socket* (`socket`), y establece una conexión al puerto establecido. Es únicamente en ese momento cuando existe la conexión y ésta durará hasta que se libere (`close()`).

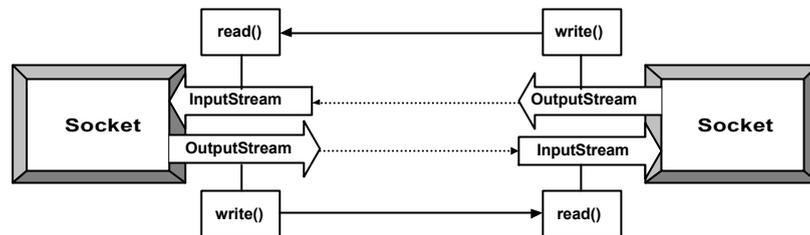
Los *sockets* tienen asociados un “*Stream*” de entrada (`InputStream`) y otro de salida (`OutputStream`) a través de los cuales se puede leer y escribir datos respectivamente.

La forma de obtener estos *streams* a partir del *socket* es la siguiente:

`objetoDeTipoSocket.getInputStream()` Devuelve un objeto de tipo `InputStream`

`objetoDeTipoSocket.getOutputStream()` Devuelve un objeto de tipo `OutputStream`

Comunicaciones TCP/IP



Envío de datos:

Para enviar datos, pueden utilizarse los `OutputStream` de los *socket* directamente si lo que se pretende enviar es un flujo de bytes sin buffer, o puede crearse un objeto de tipo *stream* de datos más evolucionado basado en el `OutputStream` que proporciona el *socket*.



Por ejemplo:

```
...
Socket sock = new Socket("127.0.0.1",1500);
PrintWriter escritura;
BufferedReader teclado;
String linea;
teclado=new BufferedReader(new InputStreamReader(System.in));
escritura=new PrintWriter(sock.getOutputStream(),true);
do {
    linea=teclado.readLine();
    escritura.println(linea);
} while (linea.compareTo("#")!=0);
...
```

Recepción de datos:

Para recibir datos, al igual que para el envío, puede utilizarse el `InputStream` que proporciona el *socket* o definir un nuevo objeto de tipo *stream* más eficiente.



Por ejemplo:

```
...
BufferedReader lectura;
String s;
```

Comunicaciones TCP/IP

```
lectura=new BufferedReader( new
InputStreamReader(socketRecepcion.getInputStream()));
while ("#".compareTo(s = lectura.readLine())!=0) {
System.out.println(nombre+": "+s);
}
```

7.5.3.3 Ejemplo.

Habitualmente es necesario construir dos programas, uno para el cliente y otro para el servidor, para poder realizar algún tipo de prueba. Una solución puede pasar por emplear un cliente o servidor ya construido para realizar esas pruebas, con lo que sólo sería necesario construir el programa complementario.

Por ejemplo, se puede utilizar el programa *telnet* como cliente de muchas aplicaciones simplemente especificando el puerto al que se desea conectar (ya que por defecto este programa se conecta al puerto TCP 23 dónde se encuentra el servidor).

Del mismo modo, muchos sistemas operativos (excepto Windows 95) incluyen un servidor de eco TCP en el puerto 7 (el mismo puerto que en UDP) que puede permitir verificar que una aplicación cliente, que se está desarrollando, al menos comunica correctamente.

En el siguiente ejemplo se presentan el cliente y el servidor de eco. Es importante señalar que ambos funcionan por líneas y no por caracteres, es decir, las peticiones finalizan con CR+LF y el eco se devuelve al final de cada línea.



```
import java.net.*;
import java.io.*;
class ClienteEcoTcp {
    public static void main(String args[]) throws
        IOException, SocketException {
        // servidor ZOLTAR (Linux)
        Socket tcp=new Socket("158.42.53.127",7);
        System.out.println("Con:"+tcp.toString());
        // conectamos salida
        OutputStream salida=tcp.getOutputStream();
        // conectamos entrada
        InputStream entrada=tcp.getInputStream();
        // buffer transmision
        byte[] buffer= new byte[256];
        // buffer recepcion
```

Comunicaciones TCP/IP

```
byte[] buffer2= new byte[256];
int longitud,longitud2=0;
while (true) {
    // leemos una linea
    longitud=System.in.read(buffer);
    // la enviamos por el socket
    salida.write(buffer,0,longitud);
    // leemos la respuesta
    longitud2=entrada.read(buffer2);
    // mostramos respuesta
    System.out.write(buffer2,0,longitud2);
}
}
```



```
import java.net.*;
import java.io.*;
class ServidorEcoTcp {
    public static void main(String args[]) throws
        IOException, SocketException {
        // solo funcionara en W95 (no en UNIX)
        ServerSocket ss=new ServerSocket(7);
        Socket tcp;
        System.out.println("Esperando conexion ...");
        tcp=ss.accept();
        System.out.println("Con:"+tcp.toString());
        // conectamos salida
        OutputStream salida=tcp.getOutputStream();
        // conectamos entrada
        InputStream entrada=tcp.getInputStream();
        byte[] buffer= new byte[256];
        int longitud;
        while (true) {
            // leemos una linea de socket
            longitud=entrada.read(buffer);
            // salimos si solo fue CRLF
            if (longitud<3) {tcp.close(); break;}
            // la enviamos a pantalla
            System.out.write(buffer,0,longitud);
            // mostramos respuesta
            salida.write(buffer,0,longitud);
        }
    }
}
```

Comunicaciones TCP/IP

Si se desea probar estos ejemplos en Windows NT o UNIX bastará con cambiar el número de puerto y escoger uno por encima del 1023, para no colisionar con los ya asignados y poder utilizar un puerto sin necesidad de permisos adicionales (*root*).

7.6 CUESTIONES

1. ¿Qué dos tipos de servicio básicos ofrece la comunicación en Java?
2. ¿Dónde está, habitualmente, la interface de usuario en un programa cliente-servidor?
3. ¿Qué es un *socket*?
4. ¿Qué tipos de *socket* proporciona Java y en qué “paquete” se encuentran?
5. ¿Cuál es la finalidad de un *RAW SOCKET*?
6. ¿Por qué se han creado dos *threads* en el programa del eco UDP? ¿De qué se encarga cada uno?
7. ¿Por qué en el programa de eco UDP se reciben dos ecos de una misma línea?
8. ¿Por qué se dice que un *stream socket* permite una comunicación fiable?
9. ¿Cuál es la finalidad de la clase `ServerSocket`?
10. ¿Cuál es la condición de finalización del servidor ejemplo de eco TCP?

A. LAS HERRAMIENTAS JAVA.

El JDK 1.1 proporciona un conjunto de herramientas para la creación de aplicaciones en Java. Entre ellas se destacan el compilador y el intérprete (JVM) o máquina virtual Java. También dispone de un depurador, un desensamblador y un generador de documentación en formato HTML (*HyperText Markup Language*).

En este apéndice únicamente van a tratarse las herramientas básicas y necesarias para seguir este libro.

A.1 *javac* (compilador).

El programa **javac** es el compilador de Java.

javac_g es una versión que no genera código optimizado. Se usa conjuntamente con el depurador **jdb**. Acepta los mismos parámetros que **javac**.

Sintaxis:

```
javac [ Opciones ] nombreFichero.java
```

javac compila programas escritos en Java; el fichero fuente debe tener la extensión **.java** se generan tantos ficheros en *bytecodes* con la extensión **.class** como clases tenga el mismo. Se crean en el mismo directorio y, si alguno tiene método `main()`, podrá ser interpretado por la herramienta **java**.

La variable de entorno CLASSPATH definida en el fichero autoexec.bat contiene los directorios donde se encuentran almacenadas las clases, por lo tanto, para que el compilador encuentre clases en el directorio en el que se está trabajando, es conveniente añadir el directorio actual (.). Ejemplo:

```
set classpath= . ; c:\www\java\java\lib ; c:\www\java\misclases
```

A.1.1 Opciones.

Las siguientes opciones pueden aplicarse tanto a **javac** como a **javac_g**.

Las herramientas Java.

-classpath *directorios*

Especifica los directorios donde el compilador `javac` buscará las clases. Este *path* anulará la variable de entorno **classpath** declarada en el fichero `autoexec.bat`. Ejemplo:

```
javac MiClase.java -classpath .;c:\MisClases; c:\MisClases\Ejemplos
```

-d *directorio*

Especifica el directorio raíz para guardar las clases que se compilen. Ejemplo:

```
javac MiClase.java -d c:\MisClases
```

Hace que las clases (.class) se almacenen en `c:\MisClases`.

Si no se indica esta opción, las clases (.class) se guardarán en el mismo directorio en el que se encuentran los ficheros fuente.

-depend

Indica que sean recompiladas las clases a las que se hace referencia. Normalmente no se recompilan a no ser que no existan o estén desfasadas de fecha.

-deprecation

Genera mensajes sobre qué métodos o atributos se están usando, que se mantienen por compatibilidad con versiones anteriores del JDK pero han sido sustituidas por otros en el actual JDK.

-g

Hace que se generen tablas de información que será utilizada por el depurador **jdb**.

-nowarn

Desactiva la generación de mensajes de aviso (*warnings*) por el compilador.

-nowrite

Compila normalmente pero no genera ficheros *bytecodes* (.class).

Las herramientas Java.

-O

Optimiza el código generado (el tamaño puede ser mayor) haciendo que los métodos `static`, `final` y `private` sean “*inline*”.

-verbose

Hace que el compilador y enlazador muestren mensajes sobre qué ficheros fuentes se están compilando y qué clases se están cargando.

A.2 java (Intérprete).

Es el intérprete de Java. Interpreta *bytecodes* generados por el compilador **javac**.

java_g es una versión que se usa conjuntamente con el depurador **jdb**. Acepta los mismos parámetros que java.

Sintaxis:

java [opciones] nombreDeClase [argumentos]

nombreDeClase es el nombre del fichero `.class` que se ejecutará (a diferencia del compilador, en que hay que poner `.java` en el nombre del fichero fuente, en este caso NO se pone la extensión `.class`). Esta clase debe contener una función `main()` para ejecutarse. Normalmente debe indicarse el nombre completo incluyendo el paquete en el que se encuentra definida (por ejemplo `java java.lang.String`), a no ser que no se haya indicado paquete y, por lo tanto, pertenezca al paquete por defecto.

argumentos es la lista de argumentos (opcionales) que se pasa a la función `main()` de la clase.

A.2.1 Opciones.

-cs o -checksource

Para cada una de las clases compiladas `.class` que se cargan se compara la fecha de última modificación con la de los ficheros fuente (`.java`). Si esta fecha es anterior, se recompilan los fuentes correspondientes, de forma que se asegura que las clases siempre estén actualizadas.

Las herramientas Java.

-classpath *directorios*

Especifica los directorios donde se buscarán las clases (.class). Este path anulará la variable de entorno classpath declarada en el fichero autoexec.bat. Ejemplo:

```
java -classpath .;c:\MisClases; c:\MisClases\Ejemplos MiClase
```

-debug

Permite al depurador jdb trabajar en la sesión Java iniciada. La máquina virtual Java, proporciona un password al compilar con esta opción. Este password deberá ser suministrado al depurador jdb para trabajar en esta sesión (☞ véase el punto A.4 jdb (Depurador).En la página 259)

-DnombreProperty=nuevoValor

Establece la propiedad del sistema (*Property*) al valor indicado.

Ejemplo: java -Dline.separator=X HolaMundo Causaría que la salida del típico programa Hola, mundo
(`System.out.println("Hola, mundo");`)
fuera: Hola, mundoX.

-help

Muestra las opciones que acepta el intérprete.

-msn

Fija la cantidad mínima de memoria que se debe asignar en bytes al recolector de basura (por defecto es 1 Mb). Si se desea especificar la cantidad en Kb, se añade una k al final de la cantidad, y si se desea expresar en Mb, se añade una m. La cantidad mínima es n = 1000 bytes. Ejemplo:

```
java -ms10m MiClase (asigna 10 Mb de memoria mínima).
```

-mxn

Fija la cantidad mínima de memoria asignada al recolector de basura (de forma similar a la opción anterior). por defecto son 16 Mb.

Las herramientas Java.

-noasyncgc

Desactiva la recolección de basura asíncrona. Por defecto está activada y se ejecuta en paralelo (como un thread independiente). Si se desactiva con esta opción, el recolector de basura (*garbage collector*) sólo se ejecuta cuando se sobrepasa la cantidad de memoria asignada al recolector de basura.

-noclassgc

Desactiva la recolección de basura de las clases. Normalmente esta opción está activada y permite que el recolector de basura libere la memoria asignada a clases que no se están utilizando.

-noverify

Desactiva el verificador de código Java. Véase la opción **-verify** y **-verifyremote**.

-prof [*fichero*]

Genera un fichero de salida con información sobre las operaciones ejecutadas por la MVJ. Si no se indica un fichero de salida, se creará un fichero `java.prof` en el directorio actual.

-ossn

Especifica la cantidad de memoria de *stack* asignada en un thread a código escrito en Java. La cantidad se especifica en bytes, siendo la cantidad mínima $n = 1000$ bytes. Se puede expresar la cantidad en Kb añadiendo la letra K y en Mb añadiendo la letra m. Ejemplo:

```
java -oss3m MiClase           (asigna 3Mb de stack para Java).
```

-ssn

Especifica la cantidad de memoria de *stack* asignada en un thread a código escrito en C. La cantidad se especifica en bytes, siendo la cantidad mínima $n = 1000$ bytes. Se puede expresar la cantidad en Kb añadiendo la letra K y en Mb añadiendo la letra m. Ejemplo:

```
java -ss200k MiClase         (asigna 200 Kb de stack para C).
```

Las herramientas Java.

-t (Esta opción sólo se encuentra en `java_g`).

Muestra en pantalla una traza de cada una de las instrucciones (*bytecode*) ejecutadas por la JVM.

-v o -verbose

Muestra un mensaje con información por cada clase que es cargada en memoria.

-verbosegc

Muestra un mensaje cada vez que el recolector de basura (*garbage collector*) actúa liberando memoria.

-verify

Ejecuta el verificador⁴⁹ para todo el código Java.

-verifyremote

Ejecuta el verificador para todo el código Java cargado en el sistema mediante el *classloader* (cargador de clases). Esta opción está activada en el intérprete **java**.

-version

Muestra la versión del compilador y termina. No ejecuta código.

A.3 *javaw (Intérprete).*

Es el un intérprete de Java, exactamente igual y con las mismas opciones que el intérprete **java**. (↪ Véase el punto A.2 **java (Intérprete)**, en la página 255). Se diferencia de éste en que al intérprete **javaw** no se le asocia una consola Windows (DOS). Normalmente se utiliza éste para ejecutar aplicaciones con interface de usuario gráfico (GUI).

javaw_g es una versión que se usa conjuntamente con el depurador **jdb**. Acepta los mismos parámetros que **java**.

⁴⁹ El verificador es un mecanismo de la máquina virtual Java (JVM) que comprueba que no se viole ninguna restricción del lenguaje Java. Se utiliza para evitar ataques de virus o programas (o applets) malintencionados.

A.4 *jdb* (Depurador).

El *jdb* es un depurador de línea de comandos. Sirve para encontrar y corregir errores en programas Java. Actúa en conjunción con unas API's que se encuentran en el paquete `sun.tools.debug`.

Existen dos formas de utilizar el depurador *jdb*:

La más usual es la de sustituir el comando `java fichero.class` por `jdb fichero.class`. En este caso, es el depurador quien lanza la máquina virtual Java (pueden utilizarse los mismos parámetros que con el programa `java`). Se carga la clase y el intérprete se detiene antes de ejecutar la primera instrucción.

Ejemplo:

```
C:\j>jdb Hola
Initializing jdb...
0xe80698: class(Hola)
>
```

Una vez iniciada una sesión de la JVM (máquina virtual Java), y siempre que se haya utilizado la opción `-debug`⁵⁰, puede acoplarse una sesión de depuración. Para ello se utiliza el password proporcionado por el intérprete (☞ Véase la opción `-debug` del punto A.2 `java` (Intérprete). En la página 256)

La sintaxis para trabajar en esta forma es:

`jdb -host nombreHost -password password`

Ejemplo:

Suponiendo que se esté trabajando con Windows95, pueden iniciarse dos sesiones DOS. En una de ellas se ejecutará `java -debug nombreFichero.class`. Esto iniciará una sesión Java y mostrará en pantalla un password. En la otra sesión DOS, puede iniciarse el depurador con `jdb -host 127.0.0.1`⁵¹ `-password clave` (donde `clave` es la proporcionada por el intérprete en la otra sesión DOS).

⁵⁰ Esto es por razones de seguridad. Piénsese qué podría pasar si se pudiera depurar y cambiar contenidos de atributos en una programa Java en ejecución.

⁵¹ 127.0.0.1 es la dirección IP de *loopback*. Es una dirección para pruebas. Algunos sistemas aceptan también la dirección 127.0.0.0

A.4.1 Comandos básicos.

Sólo se van a indicar los comandos más importantes. Se puede obtener una lista completa con una pequeña explicación de cada uno de ellos mediante el comando **?** o **help**.

? o help

Muestra una lista de los comandos que acepta el depurador.

dump *expresión*

Muestra la información referente a un determinado objeto visualizando los valores de los atributos. También pueden mostrarse expresiones Java (actualmente no soporta la llamada a métodos, pero sí se puede acceder a atributos). Ejemplo: > **dump** **MiClase.miVar**. El objeto puede ser indicado por nombre o por objectID (número en hexadecimal que lo identifica).

exit o **quit**

Finaliza la sesión del depurador.

print *expresión*

Muestra un determinado objeto llamando a su método `toString()`. También pueden mostrarse expresiones Java (actualmente no soporta la llamada a métodos, pero sí se puede acceder a atributos). Ejemplo: > **print** **MiClase.miVar**. El objeto puede ser indicado por nombre o por objectID (número en hexadecimal que lo identifica).

threads

Muestra los threads actuales organizados por grupos de threads. Los threads son identificados por un objectID o, si están en el grupo de threads por defecto, con la forma `t@índice`

where

Vuelca en pantalla el contenido de la pila (*stack*) del thread actual.

where all

Vuelca en pantalla el contenido de las pilas (*stack*) de todos los threads.

where *idThread*

Vuelca el contenido de la pila del thread indicado por el objectID o mediante la forma *t@índice*.

A.5 ***javah*** (Generador de cabeceras C).

La herramienta **javah** genera ficheros de cabecera C para comunicación entre aplicaciones Java y aplicaciones C.

Sintaxis:

javah [*opciones*] *nombreDeClase*

La nueva interfaz de métodos nativos JNI (*Java Native Interface*) no necesita información del fichero de cabecera .h.

Hay que evitar en lo posible el uso de código no Java, ya que impide la portabilidad. El uso de estas “posibilidades” excede al cometido del libro.

javah_g es una versión que no optimiza. Se utiliza para el **jdb**.

Opciones:

- help** muestra las opciones y una breve explicación de las mismas
- o *fichero*** Indica el nombre del fichero de salida.
- d *directorio*** Indica el directorio de salida.
- jni** Crea un fichero cabecera .h al estilo JNI.
- td** Indica el directorio para ficheros temporales.
- stubs** Crea un fichero de *stubs*.
- trace** Añade información de seguimiento al fichero de *stubs*.
- v *verbose*** Información adicional
- classpath** Directorios classpath.
- version** muestra la versión de **javah**

A.6 *javap* (Desensamblador).

La herramienta **javap** es un desensamblador. A partir de un fichero en *bytecodes* (.class) muestra información sobre la clase, los atributos y los métodos. Puede ser útil para conocer los métodos de los que dispone una clase de la que no se dispone el código fuente ni documentación.

Ejemplo extraído de la documentación del JDK 1.1:

```
import java.awt.*;
import java.applet.*;

public class DocFooter extends Applet {
    String date;
    String email;
    public void init() {
        resize(500,100);
        date = getParameter("LAST_UPDATED");
        email = getParameter("EMAIL");
    }
    public void paint(Graphics g) {
        g.drawString(date + " by ",100, 15);
        g.drawString(email,290,15);
    }
}
```

La compilación del programa anterior **javac DocFooter.java** genera el fichero DocFooter.class.

Aplicando el desensamblador **javap DocFooter** se muestra la información:

```
Compiled from DocFooter.java
public class DocFooter extends
java.applet.Applet {
    java.lang.String date;
    java.lang.String email;
    public void init();
    public void paint(java.awt.Graphics);
    public DocFooter();
}
```

La sintaxis es la siguiente:

javap [opciones] nombreDeClase

A.6.1 Opciones.

-b

Asegura la compatibilidad con versiones anteriores del JDK.

-c

Muestra el código desensamblado (bytecodes) de los métodos.

Ejemplo (del Fichero ejemplo Hola.java):

```
class Hola {
    public static void main(String argumentos[]) {
        System.out.println("hola, mundo");
    }
}
```

El desensamblado **javap -c Hola** genera la siguiente salida:

```
Compiled from Hola.java
synchronized class Hola extends java.lang.Object
    /* ACC_SUPER bit set */
    {
        public static void main(java.lang.String[]);
        Hola();
    }
```

```
Method void main(java.lang.String[])
    0 getstatic #7 <Field java.io.PrintStream out>
    3 ldc #1 <String "hola, mundo">
    5 invokevirtual #8 <Method void
println(java.lang.String)>
    8 return
```

```
Method Hola()
    0 aload_0
    1 invokespecial #6 <Method java.lang.Object()>
    4 return
```

Las herramientas Java.

-classpath *directorios*

Especifica los directorios donde el desensamblador `javap` buscará las clases. Este path anulará la variable de entorno **classpath** declarada en el fichero `autoexec.bat`. Ejemplo:

```
javap MiClase -classpath .;c:\MisClases; c:\MisClases\Ejemplos
```

-J *flag*

Pasa el *flag*, directamente al sistema de ejecución.

-h

Genera código que puede ser utilizado como fichero de cabecera (.h) en C.

-l

Muestra tablas de variables locales y números de líneas.

-package

Muestra sólo las clases y métodos `protected` y `public` del paquete (esta es la opción por defecto).

-protected

Muestra sólo las clases y atributos `protected` y `public`.

-private

Muestra todas las clases y atributos.

-public

Muestra sólo las clases y atributos públicos (`public`).

-verbose

Muestra el tamaño de la pila, número de variables y parámetros de los métodos.

-version

Muestra la versión de **javap**.

A.7 javadoc (Generador de documentación).

Sintaxis:

```
javadoc [ opciones ] { [ paquete | fuente.java ] }
```

javadoc rastrea ficheros fuente .java en busca de unos comentarios especiales que sirven para generar páginas en formato HTML con documentación sobre las clases al estilo de la documentación del JDK. Se puede indicar una lista de ficheros fuente .java o de paquetes. javadoc genera un fichero por cada fichero .java encontrado, uno para los paquetes (packages.html), otro para la jerarquía de clases (tree.html), y otro con el índice de todos los elementos (AllNames.html).

Un comentario de documentación comienza con `/**` y termina con `*/`. Puede constar de varias líneas. Todos los caracteres *, espacios y tabuladores iniciales de cada una de estas líneas son ignorados. En los comentarios se pueden incluir etiquetas HTML, siempre que no sean de título `<H1>`, `<H2>`, ..., `<H6>`, ya que **javadoc** genera toda una jerarquía de páginas y títulos..



```
/**  
 * Esto es un comentario de <I> documentación </I>.  
 */
```

El primer comentario de documentación debería ser un pequeño resumen, que terminará siempre con un punto. Este comentario se insertará en el resumen al principio del documento HTML generado.

Los comentarios de documentación sólo son reconocidos si están justo antes de la declaración de clases (class), interfaces, constructores, métodos, o atributos.

Existen unas etiquetas especiales que generan la documentación API al estilo de la documentación del JDK. Estas etiquetas comienzan con el carácter arroba (@) siempre al principio de línea.

A.7.1 Etiquetas de clase o interface.

Estas etiquetas que se enumeran a continuación deben colocarse justo antes de la declaración de una clase o interface

@author *texto*

Crea una entrada de autor de la clase. Puede haber varias etiquetas @author. Debe incluirse la opción `-author` en **javadoc** para que aparezca la entrada autor.

@version *texto*

Crea una entrada (debe ser única) que indica la versión de la clase o interface. Debe incluirse la opción `-version` en **javadoc** para que aparezca esta entrada.

@since *texto*

Crea una entrada para indicar desde qué momento existe.

@see *nombreDeClase*

Crea un hipervínculo *see also* (véase también). Puede haber varias etiquetas @see.

Ejemplos:

```
@see java.lang.String
@see String
@see String#equals
@see java.lang.Object#wait(int)
@see Character#MAX_RADIX
@see <a href="spec.html">Java Spec</a>
```



```
/**
 * Esta clase es un ejemplo típico.
 * Muestra en pantalla el texto:
 * <pre>
 *   hola, mundo
 * </pre>
 *
 * @author Víctor Alonso
 * @author Miguel Sánchez
```

Las herramientas Java.

```
* @version 1.0
* @see    java.System.out
* @see    java.System.println
*/
class Hola {
    public static void main(String argumentos[]) {
        System.out.println("hola, mundo");
    }
}
```

javadoc -author -version -private Hola.java

```
Generating package.html
Generating documentation for class Hola
Generating index
Sorting 2 items...done
Generating tree
```

Documento Hola.html generado:

Class Hola

```
java.lang.Object
|
+----Hola
```

```
class Hola
extends Object
```

Esta clase es un ejemplo típico. Muestra en pantalla el texto:

```
    hola, mundo
```

Version:

1.0

Author:

Victor Alonso, Miguel Sánchez

See Also:

out, println

Constructor Index

- [Hola\(\)](#)

Method Index

- [main\(String\[\]\)](#)

Constructors

- **Hola**

```
Hola()
```

Methods

- **main**

```
public static void main(String argumentos[])
```

A.7.2 Etiquetas de atributo.

Esta etiqueta debe colocarse justo antes de la declaración del atributo.

@see *nombreDeClase*

Es la única etiqueta que puede tener un atributo. Crea un hipervínculo *see also* (véase también). Puede haber varias etiquetas @see

A.7.3 Etiquetas de constructor o método.

Estas etiquetas que se enumeran a continuación deben colocarse justo antes de la declaración de un constructor o método.

@param *nombreDeParámetro descripciónDeParámetro*

Añade una entrada de parámetro en la sección “Parameters”. Esta descripción puede ocupar más de una línea. Puede haber varias etiquetas @param.

@exception *nombreDeClase descripción*

Añade una sección “Throws” con un enlace a la documentación de la excepción indicada

@return *texto*

Añade una sección “Returns” con la descripción del valor devuelto.

@see *nombreDeClase*

Crea un hipervínculo *see also* (véase también). Puede haber varias etiquetas @see.

A.7.4 Opciones.

-author

Inserta las etiquetas @author (omitidas por defecto).

Las herramientas Java.

-classpath *directorios*

Especifica los directorios donde el javadoc buscará tanto sus propias clases como los ficheros fuente (.java). Este path anulará la variable de entorno **classpath** declarada en el fichero autoexec.bat.

-d *directorioDestino*

Indica el directorio de salida donde javadoc creará la documentación HTML.

-J *flag*

Pasa el *flag*, directamente al sistema de ejecución.

-noindex

Omite la generación del índice de paquetes (producido por defecto).

-notree

Omite la generación de la jerarquía de clases e interfaces (producido por defecto).

-package

Muestra sólo las clases y métodos `protected` y `public` del paquete.

-protected

Muestra sólo las clases y atributos `protected` y `public` (esta es la opción por defecto).

-private

Muestra todas las clases y atributos.

-public

Muestra sólo las clases y atributos públicos (`public`).

-sourcepath *directorios*

Indica los directorios donde se encuentran los ficheros fuente (.java). Por ejemplo, si se deseara generar documentación sobre el paquete `java.lang` y éste se encontrara en el directorio `c:\jdk\java\lang`, se indicaría la opción **-sourcepath c:\jdk** (hay que indicar el directorio raíz del paquete). Si por

Las herramientas Java.

el contrario se deseara generar documentación sobre la clase `java.lang.String`, se incluiría la opción `-sourcepath c:\jdk\java\lang`.

-verbose

Muestra más información al generar la documentación.

B. CONVERSIÓN DE TIPOS.

Toda expresión en Java tiene un tipo de dato asociado, ya sea simple o referencial, y puede ser deducido de la estructura de la expresión y los literales que pudiera contener.

Las conversiones de tipos pueden ser apropiadas para cambiar el tipo de una determinada expresión que no se corresponda con el tipo necesario.

Tipos de conversiones de tipos:

Conversión por ampliación: Consiste en cambiar el tipo de dato por otro cuyo rango es mayor y por lo tanto, contiene al primero. En este tipo de conversión no se pierde información (aunque puede perderse precisión en la conversión de datos enteros a reales).

Conversión por reducción: En este tipo de conversión sí que puede perderse información debido a que se pretende almacenar más información de la que “cabe” en el tipo de destino. Al ser una conversión “semidestructiva”, es necesario indicar que se pretende realizar la misma explícitamente para evitar pérdidas de información por conversiones por reducción inadvertidas.

B.1 Conversión por ampliación de tipos de datos simples.

En estos casos no se pierde información sobre la magnitud de los valores numéricos.

Tipo a convertir	Tipo resultante
byte	short int long float double

Conversión de tipos.

short	int long float double
char	int long float double
int	long float double
long	float double
float	double

Para los casos de asignación no es necesario realizar ninguna conversión explícita de tipos. Por ejemplo, se puede asignar un valor de tipo byte a una variable de tipo short, int, long, float o double.

También se puede asignar un carácter (char) a una variable de tipo int, long, float o double.

```
int i = 'A';
```

Los únicos casos en los que no se preserva necesariamente la exactitud entre los valores del tipo a convertir y el tipo resultante, ya que **puede perderse precisión** son:

Tipo a convertir	Tipo resultante
int	float
long	float double

En estos casos de pérdida de precisión, se redondea al valor más cercano según el estándar IEEE 754.

Conversión de tipos.



Ejemplo de pérdida de precisión:

```
class Conversion1 {
    public static void main(String arg[]) {
        int i = 1234567890;
        float f = i;
        System.out.println("f="+f);
        // (int)f convierte f, de tipo float, a entero
        System.out.println(i-(int)f);
    }
}
```

Produce la siguiente salida:

```
f=1.23456794E9
-46
```

Esto es debido a que el tipo `float` sólo es preciso hasta 8 dígitos significativos.

B.2 Conversión por reducción de tipos de datos simples.

En estos casos puede perderse información ya que los rangos de representación son distintos, no estando el del tipo de dato a convertir incluido en el rango del tipo de destino.

La forma de convertir de un tipo a otro es trasladando los n bits menos significativos a la zona de memoria de destino con capacidad para esos n bits. De esta forma puede perderse incluso el signo de valores numéricos (representados en complemento a dos).

Tipo a convertir	Tipo resultante
byte	char
short	byte char
char	byte short
int	byte short char

Conversión de tipos.

long	byte short char int
float	byte short char int long
double	byte short char int long float

En estos casos, en que puede perderse información en el proceso de conversión de tipos, es necesario indicarlo explícitamente para evitar que esta pérdida se produzca de forma accidental, o por lo menos de forma inadvertida por el programador.

La forma de realizar la conversión de tipo por reducción es mediante la anteposición a la expresión a convertir, el tipo de destino encerrado entre paréntesis.



```
class Conversion2 {  
    public static void main(String arg[]) {  
        float f1 = 1.234567f;  
        float f2 = 7.654321f;  
        int i1 = (int)f1;  
        int i2 = (int)f2;  
        System.out.println("f1="+f1+" i1="+i1);  
        System.out.println("f2="+f2+" i2="+i2);  
        int i = 1234567;  
        short s = (short)i;  
        System.out.println("i="+i+" s="+s);  
    }  
}
```

Conversión de tipos.

Salida por pantalla:

```
f1=1.234567 i1=1
f2=7.654321 i2=7
i=1234567 s=-10617
```

B.3 Conversión por ampliación de tipos de datos referenciales.

Al igual que ocurre con los tipos de datos simples, también para los objetos pueden realizarse conversiones tanto por ampliación como por reducción.

Las conversiones por ampliación permitidas son:

Tipo a convertir	Tipo resultante
La clase null	- Cualquier clase, interface o vector
Cualquier clase C	- Cualquier clase R siempre que C sea subclase de R - Cualquier interface I si C implementa I - La clase Object
Cualquier interface I	- Cualquier interface K siempre que I sea subinterfaz de K - La clase Object
Cualquier vector A	- La clase Object - La clase Cloneable - Cualquier vector R siempre que el tipo de los elementos de A y R sean datos referenciales y exista una conversión por ampliación entre ellos.

En estos casos no es necesaria ninguna conversión explícita.



```
class C1 {
    public String clase;
    public C1() {
        clase = "class c1";
    }
}
class C2 extends C1{
    public C2() {
        clase = "class c2";
    }
}
```

Conversión de tipos.

```
    }  
}  
  
class Conversion3 {  
    public static void main(String arg[]) {  
        C1 c1 = new C2();  
        System.out.println(c1.clase);  
    }  
}
```

Produce la siguiente salida por pantalla:

```
clase c2
```

En el ejemplo, se ha declarado un objeto `c1` de la clase `C1`, que es subclase de `C2`. A `c1` se le asigna un objeto de la clase `C2` (al realizar la llamada al constructor de `C2`). Esto puede hacerse sin ninguna conversión explícita ya que `C2` es subclase de `C1`. Casi gráficamente puede entenderse mediante un ejemplo de la siguiente forma: imaginemos una clase `Ave` y una subclase `Pato`; si se dispone de una variable de la clase `Ave`, a dicha variable se le puede asignar un “Pato”, ya que un pato es, efectivamente, un ave. Por lo tanto, a una variable de la clase `Ave`, se le puede asignar un objeto “Pato” sin ninguna conversión explícita ya que es un proceso “natural”. El caso contrario no siempre es cierto. Si se dispone de una variable de la clase `Pato`, no se le puede asignar un objeto de la clase “Ave”, ya que cualquier ave no es un pato. Si se dispone de una variable de la clase `Pato` y se le desea asignar un objeto de la clase `Ave`, habrá que realizar una conversión explícita (ver punto siguiente) para “asegurar” que el ave que asignamos es “realmente” un pato.

B.4 Conversión por reducción de tipos de datos referenciales.

Las conversiones por reducción permitidas son:

Tipo a convertir	Tipo resultante
La clase Object	- Cualquier vector - Cualquier interface - Cualquier clase
Cualquier clase C	- Cualquier clase R siempre que C sea superclase de R - Cualquier interface I si C no es final y no implementa I
Cualquier interface I	- Cualquier clase C que no sea final

Conversión de tipos.

	<ul style="list-style-type: none">- Cualquier clase C que sea final siempre que C implemente I- Cualquier interface K siempre que I no sea subinterface de K y no tengan métodos con el mismo nombre pero distinto tipo de dato de retorno.
Cualquier vector A	<ul style="list-style-type: none">- Cualquier vector R siempre que el tipo de los elementos de A y R sean datos referenciales y exista una conversión por reducción entre ellos.

En estos casos es necesario indicarlo explícitamente.

La forma de realizar la conversión de tipo por reducción es mediante la anteposición a la expresión a convertir, el tipo de destino encerrado entre paréntesis.

El siguiente ejemplo muestra una conversión no permitida y genera un mensaje de error por el compilador:



```
class C1 {
    public String clase;
    public C1() {
        clase = "class c1";
    }
}
class C2 extends C1{
    public C2() {
        clase = "class c2";
    }
}

class Conversion4 {
    public static void main(String arg[]) {
        C1 c1 = new C1();
        C2 c2 = c1;
        System.out.println(c2.clase);
    }
}
```

Conversión de tipos.

```
Conversion4.java:16:      Incompatible      type      for
declaration. Explicit cast needed to
convert C1 to C2.
```

```
    C2 c2 = c1;
        ^
```

1 error

la conversión de un objeto de la clase C1 a la clase C2 necesita del programador la indicación de que se sabe que se está asignando un valor de la clase C1 a una variable de la clase C2, pero que en realidad c1 no contiene un objeto de la clase C1 sino un objeto de la clase C2.

Para realizar la conversión en el ejemplo anterior se antepone la clase a convertir encerrada entre paréntesis de la siguiente manera:



```
...
class Conversion4 {
    public static void main(String arg[]) {
        C1 c1 = new C1();
        // Conversión por reducción de tipo de
        // dato referencial
        C2 c2 = (C2) c1;
        System.out.println(c2.clase);
    }
}
```

Con esta conversión explícita se indica que el objeto almacenado en c1 es, en realidad, de la clase C2 (lo cuál no es cierto). Ahora el compilador no genera ningún error, pero al ejecutar el programa se genera una excepción:

```
java.lang.ClassCastException:
    at Conversion5.main(Conversion5.java:17)
```

No se habría generado ninguna excepción si en c1 hubiera un objeto de la clase C2 de la siguiente forma:



```
...
class Conversion4 {
    public static void main(String arg[]) {
        C1 c1 = new C2(); // Conversión por ampliación
        // Conversión por reducción de tipo de dato
        // referencial
    }
}
```

Conversión de tipos.

```
C2 c2 = (C2) c1;  
System.out.println(c2.clase);  
}  
}
```

La salida por pantalla sería:

```
clase c2
```

B.5 Conversiones de tipos no permitidas.

Tipo a convertir	Tipo resultante
Tipo referencial	Tipo simple
Tipo simple	Tipo referencial (Excepto al tipo String)
Null	Tipo simple
Cualquier tipo	Tipo boolean
Tipo boolean	Cualquier otro tipo (Excepto al tipo String)
Cualquier clase C	<ul style="list-style-type: none">- Cualquier clase R siempre que C no sea subclase de R y R no sea subclase de C (Excepto a la clase String)- Cualquier interface I si C es “final” y no implementa I- Cualquier vector (Excepto si C es de la clase Object)
Cualquier interface I	<ul style="list-style-type: none">- Cualquier clase C (Excepto si C es String) si es “final” y no implementa I- Cualquier interface K si I y K declaran métodos con la misma cabecera y distinto tipo de dato de retorno.
Cualquier vector	<ul style="list-style-type: none">- Cualquier interface excepto al interface Cloneable.- Cualquier otro vector si no hay conversión entre los tipos de los elementos (excepto la conversión a String).

C. LAS CLASES STRING Y STRINGBUFFER.

C.1 La clase String

C.1.1 Constructores de la clase String.

public String();

Construye un *string* vacío.

public String(byte bytes[]);

Construye un *string* a partir de un vector de bytes codificados dependiendo de la plataforma (por ejemplo, ASCII), por lo tanto, la longitud del vector no tiene porqué coincidir siempre con la del *string*.

Ejemplo: `byte bytes[]={65,66};`
 `String s = new String(bytes);` “AB”

public String(byte bytes[], int posición, int cantidad);

Construye un *string* a partir de un vector de bytes codificados dependiendo de la plataforma. Para ello toma el primer byte que se encuentra en la posición indicada como parámetro y tantos bytes del vector como se indica en el parámetro “*cantidad*”.

Ejemplo: `byte bytes[]={65,66,67,68,69};`
 `String s = new String(bytes, 1 , 3);` BCD

public String(char valor[])

Construye un *string* inicializado por un vector de caracteres (*valor*).

Ejemplo: `char c[] = {'H','o','l','a'};`
 `String s = new String(c);`

public String(char valor[], int posición, int cantidad);

Construye un *string* inicializado por un subvector de caracteres. El primer carácter del *string* será el indicado por el primer entero (*posición*),

Las clases `String` y `StringBuffer`.

teniendo en cuenta que los vectores comienzan por el índice cero. La longitud del string será especificado por el segundo entero (*cantidad*).

Ejemplo:

```
char c[] = {'H','o','l','a'};
String s = new String( c , 1 , 2 );
```

 “ol”

public String(String valor);

Construye un `String` que es copia del `String` especificado como parámetro.

Ejemplo:

```
String s = new String("Hola mundo");
```

public String(StringBuffer buffer);

Construye un nuevo `String` a partir de un `StringBuffer`.

C.1.2 Métodos de la clase `String`.

public char charAt(int posición);

Devuelve el carácter en la posición especificada por el entero que se pasa como parámetro, teniendo en cuenta que el primer elemento tiene el índice cero.

public int compareTo(String otroString);

Compara el string con el *string* que se pasa como parámetro, devolviendo 0 si son iguales, un número menor que cero si es menor, y un número mayor que cero si es mayor.

public String concat(String str);

Concatena el *string* especificado como parámetro al final de este *string*.

public static String copyValueOf(char datos[], int posición, int cantidad);

Devuelve un `String` equivalente al vector de caracteres con inicio en el primer entero y longitud del segundo entero.

public static String copyValueOf(char datos[]);

Equivalente al método anterior pero con el vector entero.

Las clases `String` y `StringBuffer`.

`public boolean endsWith(String sufijo);`

Devuelve `true` si el *string* tiene como sufijo el `String` que se pasa como parámetro y `false` en caso contrario.

`public boolean equals(Object objeto);`

Compara este `String` con el objeto especificado. Para devolver `true`, este objeto no debe ser `null` y debe ser un `String` con la misma secuencia de caracteres.

`public boolean equalsIgnoreCase(String otroString);`

Devuelve `true` si los dos `Strings` son iguales, pero antes de comparar, convierte el `String` a minúsculas.

`public byte[] getBytes();`

Copia caracteres del `String` en un vector de bytes utilizando la codificación dependiente de la plataforma.

`public void getChars(int inicioOrigen, int finOrigen, char destino[], int inicioDestino);`

Copia desde la posición `inicioOrigen` hasta `finOrigen` todos los caracteres del `String` y los guarda en el vector de caracteres destino empezando por la posición `inicioDestino`.

`public int hashCode();`

Devuelve el código *hash* para este `String`.

`public int indexOf(char ch);`

Devuelve el índice de la primera ocurrencia del carácter especificado.

`public int indexOf(char ch, int desdePosición);`

Similar al método anterior, pero empezando a buscar desde la posición indicada por el parámetro.

`public int indexOf(String subStr);`

Devuelve el índice de la primera ocurrencia del *substring* especificado dentro de este `String`.

Las clases `String` y `StringBuffer`.

`public int indexOf(String subStr, int desdePosición);`

Similar al método anterior pero se empieza a buscar a partir del índice especificado por el entero.

`public String intern();`

Devuelve un `String` que es igual a este `String` pero se garantiza que se incluye en el *pool* único de `Strings`.

`public int lastIndexOf(char ch);`

Devuelve el índice de la última ocurrencia del carácter especificado.

`public int lastIndexOf(char ch, int desdePosición);`

Similar al método anterior, pero empezando a buscar en la posición indicada.

`public int lastIndexOf(String subStr);`

Devuelve el índice de la última ocurrencia del substring especificado dentro de este `String`.

`public int lastIndexOf(String subStr, int desdePosición);`

Similar al método anterior pero se empieza a buscar a partir del índice especificado por el entero.

`public int length();`

Devuelve la longitud del string.

`public boolean regionMatches(int desdePosición, String otroStr, int desdePosiciónOtro, int longitud);`

Devuelve true si un substring coincide con otro substring que se pasa como parámetro y false en caso contrario.

El primer entero (`desdePosición`) es el índice de este *string* a partir del cual se va a comparar.

El segundo entero (`desdePosiciónOtro`) es el índice del *string* parámetro a partir del cual se va a comparar.

El tercer entero (`longitud`) es el número de caracteres que se van a comprobar.

Las clases `String` y `StringBuffer`.

public boolean regionMatches(boolean ignorarMayúsc, int desdePosición, String otroStr, int desdePosiciónOtro, int longitud);

Similar al anterior, si se pasa un valor `true` en `ignorarMayúsc`, se ignorarán mayúsculas y minúsculas y si es `false` no.

public String replace(char charAntiguo, char charNuevo);

Devuelve un `String` resultado de reemplazar todos los caracteres que coincidan con el primer parámetro por el segundo.

public boolean startsWith(String prefijo);

Devuelve `true` si el `string` tiene como prefijo el `String` que se pasa como parámetro y `false` en caso contrario.

public boolean startsWith(String prefijo, int desdePosición);

Devuelve `true` si el `string` tiene como prefijo el `String` que se pasa como parámetro y `false` en caso contrario, pero empezando por la posición indicada por el entero.

public String substring(int desdePosición);

Devuelve el substring que comienza en la posición indicada por el entero (inclusive).

public String substring(int desdePosición, int hastaPosición);

Devuelve el substring que comienza en la posición indicada por el primer entero (inclusive) hasta la posición indicada por el segundo entero (exclusive).

public char[] toCharArray();

Devuelve un vector de caracteres equivalente a este `string`.

public String toLowerCase();

Devuelve un `String` resultado de convertir todos los caracteres a minúsculas.

public String toLowerCase(Locale local);

Devuelve un `String` resultado de convertir todos los caracteres a minúsculas utilizando las reglas de conversión particulares del país (`locale`).

Las clases `String` y `StringBuffer`.

`public String toUpperCase();`

Devuelve un `String` resultado de convertir todos los caracteres a mayúsculas.

`public String toUpperCase(Locale local);`

Devuelve un `String` resultado de convertir todos los caracteres a mayúsculas utilizando las reglas de conversión particulares del país (`locale`).

`public String trim();`

Elimina los espacios al principio y al final del `String`.

`public static String valueOf(Object objeto);`

Devuelve un `String` resultado de evaluar el método `toString()` del objeto.

`public static String valueOf(char datos[]);`

Devuelve un `String` equivalente al vector de caracteres.

`public static String valueOf(char datos[], int desdePosición, int cantidad);`

Devuelve un `String` equivalente al vector de caracteres, comenzando por el primer entero y de la longitud especificada por el segundo entero.

`public static String valueOf(boolean b);`

Devuelve un `String` equivalente al valor `boolean` del parámetro.

`public static String valueOf(char ch);`

Devuelve un `String` equivalente al carácter.

`public static String valueOf(int i);`

Devuelve un `String` equivalente al entero.

`public static String valueOf(long l);`

Devuelve un `String` equivalente al entero largo.

`public static String valueOf(float f);`

Devuelve un `String` equivalente al real de tipo `float`.

Las clases `String` y `StringBuffer`.

`public static String valueOf(double d);`

Devuelve un `String` equivalente al real de tipo `double`.

C.2 La clase `StringBuffer`.

Si se quiere utilizar un `String` que pueda variar de tamaño o contenido, se deberá crear un objeto de la clase `StringBuffer`.

C.2.1 Constructores de la clase `StringBuffer`.

`public StringBuffer();`

Crea un `StringBuffer` vacío.

`public StringBuffer(int longitud);`

Crea un `StringBuffer` vacío de la longitud especificada por el parámetro.

`public StringBuffer(String str);`

Crea un `StringBuffer` con el valor inicial especificado por el `String`.

C.2.2 Métodos de la clase `StringBuffer`.

`public synchronized StringBuffer append(Object objeto)`

Añade un objeto al final del `StringBuffer`.

`public synchronized StringBuffer append(String str);`

Añade un `String` al final del `StringBuffer`.

`public synchronized StringBuffer append(char str[]);`

Añade un vector de caracteres al final del `StringBuffer`.

Las clases `String` y `StringBuffer`.

`public synchronized StringBuffer append(char str[],int desdePosición, int cantidad);`

Añade parte de un vector de caracteres al final del `StringBuffer`.

El primer entero es el índice del primer elemento del vector.

El segundo entero es la longitud del subvector.

`public StringBuffer append(boolean b);`

Añade un valor boolean al final del `StringBuffer`.

`public synchronized StringBuffer append(char ch);`

Añade un carácter al final del `StringBuffer`.

`public StringBuffer append(int i);`

Añade un entero al final del `StringBuffer`.

`public StringBuffer append(long l);`

Añade un entero largo al final del `StringBuffer`.

`public StringBuffer append(float f);`

Añade un real de tipo float al final del `StringBuffer`.

`public StringBuffer append(double d);`

Añade un real de tipo `double` al final del `StringBuffer`.

`public int capacity();`

Devuelve la capacidad actual del `StringBuffer`.

`public synchronized char charAt(int posición);`

Devuelve el carácter especificado por el índice que se pasa como parámetro.

`public synchronized void ensureCapacity(int capacidadMínima);`

Asegura que la capacidad del `StringBuffer` sea como mínimo la especificada como parámetro.

Las clases String y StringBuffer.

public synchronized void getChars(int inicioOrigen, int finOrigen, char destino[],int inicioDestino);

Copia caracteres del `StringBuffer` en un vector de caracteres.

El primer entero (`inicioOrigen`) es el índice del primer carácter del `StringBuffer`.

El segundo entero (`finOrigen`) es el índice del último carácter del `stringBuffer`.

El tercer entero (`inicioDestino`) es el índice del primer elemento del vector de caracteres que se llenará.

public synchronized StringBuffer insert(int posición, Object objeto);

Inserta un objeto en la posición especificada por el entero.

public synchronized StringBuffer insert(int posición, String str);

Inserta un `String` en la posición especificada por el entero.

public synchronized StringBuffer insert(int posición, char str[]);

Inserta un vector de caracteres en la posición especificada por el entero.

public StringBuffer insert(int posición, boolean b);

Inserta un valor `boolean` en la posición especificada por el entero.

public synchronized StringBuffer insert(int posición, char ch);

Inserta un carácter en la posición especificada por el entero.

public StringBuffer insert(int posición, int i);

Inserta un entero (`int`) en la posición especificada por el entero.

public StringBuffer insert(int posición, long l);

Inserta un entero (`long`) en la posición especificada por el entero.

public StringBuffer insert(int posición, float f);

Inserta un real (`float`) en la posición especificada por el entero.

Las clases `String` y `StringBuffer`.

`public StringBuffer insert(int posición, double d);`

Inserta un real (`double`) en la posición especificada por el entero.

`public int length();`

Devuelve el número de caracteres que tiene el `StringBuffer`.

`public synchronized StringBuffer reverse();`

Invierte el `StringBuffer`.

`public synchronized void setCharAt(int posición, char ch);`

Cambia el carácter en la posición especificada por el entero por el carácter especificado como parámetro.

`public synchronized void setLength(int nuevaLongitud);`

Cambia la longitud del `StringBuffer` por el entero que se pasa como parámetro. Si la longitud nueva es menor que la anterior, los caracteres del final se pierden. Si la nueva longitud es mayor que la anterior, los nuevos caracteres se inicializan a cero.

D. CLASES ANIDADAS.

Las clases anidadas o interiores (*inner classes*) son nuevas en el JDK 1.1. Hasta entonces, en la versión 1.02, las clases únicamente podían ser declaradas en el primer nivel de forma global como miembros del paquete al que pertenecen.

A partir del JDK 1.1, las clases pueden ser declaradas dentro de los siguientes ámbitos del programa:

Como miembros de otras clases, al nivel de los atributos.

Como clase local dentro de un método.

Dentro de un bloque de sentencias parentizado por las llaves { ... }.

Dentro de una expresión (como clase anónima).

Las clases anidadas son el resultado de la combinación de la POO basada en clases con la programación estructurada en bloques de código.

Así, a partir del JDK 1.1 se pueden encontrar clases anidadas con la siguiente estructura:

```
class Clase1 {
    ...
}

class Clase2 {
    class Clase2_1 {
        ...
    }
    class Clase2_2 {
        ...
    }
    ...
}

...
class ClaseN {
    ...
}
```

Y estas clases interiores pueden, a su vez, contener otras clases anidadas dentro de ellas.

Clases anidadas.

Para acceder al constructor de las clases interiores desde fuera de la clase en la cual están declaradas es necesario, de forma general (a no ser que éstas sean `static`), crear un objeto de la clase de primer nivel y después utilizar la palabra reservada `new` cualificada por el nombre del objeto:

```
Clase2 c2=new Clase2();  
Clase2_1 c2_1=c2.new Clase2_1();
```

Las clases interiores tienen algunas restricciones:

No pueden contener atributos `static`.

D.1 Utilidad de las clases anidadas.

Las clases anidadas no son estrictamente necesarias. En realidad, el compilador realiza una traducción a código para la JVM 1.0 (versión sin clases anidadas). Únicamente son útiles debido a la simplificación de la tarea de programación por las siguientes razones:

Las clases interiores no pueden usarse, en principio, fuera del ámbito en el que están declaradas, a excepción del uso de nombres cualificados de la forma: objetoDeLaClase2.Clase2_1. Esto consigue una mayor estructuración de las clases en un paquete.

Los identificadores utilizados dentro de una clase interior no necesitan ser cualificados ni para los atributos y métodos de la clase interior ni para los pertenecientes a la estructura que encierra a la clase interior. Así, por ejemplo, desde la clase Clase2_1 podría accederse a un atributo (incluso privado) de la clase Clase1 simplemente utilizando el identificador de este atributo (sin cualificar de la forma Clase1.atributo).

Se consigue eliminar las restricciones del lugar de declaración de las clases, de forma que las reglas de ámbito se hacen más generales y similares a las estructuras de bloques anidados de lenguajes estructurados como el Pascal.

D.2 Clases anidadas como miembros de otras clases.

Este tipo de anidamiento se corresponde con la forma:

```
class Clase1 {
    class Clase1_1 {
        ...
    }
    class Clase1_2 {
        ...
    }
    ...
}
```

En este caso, desde el interior de las clases Clase1_1 y Clase1_2 se puede acceder a los métodos y atributos de la clase Clase1 sin necesidad de ningún tratamiento especial, sino tal y como puede hacerse desde cualquier método de Clase1.

Los modificadores para las clases interiores (miembros de otras clases) son los mismos que los aplicables a los atributos. Es decir, además de los modificadores vistos para las clases de primer nivel `abstract`, `final` y `static`, pueden utilizarse los modificadores `private` y `protected`.

El resto de clases anidadas que no son miembros de una clase (declarados al nivel de los atributos) no pueden utilizar los modificadores `public`, `private`, `protected` o `static`, ya que no son miembros de la clase.

El efecto de declarar una clase anidada `static` es el de indicar que es una clase de primer nivel (únicamente se incluye anidada dentro de otra por motivos de organización), y, por tanto, puede utilizarse como cualquier otra clase de primer nivel simplemente cualificándola anteponiéndole el nombre de la clase en la cual está incluida.

```
class Clase1 {
    static class Clase1_1 {
        ...
    }
    ...
}
```

Puede utilizarse Clase1_1 de la siguiente forma:

Clases anidadas.

```
objeto1_1 = new Clase1.Clase1_1();
```



```
interface Hablador {
    public void habla();
}

class Granja {
    // numAnimales contiene el número de animales
    // que hay en la granja.
    private static int numAnimales = 0;
    private static int numCerdos=0;
    class Cerdo implements Hablador {
        public Cerdo() {
            numAnimales+=2;
            // Los cerdos valen por dos.
            // Todos los animales son iguales,
            // pero algunos son más iguales
            // que otros.
            numCerdos++;
        }
        public void habla() {
            System.out.println("Soy un cerdo");
        }
        protected void finalize() {
            numAnimales-=2;
            numCerdos--;
        }
    }
    class Gallina implements Hablador {
        public Gallina() {
            numAnimales++;
        }
        public void habla() {
            System.out.println("Soy una gallina");
        }
        protected void finalize() {
            numAnimales--;
        }
    }
    static class Caballo implements Hablador {
        public Caballo() {
            numAnimales++;
        }
        public void habla() {
            System.out.println("Soy un caballo");
            System.out.println(
                "    No necesito una granja, soy STATIC");
        }
        protected void finalize() {
```

Clases anidadas.

```
        numAnimales--;
    }
}
private class Oveja implements Hablador {
    public Oveja() {
        numAnimales++;
    }
    public void habla() {
        System.out.println("Soy una Oveja");
        System.out.println("    Soy tímida (PRIVATE)");
    }
    protected void finalize() {
        numAnimales--;
    }
}
public int getNumAnimales() {
    return numAnimales;
}
public int getNumCerdos() {
    return numCerdos;
}
public Oveja getOveja() {
    return new Oveja();
}
}

class Principal {
    public static void main(String arg[]) {
        Granja g = new Granja();
        Hablador h1,h2,h3,h4;
        h1=g.new Cerdo();
        h2=g.new Gallina();
        h3=new Granja.Caballo();
        h4=g.getOveja();
        h1.habla();
        h2.habla();
        h3.habla();
        h4.habla();
        System.out.println("\nHay "+g.getNumAnimales()+
            " animales en la granja");
        System.out.println("Hay "+g.getNumCerdos()+
            " cerdos");
    }
}
```

Clases anidadas.

El resultado de ejecutar el programa anterior es el siguiente:

```
Soy un cerdo
Soy una gallina
Soy un caballo
No necesito una granja, soy STATIC
Soy una Oveja
    Soy tímida (PRIVATE)

Hay 5 animales en la granja
Hay 1 cerdos
```

Puede observarse que el compilador a creado las siguientes clases (ficheros .class):

```
Hablador.class
Granja.class
Principal.class
Granja$Cerdo.class
Granja$Gallina.class
Granja$Caballo.class
Granja$Oveja.class
```

Las cuatro últimas se distinguen de las anteriores por estar calificadas por el nombre de la clase en la que están contenidas y el símbolo \$. Como se dijo en el punto anterior, el compilador realiza una traducción a código compatible con la versión 1.0 de la JVM.

En el ejemplo anterior se han declarado 4 clases interiores a la clase **Granja**. Estas clases son **Cerdo**, **Gallina**, **Caballo** y **Oveja**. Todas ellas acceden al atributo de clase **numAnimales** de la clase que las contiene. Esto no supone ningún problema, incluso como es el caso, si este atributo es `private` (además, la clase **Cerdo** accede también al atributo **numCerdos**). Todas ellas implementan la interface **Hablador**, lo que significa que deben redefinir el método **habla()**.

Veamos las particularidades de cada una de las clases interiores:

La clase **Cerdo** no tiene ningún modificador (modificador por defecto). Es accesible desde cualquier clase perteneciente al paquete (en este caso, el paquete por defecto ya que no se ha declarado

Clases anidadas.

ninguno). En el método `main()` de la clase **Principal** es creada una instancia de ésta⁵². Debido a que no es `static`, necesita de una instancia de la clase que la contiene (**Granja**) para poder existir: `Granja g = new Granja();` y después es creada la instancia de **Cerdo** mediante: `h1=g.new Cerdo();`

La clase **Gallina** es, a efectos de programación, igual que la anterior.

La clase **Caballo** es `static`, por lo tanto no necesita de la clase **Granja** para poder crear una instancia de la misma. Es lo mismo que si fuera una clase de primer nivel. Únicamente se diferencia de éstas en que es necesario anteponer el nombre de la clase en la que está declarada para llamar al constructor: `h3=new Granja.Caballo();` , aunque también se podría haber utilizado el objeto `g` de la clase **Granja** creado haciendo: `h3=g.new Caballo();` al igual que se ha hecho con las clases anteriores.

La clase **Oveja** es privada (`private`). No es accesible por clases distintas a la que la contiene (**Granja**), pero la clase **Granja** posee un método que devuelve instancias de **Oveja** (`getOveja()`). Así puede crearse una instancia de **Oveja** (o referencia a **Hablador**) mediante: `h4=g.getOveja();`

Puede observarse que existe un atributo (`numCerdos`) con el modificador `static`, que sirve, en el ejemplo como contador del número de instancias de la clase **Cerdo**. Puede parecer más adecuado y elegante declarar este atributo dentro de la clase **Cerdo**, pero como ya se ha dicho anteriormente: una clase interior no puede tener atributos `static`.

D.3 Clases interiores a un bloque o locales a un método.

Pueden declararse clases interiores dentro de un método o incluso dentro de un bloque parentizado por llaves (`{ ... }`). Trataremos estas últimas ya que el primer caso es una particularidad del segundo.

⁵² En realidad se utiliza la interface **Hablador** (`Hablador h1,h2,h3,h4;`). Se realiza una conversión de tipos. ↪ Véase: B. **Conversión de tipos**. En la página 273.

Clases anidadas.

Estas clases pueden acceder a todas las variables u objetos pertenecientes al método y a los atributos de la clase que las contiene; igual que las clases anidadas miembros de otras clases del punto anterior y además, a las variables locales del método.

Únicamente son visibles dentro del bloque entre llaves ({ ... }) en el que están declaradas y, por lo tanto, no pueden tener modificadores como `private`, `public`, `static` o `protected`. No se puede acceder a ellas fuera del bloque en el que están declaradas.

Veamos una modificación del ejemplo del punto anterior:



```
interface Hablador {
    public void habla();
}

class Granja {
    // numAnimales contiene el número de animales
    // que hay en la granja.
    private static int numAnimales = 0;
    private static int numCerdos=0;
    class Cerdo implements Hablador {
        public Cerdo() {
            numAnimales+=2;
            // Los cerdos valen por dos.
            // Todos los animales son iguales,
            // pero algunos son más iguales
            // que otros.
            numCerdos++;
        }
        public void habla() {
            System.out.println("Soy un cerdo");
        }
        protected void finalize() {
            numAnimales-=2;
            numCerdos--;
        }
    }
    class Gallina implements Hablador {
        public Gallina() {
            numAnimales++;
        }
        public void habla() {
            System.out.println("Soy una gallina");
        }
    }
}
```

Clases anidadas.

```
        protected void finalize() {
            numAnimales--;
        }
    }
    static class Caballo implements Hablador {
        public Caballo() {
            numAnimales++;
        }
        public void habla() {
            System.out.println("Soy un caballo");
            System.out.println(
                "    No necesito una granja, soy STATIC");
        }
        protected void finalize() {
            numAnimales--;
        }
    }
    public int getNumAnimales() {
        return numAnimales;
    }
    public int getNumCerdos() {
        return numCerdos;
    }
    public Hablador getOveja() {
        class Oveja implements Hablador {
            public Oveja() {
                numAnimales++;
            }
            public void habla() {
                System.out.println("Soy una Oveja");
                System.out.println(" Soy tímida (PRIVATE)");
            }
            protected void finalize() {
                numAnimales--;
            }
        }; // OJO con el punto y coma. Marca fin de la
            // declaración de Oveja (como cualquier otra
            // variable local).
        return new Oveja();
    }
}

class Principal2 {
    public static void main(String arg[]) {
        Granja g = new Granja();
        Hablador h1,h2,h3,h4;
        h1=g.new Cerdo();
        h2=g.new Gallina();
        h3=new Granja.Caballo();
        h4=g.getOveja();
        h1.habla();
        h2.habla();
    }
}
```

Clases anidadas.

```
h3.habla();
h4.habla();
System.out.println("\nHay "+g.getNumAnimales()+
                    " animales en la granja");
System.out.println("Hay "+g.getNumCerdos()+
                    " cerdos");
    }
}
```

La modificación se encuentra en negrita. Se ha trasladado la declaración de la clase **Oveja** al interior del método **getOveja()**. Éste método también ha sido modificado en su declaración; antes devolvía un objeto de la clase **Oveja** y ahora devuelve un **Hablador**. Esta modificación ha sido necesaria porque la clase **Oveja** sólo es visible “dentro” del método **getOveja()** y ni siquiera lo es en la declaración de la signatura del método.

D.4 Clases anónimas.

Las clases anónimas son aquellas que se declaran como parte de cualquier expresión. Se utilizan para crear clases interiores a un bloque sin necesidad de darles un nombre. En el ejemplo anterior, de hecho, la clase **Oveja** únicamente se utiliza para que el método **getOveja()** devuelva una instancia de **Hablador**. No es necesario un identificador de la clase ya que no se utiliza en ningún otro lugar del código.

Todas las restricciones propias de las clases interiores a un bloque son aplicables a las clases anónimas. Éstas tienen además la restricción de que **no pueden tener constructores**. En el caso de existir parámetros en la llamada **new**, éstos serán pasados al constructor de la superclase.

El uso de clases anónimas puede facilitar la legibilidad del código fuente siempre que éstas sean simples y consten de unas pocas líneas de código, de lo contrario puede ser más adecuado el uso de las clases interiores vistas en los puntos precedentes. También la restricción de no poder tener constructores puede ser un inconveniente insalvable.

Podría modificarse el método **getOveja()** del ejemplo de la siguiente forma:

Clases anidadas.



```
...
public Hablador getOveja() {
    // este es el único caso en que el nombre de una
    // interface puede seguir a la palabra reservada new.
    return new Hablador() {
        public void habla() {
            System.out.println("Soy una Oveja");
            System.out.println(" Soy tímida (ANÓNIMA)");
        }
    }; // OJO con el punto y coma marca el fin de la
        // expresión return.
}
...
```

Se ha eliminado el constructor y, por lo tanto, también el método **finalize()**. No porque no se pueda acceder al atributo **numAnimales** sino porque está prohibido el uso de constructores.

En este caso, en lugar de generarse la clase **Granja\$Oveja.class** se ha creado la clase **Granja\$1.class**. El compilador numera las distintas clases anónimas que pudieran existir.

E. CLASES SOBRE STREAMS Y FICHEROS.

E.1 La clase `InputStream`

E.1.1 Constructor.

public `InputStream()`;

Es el constructor por defecto. Sólo es llamado por las subclases.

E.1.2 Métodos.

public int `available()` throws `IOException`;

Determina el número de bytes que pueden ser leídos de este `InputStream` sin bloquearse. El método `available()` de `InputStream` devuelve 0. Este método debería ser redefinido por las subclases.

Devuelve:

el número de bytes que pueden ser leídos de este `InputStream` sin bloquearse.

Throws `IOException` Si ocurre un error de E/S.

public void `close()` throws `IOException`;

Cierra este `InputStream` y libera todos los recursos del sistema asociados al *stream*. El método `close()` de `InputStream` no hace nada, hay que redefinirlo.

Throws `IOException` Si ocurre un error de E/S.

Clases sobre streams y ficheros.

public void mark(int readlimit);

Marca la posición actual en este `InputStream`. Una posterior llamada al método `reset()` reposiciona este *stream* en la última posición marcada para que posteriores lecturas releen los mismos bytes.

El método `mark()` de `InputStream` no hace nada.

Parámetro:

`readlimit` - el límite máximo de bytes que pueden ser leídos antes de que la marca de posición quede invalidada.

public boolean markSupported();

Determina si este `InputStream` soporta los métodos `mark()` y `reset()`. El método `markSupported` de `InputStream` devuelve false.

Devuelve:

true si soporta los métodos `mark()` y `reset()`; y false en caso contrario.

public abstract int read() throws IOException;

Lee el siguiente byte de datos desde este `InputStream`. El valor es devuelto como un entero (`int`) en el rango 0 .. 255. Si no hay ningún byte disponible porque se ha llegado al final del *stream*, devuelve el valor -1 . Este método se bloquea hasta que haya datos disponibles , que se detecte el final del *stream* o se lance una excepción.

Una subclase debe proporcionar una implementación de este método.

Devuelve:

el siguiente byte de datos, ó -1 si se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

public int read(byte b[]) throws IOException;

Lee hasta `b.length` bytes de datos desde este `InputStream` y los almacena en un vector de bytes.

Clases sobre streams y ficheros.

El método `read()` de `InputStream` llama al método `read()` de tres parámetros con los parámetros `b`, `0`, y `b.length`.

Parámetros:

`b` - el buffer en el cual se almacenan los datos leídos

Devuelve:

el número total de bytes leídos y almacenados en el *buffer*, ó -1 si no hay más datos porque se ha llegado al final del *stream*.

Throws `IOException` si ocurre un error de E/S.

`public int read(byte b[], int desplaza, int longitud) throws IOException;`

Lee hasta `longitud` bytes de datos (o hasta llegar a la marca de final de fichero `^Z` o leer el carácter de nueva línea) desde este `InputStream` y los almacena en un vector de bytes. Este método se bloquea hasta que haya datos disponibles. Si el primer parámetro es `null`, se leen y borran hasta `longitud` bytes de datos.

El método `read()` de `InputStream` lee un solo byte de una sola vez utilizando el método `read()` de cero parámetros para llenar el vector. Las subclases deberían proporcionar una implementación más eficiente de este método.

Parámetros:

`b` - el buffer en el cuál se almacenan los datos leídos

`desplaza` - el desplazamiento inicial de los datos

`longitud` - el número máximo de bytes a leer

Devuelve:

el número total de bytes leídos y almacenados en el buffer, ó -1 si no hay más datos porque se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

Clases sobre streams y ficheros.

public void reset() throws IOException;

Reposiciona este `InputStream` en la última marca que se realizó mediante la llamada al método `mark()` en este `InputStream`

El método `reset()` de `InputStream` lanza una excepción, ya que los `InputStreams`, por defecto, no soportan los métodos `mark()` y `reset()`.

Throws `IOException` si este *stream* no ha sido marcado mediante el método `mark()` o la marca ha sido invalidada.

public long skip(long n) throws IOException

Salta y borra `n` bytes de datos de este `InputStream`. Es posible que el método `skip()` no salte los `n` bytes, sino un número menor o incluso cero. El método devuelve el número real de bytes saltados.

El método `skip()` de `InputStream` crea un vector de bytes de longitud `n`, lee `n` bytes o hasta llegar al final del *stream*, almacenándolos en el vector. Las subclases deberían proporcionar una implementación más eficiente de este método.

Parámetros:

`n` - el número de bytes a saltar

Devuelve:

el número real de bytes saltados.

Throws `IOException` si ocurre un error de E/S.

E.2 La clase *OutputStream*

E.2.1 Constructor.

public OutputStream();

Es el constructor por defecto.

E.2.2 Métodos.

public void close() throws IOException;

Cierra este `OutputStream` y libera todos los recursos del sistema asociados con éste.

El método `close()` de `OutputStream` por defecto no hace nada. Es necesario redefinirlo en la subclase para darle funcionalidad.

Throws `IOException` si ocurre un error de E/S.

public void flush() throws IOException;

Hace que se escriban todos los bytes de salida pendientes.

El método `flush()` de `OutputStream` no hace nada.

Throws `IOException` si ocurre un error de E/S.

public void write(byte b[]) throws IOException

Escribe `b.length()` bytes del vector de bytes especificado en este `OutputStream`.

El método `write` de `OutputStream` llama al método `write` de tres parámetros con los valores `b`, `0`, y `b.length()`.

Parámetro:

`b` - los datos

Throws `IOException` si ocurre un error de E/S.

public void write(byte b[], int desplaza, int longitud) throws IOException

Escribe `longitud` bytes desde el vector de bytes especificado, empezando en la posición `desplaza`, hacia este `OutputStream`.

El método `write` de `OutputStream` llama al método `write` de un parámetro para cada uno de los bytes que se van a escribir. Las subclases deberían redefinir este método para proporcionar una implementación más eficiente.

Clases sobre streams y ficheros.

Parámetros:

b - los datos

desplaza - la posición de comienzo en el vector de datos

longitud - el número de bytes a escribir

Throws `IOException` si ocurre un error de E/S.

public abstract void write(int b) throws IOException

Escribe el byte especificado hacia este `OutputStream`.

Las subclases de `OutputStream` deben proporcionar una implementación para este método.

Parámetro:

b - el byte

Throws `IOException` si ocurre un error de E/S.

E.3 La clase `PrintStream`.

E.3.1 Constructores.

public `PrintStream`(`OutputStream` out);

public `PrintStream`(`OutputStream` out, boolean autoFlush);

Ambos constructores crean un nuevo *stream* de escritura.

Parámetros:

out – Es el `OutputStream` (*stream* de salida) sobre el cuál se escribirá.

autoFlush – Si su valor es `true`, se vaciará el *buffer* (*flush*) cuando se llegue al final de la línea o se escriba el carácter ‘\n’.

E.3.2 Métodos.

public boolean checkError();

Vacía el *buffer (flush)* y comprueba su estado de error. Si se ha producido un error, el método devolverá `true`, incluso en sucesivas llamadas si existen varios errores acumulados.

Devuelve:

`true` si se ha producido algún error y `false` en caso contrario.

public void close();

Cierra el *stream*. En primer lugar realiza un vaciado del *buffer (flush)* y a continuación cierra el `OutputStream` asociado.

public void flush();

Vacía el *buffer* enviando los bytes pendientes al `OutputStream` asociado.

public void print(boolean b);

public void print(char c);

public void print(char s[]);

public void print(int i);

public void print(long l);

public void print(float f);

public void print(double d);

public void print(String s);

public void print(Object obj);

public void println();

public void println(boolean b);

public void println(char c);

public void println(char s[]);

public void println(int i);

public void println(long l);

public void println(float f);

public void println(double d);

public void println(String s);

public void println(Object obj);

En todos los casos se escribe el resultado de llamar al método `toString()` del parámetro `o`, en caso de ser tipos de datos simples, de su clase equivalente (por ejemplo `int Integer`), y convertir los caracteres a bytes codificados dependiendo de la plataforma subyacente.

Clases sobre streams y ficheros.

El método `println()`, a diferencia del método `print()` escribe un separador de línea después. Este separador de línea no tiene que ser necesariamente el carácter `'\n'`. Depende de la propiedad del sistema `line.separator`.

El método `println()` sin parámetros únicamente escribe un separador de línea.

protected void setError();

Especifica explícitamente el estado de error del *stream*, de forma que el método `checkError()` devolverá `true`.

public void write(int b);

Escribe el byte indicado por el parámetro `int` en el `OutputStream`. En este caso no hay ningún tipo de conversión. Si se desea convertir un carácter a byte en la codificación propia de la plataforma donde se ejecuta hay que utilizar el método `print(char c)`, que realiza dicha transformación.

public void write(byte b[], int desplaza, int longitud);

Escribe `longitud` bytes desde el vector de bytes especificado, empezando en la posición `desplaza`, hacia este `OutputStream`.

E.4 La clase Reader.

E.4.1 Atributo.

protected Object lock;

Objeto sobre el que se realizan bloqueos para sincronizar operaciones sobre este *stream*. Este objeto puede ser el propio *stream* (`this`) u otro objeto.

E.4.2 Constructores.

protected Reader();

protected Reader(Object lock);

En el primer caso, cuando se realicen operaciones en zonas críticas se utilizará el propio *stream* para realizar el bloqueo, y en el segundo caso será sobre el objeto que se suministra como parámetro.

E.4.3 Métodos.

public abstract void close() throws IOException;

Cierra el *stream*.

Throws `IOException` si ocurre un error de E/S.

public void mark(int readlimit) throws IOException;

Marca la posición actual en este `Reader`. Una posterior llamada al método `reset()` reposiciona este *stream* en la última posición marcada para que posteriores lecturas releen los mismos caracteres.

Parámetro:

`readlimit` - el límite máximo de caracteres que pueden ser leídos antes de que la marca de posición quede invalidada.

Throws `IOException` si ocurre un error de E/S.

public boolean markSupported();

Determina si este `Reader` soporta los métodos `mark()` y `reset()`.

Devuelve:

`true` si soporta los métodos `mark()` y `reset()`; y `false` en caso contrario.

Clases sobre streams y ficheros.

public int read() throws IOException;

Lee el siguiente carácter. El valor es devuelto como un entero (`int`) en el rango 0 .. 16383 (0x00 .. 0xffff). Si no hay ningún carácter disponible porque se ha llegado al final del *stream*, devuelve el valor -1 . Este método se bloquea hasta que haya algún carácter disponible, que se detecte el final del *stream* o se lance una excepción.

Devuelve:

el siguiente carácter, ó -1 si se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

public int read(char b[]) throws IOException;

Igual que el método anterior pero en este caso, se almacenan los caracteres leídos en un vector de caracteres que se pasa como parámetro.

Devuelve:

el número de caracteres leído, ó -1 si se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

public int read(char b[], int desplaza, int longitud) throws IOException;

Lee hasta `longitud` caracteres, hasta llegar a la marca de final de fichero, o hasta que se produzca una excepción, y los almacena en un vector de caracteres que se pasa como parámetro. Este método se bloquea hasta que haya datos disponibles.

Parámetros:

`b` - el buffer en el cuál se almacenan los caracteres leídos

`desplaza` - el desplazamiento inicial de los datos

`longitud` - el número máximo de caracteres a leer

Devuelve:

el número total de caracteres leídos y almacenados en el *buffer*, ó -1 si no hay más datos porque se ha alcanzado el final del *stream*.

Clases sobre streams y ficheros.

Throws `IOException` si ocurre un error de E/S.

public boolean ready() throws IOException;

Indica si este *stream* está preparado para la lectura.

Devuelve:

`true` si está listo y `false` en caso contrario.

Throws `IOException` si ocurre un error de E/S.

public void reset() throws IOException;

Reposiciona este *stream* en la última marca que se realizó mediante la llamada al método `mark()` en este `Reader`.

Throws `IOException` si este *stream* no ha sido marcado mediante el método `mark()`, la marca ha sido invalidada, el *stream* no acepta `reset()` o se produce cualquier error de e/s.

public long skip(long n) throws IOException

Salta `n` caracteres. Es posible que el método `skip()` no salte los `n` caracteres, sino un número menor o incluso cero. El método devuelve el número real de caracteres saltados.

Parámetro:

`n` - el número de caracteres a saltar

Devuelve:

el número real de caracteres saltados.

Throws `IOException` si ocurre un error de E/S.

E.5 La clase *Writer*.

E.5.1 Atributo.

protected Object lock;

Objeto sobre el que se realizan bloqueos para sincronizar operaciones sobre este *stream*. Este objeto puede ser el propio *stream* ([this](#)) u otro objeto.

E.5.2 Constructores.

protected Writer();

protected Writer(Object lock);

En el primer caso, cuando se realicen operaciones en zonas críticas se utilizará el propio *stream* para realizar el bloqueo, y en el segundo caso será sobre el objeto que se suministra como parámetro.

E.5.3 Métodos.

public abstract void close() throws IOException;

Cierra el *stream* después de haber vaciado el buffer (*flush*).

Throws [IOException](#) si ocurre un error de E/S.

public abstract void flush() throws IOException;

Vacía el *buffer* de salida

Throws [IOException](#) si ocurre un error de E/S.

public void write(int c) throws IOException

Escribe un carácter sobre el *stream* de salida. Ese carácter es el indicado por el código Unicode que se pasa como parámetro *c*, ignorando los 2 bytes de mayor orden y tomando los 2 bytes de menor orden.

Clases sobre streams y ficheros.

Throws `IOException` si ocurre un error de E/S.

public void write(char b[]) throws IOException;

Envía el vector de caracteres especificado como parámetro al *stream* de salida.

Throws `IOException` si ocurre un error de E/S.

public void write(char b[], int desplaza, int longitud) throws IOException;

Escribe `longitud` caracteres desde el vector de caracteres especificado, empezando en la posición `desplaza`, hacia este *stream* de salida.

Parámetros:

`b` - los datos

`desplaza` - la posición de comienzo en el vector de datos

`longitud` - el número de bytes a escribir

Throws `IOException` si ocurre un error de E/S.

public void write(String s) throws IOException;

Escribe un `String` sobre el *stream* de salida.

Throws `IOException` si ocurre un error de E/S.

public void write(String s, int desplaza, int longitud) throws IOException;

Escribe `longitud` caracteres del `String s`, empezando en la posición `desplaza`, en el *stream* de salida.

Parámetros:

`s` – el `String` a escribir

`desplaza` - la posición de comienzo en `String`

`longitud` - el número de caracteres a escribir

Throws `IOException` si ocurre un error de E/S.

E.6 La clase *BufferedReader*.

E.6.1 Constructores.

public *BufferedReader*(*Reader* in);

public *BufferedReader*(*Reader* in, int tamaño);

Ambos métodos crean un *stream* con *buffer* de entrada de caracteres a partir de un lector (*Reader*). El parámetro **tamaño** especifica el tamaño del *buffer*.

En el segundo caso, puede producirse la siguiente excepción:

IllegalArgumentException: Si tamaño <=0.

E.6.2 Métodos.

public void close() throws *IOException*;

Cierra el *stream*.

Throws *IOException* si ocurre un error de E/S.

public void mark(int readlimit) throws *IOException*;

Marca la posición actual en este *BufferedReader*. Una posterior llamada al método *reset()* reposiciona este *stream* en la última posición marcada para que posteriores lecturas releen los mismos caracteres.

El parámetro **readlimit** indica a este *BufferedReader* cuántos bytes puede leer antes de que la marca de posición quede invalidada.

Parámetro:

readlimit - el límite máximo de bytes que pueden ser leídos antes de que la marca de posición quede invalidada.

Throws *IOException* si ocurre un error de E/S.

IllegalArgumentExceptions si readlimit <=0.

Clases sobre streams y ficheros.

public boolean markSupported();

Determina si este `BufferedReader` soporta los métodos `mark()` y `reset()`.

Devuelve:

`true` si soporta los métodos `mark()` y `reset()`; y `false` en caso contrario.

public int read() throws IOException;

Lee el siguiente carácter del *stream*.

Throws `IOException` si ocurre un error de E/S.

public int read(char b[], int desplaza, int longitud) throws IOException;

Lee hasta `longitud` caracteres del *stream* (o hasta llegar a la marca de final de fichero `^Z` o leer el carácter de nueva línea) desde este `BufferedReader` y los almacena en un vector de caracteres. Este método se bloquea hasta que haya datos disponibles.

Parámetros:

`b` - el buffer (vector de caracteres) en el cuál se almacenan los datos leídos

`desplaza` - el desplazamiento inicial de los datos

`longitud` - el número máximo de bytes a leer

Devuelve:

el número total de bytes leídos y almacenados en el *buffer*, ó -1 si no hay más datos porque se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

public String readLine() throws IOException;

Lee una línea de texto del *stream* de entrada y la devuelve. El final de la línea es interpretado al leer el carácter de cambio de línea `'\n'` o el retorno de carro `'\r'`, o éste seguido del anterior.

Devuelve:

Clases sobre streams y ficheros.

El String correspondiente a la línea leída (sin el separador de línea).

Throws `IOException` si ocurre un error de E/S.

public boolean ready() throws IOException;

Indica si el *stream* de caracteres se encuentra listo para la lectura.

Devuelve:

`true` si está listo y `false` en caso contrario.

Throws `IOException` si ocurre un error de E/S.

public void reset() throws IOException;

Reposiciona este `BufferedReader` en la última marca que se realizó mediante la llamada al método `mark()` en este `BufferedReader`

Throws `IOException` si este *stream* no ha sido marcado mediante el método `mark()` o la marca ha sido invalidada.

public long skip(long n) throws IOException;

Se “salta”, e ignora, `n` caracteres del *stream*.

Throws `IOException` si ocurre un error de E/S.

E.6.3 La clase `PrintWriter`.

E.6.3.1 Constructores.

public PrintWriter(OutputStream out);

public PrintWriter(OutputStream out, boolean autoFlush);

public PrintWriter(Writer outW);

public PrintWriter(Writer outW, boolean autoFlush);

Los cuatro constructores crean un *stream* de salida de caracteres con *buffer*.

Clases sobre streams y ficheros.

Parámetros:

out – `OutputStream` hacia el que se dirige la salida de caracteres.

outW – `Writer` hacia el que se dirige la salida de caracteres.

autoFlush – si su valor es `true`, el método `println()` llamará al método `flush()` automáticamente

E.6.3.2 Métodos.

public boolean checkError();

Vacía el *buffer* de caracteres y comprueba el estado de error del *stream*.

Devuelve:

`true` si se produce un error y `false` en caso contrario.

public void close();

Cierra el *stream*.

public void flush();

Vacía el *buffer* enviando los caracteres pendientes al `OutputStream` o `Writer` asociado.

Clases sobre streams y ficheros.

public void print(boolean b);	public void println();
public void print(char c);	public void println(boolean b);
public void print(char s[]);	public void println(char c);
public void print(int i);	public void println(char s[]);
public void print(long l);	public void println(int i);
public void print(float f);	public void println(long l);
public void print(double d);	public void println(float f);
public void print(String s);	public void println(double d);
public void print(Object obj);	public void println(String s);
	public void println(Object obj);

En todos los casos se escribe el resultado de llamar al método `toString()` del parámetro `o`, en caso de ser tipos de datos simples, de su clase equivalente (por ejemplo `int Integer`).

El método `println()`, a diferencia del método `print()` escribe un separador de línea propio de la plataforma donde se ejecuta.

El método `println()` sin parámetros únicamente escribe un separador de línea.

protected void setError();

Especifica explícitamente el estado de error del *stream*, de forma que el método `checkError()` devolverá `true`.

public void write(char b[]);

Escribe el *buffer* de caracteres indicado por el parámetro.

public void write(char b[], int desplaza, int longitud);

Escribe `longitud` bytes desde el vector de caracteres especificado, empezando en la posición `desplaza`.

public void write(int c);

Escribe el carácter especificado por el parámetro (en Unicode).

Clases sobre streams y ficheros.

public void write(String s);

Escribe la cadena de caracteres que se pasa como parámetro.

public void write(String s, int desplaza, int longitud);

Escribe **longitud** bytes desde el **String** especificado, empezando en la posición **desplaza**.

E.7 La clase File.

E.7.1.1 Atributos.

public static final String pathSeparator;⁵³

String de separación entre paths (dependiente del sistema).

“;” en DOS “:” en UNIX.

public static final char pathSeparatorChar;

Carácter de separación entre paths (dependiente del sistema).

‘;’ en DOS ‘:’ en UNIX.

public static final String separator;

String de separación de directorios dentro de un path (dependiente del sistema).

“\” en DOS “/” en UNIX.

public static final char separatorChar;

Carácter de separación de directorios dentro de un path (dependiente del sistema).

‘\’ en DOS ‘/’ en UNIX.

⁵³ El valor de separador de paths están en la propiedad del sistema **path.separator** y el valor de separador de directorios en un path está en la propiedad del sistema **file.separator** . Estos valores pueden obtenerse con [System.getProperty\(\)](#).

Clases sobre streams y ficheros.

E.7.1.2 Constructores.

public File(String pathYNombre);

Crea una representación del fichero especificado por el nombre y el path indicado en el parámetro.

Throws `NullPointerException` Si el parámetro es `null`.

public File(String path, String nombre);

Crea una representación del fichero especificado por el parámetro `nombre` y el path indicado en el parámetro `path`.

public File(File path, String nombre);

Crea una representación del fichero especificado por el parámetro `nombre` y (aunque parezca raro) en el directorio al que pertenece otro fichero, el especificado por el parámetro `path`.

E.7.1.3 Métodos.

public boolean canRead();

Devuelve `true` si se puede leer del fichero y `false` en caso contrario.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer el fichero.

public boolean canWrite();

Devuelve `true` si se puede escribir en el fichero y `false` en caso contrario.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite escribir en el fichero.

public boolean delete();

Borra el fichero o directorio (si está vacío).

Devuelve:

Clases sobre streams y ficheros.

true si efectivamente ha sido borrado y false en caso contrario.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite borrar el fichero.

public boolean exists();

Devuelve true si existe el fichero y false en caso contrario.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

public String getAbsolutePath();

Devuelve el path absoluto del fichero en forma de string.

public String getCanonicalPath() throws IOException;

Devuelve el path canónico (Sin referencias relativas y dependiente del sistema).

Throws `IOException` si ocurre un error de E/S.

public String getName();

Devuelve el nombre del fichero (sin el path) en forma de string.

public String getParent();

Devuelve el path del directorio padre (anterior) del fichero.

public String getPath();

Devuelve el path del fichero.

public int hashCode();

Devuelve un código *hash* para el fichero.

public native boolean isAbsolute();

Devuelve `true` si el path del fichero es absoluto y `false` en caso contrario.

Clases sobre streams y ficheros.

public boolean isDirectory();

Devuelve true si el objeto representado por la clase `File` es un directorio y false en otro caso.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

public boolean isFile();

Devuelve true si el objeto representado por la clase `File` es un fichero (no directorio) y false en otro caso.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

public long lastModified();

Devuelve el momento (`long`) en que fue modificado por última vez (dependiente del sistema).

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

public long length();

Devuelve la longitud en bytes del fichero (0 si no existe).

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

public String[] list();

Devuelve un vector de `Strings` con los nombres de los ficheros que hay en el directorio del fichero instanciado de la clase `File`.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

Clases sobre streams y ficheros.

public String[] list(FilenameFilter filtro);

Igual que el método anterior, pero en este caso sólo se devuelve el nombre de los ficheros que satisfacen una condición de filtro especificada como parámetro.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite leer del fichero.

public boolean mkdir();

Crea el directorio especificado por el path del objeto de la clase `File`.

Devuelve:

`true` si realmente se ha creado y `false` en caso contrario.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite la escritura del fichero.

public boolean mkdirs();

Igual que el método anterior, pero en este caso se crean los directorios intermedios en caso de que no existan.

Devuelve:

`true` si realmente se ha creado y `false` en caso contrario.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite la escritura del fichero.

public boolean renameTo(File destino);

Cambia el nombre del fichero por el especificado por el objeto que se pasa como parámetro.

Devuelve:

`true` si realmente se ha cambiado y `false` en caso contrario.

Clases sobre streams y ficheros.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite la escritura del fichero.

E.8 La clase `RandomAccessFile`.

E.8.1 Constructores.

public `RandomAccessFile`(File fichero, String modo) throws `IOException`;

Crea un objeto para el acceso a un fichero mediante acceso directo, a partir de un objeto de la clase `File` y un `String` que representa el modo de acceso al mismo, pudiendo ser:

“r” para sólo lectura

“rw” para lectura y escritura.

Throws `SecurityException` si existe un sistema de seguridad activado mediante la clase `SecurityManager` que no permite la lectura y/o escritura del fichero.

`IllegalArgumentException` si el parámetro `modo` no es “r” o “rw”.

`IOException` si ocurre un error de E/S.

public `RandomAccessFile`(String fich, String modo) throws `IOException`;

Igual que el método anterior, pero en este caso se construye a partir de un `String` (con el nombre del fichero y, opcionalmente, path) en lugar de un objeto `File`.

E.8.2 Métodos.

public native void `close`() throws `IOException`;

Cierra el *stream* y libera los recursos asociados.

Throws `IOException` si ocurre un error de E/S.

Clases sobre streams y ficheros.

public final FileDescriptor getFD() throws IOException;

Devuelve el Descriptor de Fichero (FD) del fichero asociado.

Throws `IOException` si ocurre un error de E/S.

public native long getFilePointer() throws IOException;

Devuelve el puntero del fichero. Dicho de otra forma: devuelve en número de bytes existentes entre el principio del fichero y la posición dentro del mismo en el cuál se realizará la próxima operación de lectura o escritura.

Throws `IOException` si ocurre un error de E/S.

public native long length() throws IOException;

Devuelve la longitud (tamaño) en bytes del fichero.

Throws `IOException` si ocurre un error de E/S.

public native int read() throws IOException;

Lee y devuelve un byte del fichero (-1 si ha llegado al final del mismo). En caso de que no se encuentre información disponible, se esperará, bloqueando el thread actual.

Throws `IOException` si ocurre un error de E/S.

public int read(byte b[]) throws IOException;

Igual que el método anterior pero en este caso, se almacenan los caracteres leídos en un vector de bytes que se pasa como parámetro.

Devuelve:

el número de caracteres leído, ó -1 si se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

public int read(byte b[], int desplaza, int longitud) throws IOException;

Lee hasta `longitud` bytes de datos (o hasta llegar a la marca de final de fichero `^Z`) y los almacena en un vector de bytes. Este método se bloquea hasta que haya datos disponibles.

Clases sobre streams y ficheros.

Parámetros:

b - el buffer en el cuál se almacenan los datos leídos

desplaza - el desplazamiento inicial de los datos

longitud - el número máximo de bytes a leer

Devuelve:

el número total de bytes leídos y almacenados en el buffer, ó -1 si no hay más datos porque se ha alcanzado el final del *stream*.

Throws `IOException` si ocurre un error de E/S.

public final boolean readBoolean() throws IOException;

public final byte readByte() throws IOException;

public final char readChar() throws IOException;

public final double readDouble() throws IOException;

public final float readFloat() throws IOException;

public final int readInt() throws IOException;

public final String readLine() throws IOException;

public final long readLong() throws IOException;

public final short readShort() throws IOException;

public final int readUnsignedByte() throws IOException;

public final int readUnsignedShort() throws IOException;

Todos estos métodos leen del fichero un tipo distinto de dato. Todos ellos pueden generar excepciones de las siguientes clases:

Throws `IOException` si ocurre un error de E/S.

`EOFException` si se está intentando leer más allá de la longitud del fichero.

public final void readFully(byte b[]) throws IOException;

Lee completamente hasta `b.length` bytes del fichero, almacenándolos en el vector `b`, y queda bloqueado hasta que lo consigue, llega al final del fichero o se produce una excepción.

Clases sobre streams y ficheros.

Throws `IOException` si ocurre un error de E/S.

`EOFException` si se está intentando leer más allá de la longitud del fichero.

public final void readFully (byte b[], int desplaza, int longitud) throws IOException;

Igual que el método anterior pero en este caso, se lee a partir de la posición `desplaza` y `longitud` bytes.

Throws `IOException` si ocurre un error de E/S.

`EOFException` si se está intentando leer más allá de la longitud del fichero.

public final String readUTF() throws IOException;

Devuelve un `String` codificado en el formato UTF-8. Los dos primeros bytes del `String` indican la longitud del resto del `String`, cuyos bytes son codificados en el formato UTF-8 (↪ Véase F. El formato de Strings UTF-8. en la página 335).

Throws `IOException` si ocurre un error de E/S.

`EOFException` si se está intentando leer más allá de la longitud del fichero.

`UTFDataFormatException` si la codificación UTF-8 no es válida.

public native void seek(long posición) throws IOException;

Posiciona el puntero de fichero en una dirección absoluta indicada por el parámetro.

Throws `IOException` si ocurre un error de E/S.

public int skipBytes(int n) throws IOException

Salta `n` bytes del fichero. Este método se bloquea hasta que se han saltado los `n` bytes, se llegue al final del fichero o se lance una excepción.

Clases sobre streams y ficheros.

Devuelve:

el número de bytes realmente saltado.

Throws `IOException` si ocurre un error de E/S.

public native void write(int b) throws IOException;

Escribe en el fichero el byte menos significativo del entero que se pasa como parámetro.

Throws `IOException` si ocurre un error de E/S.

public void write(byte b[]) throws IOException;

Escribe en el fichero el vector de bytes que se pasa como parámetro.

Throws `IOException` si ocurre un error de E/S.

public void write(byte b[], int desplaza, int longitud) throws IOException;

Escribe `longitud` bytes desde el vector especificado, empezando en la posición `desplaza`, hacia este fichero.

Parámetros:

`b` - los datos

`desplaza` - la posición de comienzo en el vector de datos

`longitud` - el número de bytes a escribir

Throws `IOException` si ocurre un error de E/S.

public final void WriteBoolean(boolean x) throws IOException;

public final void WriteByte(int x) throws IOException;

public final void WriteBytes(String x) throws IOException;

public final void WriteChar(char x) throws IOException;

public final void writeChars(String s) throws IOException

public final void WriteDouble(double x) throws IOException

public final void WriteFloat(float x) throws IOException

public final void WriteInt(int x) throws IOException

Clases sobre streams y ficheros.

public final void WriteLong(long x) throws IOException

public final void WriteShort(int x) throws IOException

Todos estos métodos escriben en el fichero un tipo distinto de dato. Todos ellos pueden generar el mismo tipo de excepción:

Throws `IOException` si ocurre un error de E/S.

public final void writeUTF(String str) throws IOException;

Escribe en el fichero un `String` codificado de la siguiente forma: en primer lugar se escriben dos bytes como si se tratara del método `writeShort()`; a continuación se codifican los caracteres del `String` uno a uno y en secuencia en el formato UTF-8 (véase F. El formato de Strings UTF-8. en la página 335).

Throws `IOException` si ocurre un error de E/S.

F. EL FORMATO DE STRINGS UTF-8.

El formato utilizado por la máquina virtual Java (JVM *Java Virtual Machine*) internamente para representar cadenas de caracteres (*Strings*) es una pequeña variación de la codificación UTF-8. Ésta permite la codificación de los caracteres coincidentes con el código ASCII mediante un solo byte, y a la vez, permitir la codificación de caracteres utilizando hasta 16 bits.

F.1 Caracteres de un solo byte.

0	bits 0-6
---	----------

Este tipo de carácter del String se detecta porque tiene el bit de mayor orden del byte a cero. Los 7 bits menos significativos codifican el carácter representado.

Los caracteres representados por un solo byte son los comprendidos en el rango `'\u0001' ... '\u007f'`, que se corresponden con los pertenecientes al código ASCII de 7 bits:

```
"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`ab
cdefghijklmnopqrstuvwxyz{|}~•
```

F.2 Caracteres de dos bytes.

J	1	1	0	bits 6-10	K	1	0	bits 0-5
---	---	---	---	-----------	---	---	---	----------

Este tipo de carácter puede detectarse viendo que el primer bit del byte menos significativo está a uno “1” y el segundo bit a cero “0”, lo cual significará que el siguiente byte contiene parte del código del carácter. Se detecta que consta de sólo dos bytes y no 3 porque en el segundo byte, el primer cero se encuentra en la tercera posición empezando por la izquierda.

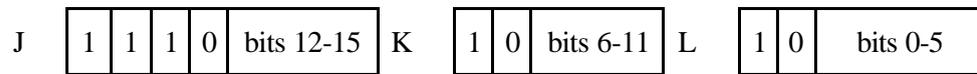
Los caracteres representados por dos bytes son los comprendidos en el rango `'\u0080' ... '\u07ff'`, y el carácter nulo (`'\u0000'`).

El formato de Strings UTF-8.

Puede obtenerse el código del carácter que representan mediante la operación:

$$\text{códigoCarácter} = ((J \& 0x1f) \ll 6) + (K \& 0x3f)$$

F.3 Caracteres de tres bytes.



Se detecta un carácter de tres bytes, una vez detectado que tiene más de un byte, comprobando la posición del primer cero por la izquierda en el segundo byte (está en la segunda).

Los caracteres representados por dos bytes son los comprendidos en el rango `'\u0800' ... '\uffff'`.

Puede obtenerse el código del carácter que representan mediante la operación:

$$\text{códigoCarácter} = ((J \& 0xf) \ll 12) + ((K \& 0x3f) \ll 6) + (L \& 0x3f)$$

G. CLASES EN JAVA.NET.

El paquete `java.net` contiene las clases que soportan la comunicación en Java mediante la implementación de la interface de sockets de UNIX. Realmente se trata de una versión adaptada y simplificada. Seguidamente se pasará revista a las clases implicadas, sus constructores y métodos.

Se puede encontrar una documentación más actualizada (en inglés) en la documentación del JDK.

G.1 *Class Socket*

```
public final class java.net.Socket extends java.lang.Object {
    // Constructores
    public Socket(InetAddress address, int port);
    public Socket(InetAddress address, int port, boolean stream);
    public Socket(String host, int port);
    public Socket(String host, int port, boolean stream);
    // Métodos
    public void close();
    public InetAddress getInetAddress();
    public InputStream getInputStream();
    public int getLocalPort();
    public OutputStream getOutputStream();
    public int getPort();
    public static void setSocketImplFactory(SocketImplFactory fac);
    public String toString();
}
```

Clases en java.net.

Esta clase implementa *sockets* clientes (también llamados simplemente “*sockets*”). Un *socket* es un extremo de una comunicación entre dos máquinas.

El trabajo real del *socket* es desarrollado por una instancia de la clase `SocketImpl`. Una aplicación, cambiando la compañía del *socket* que crea la implementación del *socket*, puede configurarse para crear *sockets* apropiados al cortafuegos local.

G.1.1 CONSTRUCTORES:

public Socket(InetAddress address, int port) throws IOException;

Crea un stream socket y lo conecta al número de *Port* y la dirección IP especificada.

Si la aplicación ha especificado una compañía de *socket*, el método `createSocketImpl()` de esa compañía es llamado para crear la implementación del *socket*. De lo contrario se crea un *socket* simple.

Parámetros:

`address` - la dirección IP

`port` - el número de puerto

Throws:

`IOException`

Si ocurre un error de E/S durante la creación del *socket*.

public Socket(InetAddress address, int port, boolean stream) throws IOException;

Crea un stream socket y lo conecta al número de *Port* y la dirección IP especificada.

Si el parámetro `stream` es `true`, se crea un stream socket. Si el parámetro `stream` es `false`, se crea un datagram socket.

Clases en java.net.

Si la aplicación ha especificado una compañía de socket, el método `createSocketImpl()` de esa compañía es llamado para crear la implementación del socket. De lo contrario se crea un socket simple.

Parámetros:

`address` - la dirección IP

`port` - el número de puerto

`stream` - si es true, crea un stream socket; si es false, crea un datagram socket

Throws

`IOException`

Si ocurre un error de E/S durante la creación del socket.

public Socket(String host, int port) throws UnknownHostException, IOException;

Crea un stream socket y lo conecta al número de Port y el nombre de Host especificado.

Si la aplicación ha especificado una compañía de socket, el método `createSocketImpl()` de esa compañía es llamado para crear la implementación del socket. De lo contrario se crea un socket simple.

Parámetros:

`host` - el nombre del host

`port` - el número de puerto

Throws

`IOException`

Si ocurre un error de E/S durante la creación del socket.

Clases en java.net.

public Socket(String host, int port, boolean stream) throws IOException;

Crea un stream socket y lo conecta al número de Port y el nombre de Host especificado.

Si el parámetro stream es true, se crea un stream socket. Si el parámetro stream es false, se crea un datagram socket.

Si la aplicación ha especificado una compañía de socket, el método `createSocketImpl()` de esa compañía es llamado para crear la implementación del socket. De lo contrario se crea un socket simple.

Parámetros:

host - el nombre del host

port - el número de puerto

stream - si es true, crea un stream socket; si es false, crea un datagram socket

Throws

`IOException`

Si ocurre un error de E/S durante la creación del socket.

G.1.2 MÉTODOS

public void close() throws IOException;

Cierra este socket.

Throws

`IOException`

Si ocurre un error de E/S durante la creación del socket.

public InetAddress getInetAddress();

Devuelve:

Clases en java.net.

La dirección IP remota a la cual está conectado el socket.

public InputStream getInputStream() throws IOException;

Devuelve:

un input stream para leer bytes desde este socket.

Throws

`IOException`

Si ocurre un error de E/S durante la creación del input stream.

public int getLocalPort();

Devuelve:

el Port local al cual está conectado este socket.

public OutputStream getOutputStream() throws IOException;

Devuelve:

un output stream para escribir bytes hacia este socket.

Throws

`IOException`

Si ocurre un error de E/S durante la creación del output stream.

public int getPort();

Devuelve:

el Port remoto al cual está conectado este socket.

**public static void setSocketImplFactory(SocketImplFactory fac) throws
IOException;**

Establece la compañía de implementación del socket para la aplicación.
La compañía puede ser especificada una sola vez.

Clases en java.net.

Cuando una aplicación crea un nuevo cliente socket, se llama al método `createSocketImpl()` de la compañía que implementa el socket para crear la implementación de socket.

Parámetros:

`fac` - la compañía deseada

Throws

`SocketException`

Si la compañía ya está definida.

Throws

`IOException`

Si ocurre un error de E/S durante el establecimiento de la compañía.

public String toString();

Devuelve:

La representación en forma de `String` de este socket.

G.2 Class ServerSocket

```
public final class java.net.ServerSocket extends java.lang.Object {  
  // Constructores  
  public ServerSocket(int port);  
  public ServerSocket(int port, int count);  
  // Métodos  
  public Socket accept();  
  public void close();  
  public InetAddress getInetAddress();
```

Clases en java.net.

```
public int getLocalPort();  
public static void setSocketFactory(SocketImplFactory fac);  
public String toString();  
}
```

Esta clase implementa sockets servidores. Un socket servidor espera peticiones de conexión a través de la red. Realiza alguna operación basada en la petición, y entonces, posiblemente devuelve un resultado al peticionario.

El trabajo real del socket es desarrollado por una instancia de la clase `SocketImpl`. Una aplicación, cambiando la compañía del socket que crea la implementación del socket, puede configurarse para crear sockets apropiados al cortafuegos local.

G.2.1 CONSTRUCTORES:

```
public ServerSocket(int port) throws IOException;
```

Crea un socket servidor en un Port determinado. Un puerto 0 crea un socket en cualquier Port libre.

La longitud máxima de indicaciones de conexiones entrantes (una petición de conexión) se establece en 50. Si una indicación de conexión llega cuando la cola está llena, la conexión es rechazada.

Si la aplicación ha especificado una compañía de socket, el método `createSocketImpl()` de esa compañía es llamado para crear la implementación del socket. De lo contrario se crea un socket simple.

Parámetros:

`port` - el número de puerto, ó 0 para usar cualquier puerto libre.

Throws

`IOException`

Si ocurre un error de E/S al abrir el socket.

public ServerSocket(int port, int count) throws IOException;

Crea un socket servidor y lo enlaza (*bind*) al puerto local especificado. Un número de puerto 0 crea un socket en cualquier puerto libre.

La longitud máxima de indicaciones de conexiones entrantes (una petición de conexión) se establece en 50. Si una indicación de conexión llega cuando la cola está llena, la conexión es rechazada.

Si la aplicación ha especificado una compañía de socket, el método `createSocketImpl()` de esa compañía es llamado para crear la implementación del socket. De lo contrario se crea un socket simple.

Parámetros:

port - el número de puerto, ó 0 para usar cualquier puerto libre.

count - la máxima longitud de la cola

Throws

`IOException`

Si ocurre un error de E/S al abrir el socket.

G.2.2 MÉTODOS

public Socket accept() throws IOException;

Escucha la llegada de una conexión a este socket y la acepta. El método se bloquea hasta que se realice una conexión.

Throws

`IOException`

Si ocurre un error de E/S esperando la conexión.

public void close() throws IOException;

Cierra este socket.

Clases en java.net.

Throws

`IOException`

Si ocurre un error de E/S al cerrar el socket.

public InetAddress getInetAddress();

Devuelve:

La dirección IP remota a la cual está conectado el socket, o `null` si el socket no está conectado todavía.

public int getLocalPort();

Devuelve:

el puerto en el cual está escuchando este socket.

**public static void setSocketFactory(SocketImplFactory fac) throws
IOException;**

Establece la compañía de implementación del socket servidor para la aplicación. La compañía puede ser especificada una sola vez.

Cuando una aplicación crea un nuevo cliente socket, se llama al método `createSocketImpl()` de la compañía que implementa el socket para crear la implementación de socket.

Parámetros:

`fac` - la compañía deseada

Throws

`SocketException`

Si la compañía ya está definida.

Throws

`IOException`

Clases en java.net.

Si ocurre un error de E/S durante el establecimiento de la compañía.

```
public String toString();
```

Devuelve:

un `String` representando este socket.

G.3 Class DatagramSocket

```
public class java.net.DatagramSocket extends java.lang.Object {  
    // Constructores  
    public DatagramSocket();  
    public DatagramSocket(int port);  
    // Métodos  
    public void close();  
    protected void finalize();  
    public int getLocalPort();  
    public void receive(DatagramPacket p);  
    public void send(DatagramPacket p);  
}
```

Esta clase implementa un socket para enviar y recibir paquetes datagramas.

Un datagram socket es el extremo de envío o recepción de un servicio de entrega de paquetes sin conexión. Cada paquete enviado o recibido en un datagram socket es direccionado y encaminado individualmente. Múltiples paquetes enviados de una máquina a otra pueden ser encaminados de forma diferente y llegar en cualquier orden.

G.3.1 CONSTRUCTORES

public DatagramSocket() throws SocketException;

Construye un datagram socket y lo enlaza a cualquier puerto disponible en la máquina local.

Throws

[SocketException](#)

Si el socket no pudo ser abierto, o el socket no pudo ser enlazado al port local especificado.

public DatagramSocket(int port) throws SocketException;

Construye un datagram socket y lo enlaza al puerto especificado en la máquina local.

Parámetros:

port - número de puerto a usar

Throws

[SocketException](#)

Si el socket no pudo ser abierto, o el socket no pudo ser enlazado al port local especificado.

G.3.2 MÉTODOS

public void close();

Cierra este datagram socket.

protected void finalize();

Asegura que este socket se cierra si no hay más referencias a este socket.

public int getLocalPort();

Devuelve:

el número de puerto en el host local al cual está conectado este socket.

public void receive(DatagramPacket p) throws IOException;

Recibe un datagram packet de este socket. Cuando este método termina, el buffer `DatagramPacket` se llena con los datos recibidos. El datagram packet también contiene la dirección IP del emisor, y el número de puerto de la máquina emisora.

Este método se bloquea hasta que se recibe un datagrama. El campo longitud del datagram packet contiene la longitud del mensaje recibido. Si el mensaje es más largo que la longitud del *buffer*, el mensaje se trunca.

Parámetros:

`p` - el `DatagramPacket` dentro del cual están los datos que llegan

Throws

`IOException`

Si ocurre un error de E/S.

public void send(DatagramPacket p) throws IOException;

Envía un datagram packet a través de este socket. El `DatagramPacket` incluye información indicando los datos a enviar, su longitud, la dirección IP del host remoto, y el puerto en el host remoto.

Parámetros:

`p` - el `DatagramPacket` dentro del cual están los datos que llegan

Throws

`IOException`

Clases en java.net.

Si ocurre un error de E/S.

G.4 Class *InetAddress*

```
public final class java.net.InetAddress extends java.lang.Object {  
    // Métodos  
    public boolean equals(Object obj);  
    public byte[] getAddress();  
    public static InetAddress[] getAllByName(String host);  
    public static InetAddress getByName(String host);  
    public String getHostName();  
    public static InetAddress getLocalHost();  
    public int hashCode();  
    public String toString();  
}
```

Esta clase implementa la dirección en Internet Protocol (IP).

Las aplicaciones deberían usar los métodos `getLocalHost()`, `getByName()`, o `getAllByName()` para crear una nueva instancia `InetAddress`.

G.4.1 MÉTODOS

```
public boolean equals(Object obj);
```

El resultado es true si y sólo si el parámetro no es `null` e implementa la misma dirección IP que este objeto.

Dos instancias de `InetAddress` implementan la misma dirección IP si la longitud del vector de bytes devuelto por `getAddress()` es la misma para ambos, y cada uno de los componentes del vector de componentes es el mismo que el vector de bytes.

Clases en java.net.

Parámetros:

`obj` - el objeto con el cual comparar

Devuelve:

true si los objetos son los mismos y false en caso contrario.

public byte[] getAddress();

Determina la dirección raw IP de este objeto `InetAddress`. El resultado está en orden de red: el byte de mayor orden de la dirección está en `getAddress()[0]`.

Devuelve:

la dirección raw IP de este objeto.

**public static InetAddress[] getAllByName(String host) throws
UnknownHostException;**

Parámetros:

`host` - el nombre del host

Devuelve:

un vector con todas las direcciones IP addresses del host especificado.

Throws

`UnknownHostException`

Si no se pudo encontrar una dirección IP para el host.

**public static InetAddress getByName(String host) throws
UnknownHostException;**

Determina la dirección IP de un host, dado el nombre de host. El nombre de host puede ser un nombre, como "java.sun.com" o un `String` representando su dirección IP, como "206.26.48.100".

Clases en java.net.

Parámetros:

`host` - el host especificado, o `null` para el host local

Devuelve:

una dirección IP para el nombre de host.

Throws

`UnknownHostException`

Si no se pudo encontrar una dirección IP para el host.

public String getHostName();

Devuelve:

el nombre de host para esta dirección IP.

public static InetAddress getLocalHost() throws UnknownHostException;

Devuelve:

la dirección IP para el host local.

Throws

`UnknownHostException`

Si no se pudo encontrar una dirección IP para el host.

public int hashCode();

Devuelve:

un código *hash* para esta dirección IP.

public String toString();

Devuelve:

un `String` representando la dirección IP.

G.5 Class *DatagramPacket*

```
public final class java.net.DatagramPacket extends java.lang.Object {
    // Constructores
    public DatagramPacket(byte ibuf[], int ilength);
    public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int
        iport);
    // Métodos
    public InetAddress getAddress();
    public byte[] getData();
    public int getLength();
    public int getPort();
}
```

Esta clase implementa un datagram packet.

Los datagram packets se usan para implementar un servicio de reparto de paquetes sin conexión. Cada mensaje es encaminado de una máquina a otra basándose únicamente en la información contenida en ese paquete. Múltiples paquetes enviados de una máquina a otra pueden ser encaminados de forma distinta y pueden llegar en cualquier orden.

G.5.1 CONSTRUCTORES

```
public DatagramPacket(byte ibuf[], int ilength);
```

Construye un `DatagramPacket` para recibir paquetes de longitud `ilength`.

El parámetro `ilength` debe ser menor o igual que `ibuf.length`

Parámetros:

`ibuf` - buffer para almacenar el datagrama entrante

Clases en java.net.

`ilength` - número de bytes a leer

```
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int  
iport);
```

Construye un `DatagramPacket` para enviar paquetes de longitud `ilength` al número de puerto del host especificado.

El parámetro `ilength` debe ser menor o igual que `ibuf.length`

Parámetros:

`ibuf` - los datos del paquete

`ilength` - la longitud del paquete

`iaddr` - la dirección de destino

`iport` - el número de puerto de destino

G.5.2 MÉTODOS

```
public InetAddress getAddress();
```

Devuelve:

La dirección IP de la máquina a la cual este datagrama se va a enviar o del cual el datagrama fue recibido.

```
public byte[] getData();
```

Devuelve:

los datos recibidos o los datos a enviar.

```
public int getLength();
```

Devuelve:

la longitud de los datos a enviar o la longitud de los datos recibidos.

Clases en java.net.

public int getPort();

Devuelve:

el número de puerto en el host remoto al cual este datagrama va a ser enviado o del cual el datagrama fue recibido.