

CONTENIDO

1. Introducción
2. Generalidades:
3. Historia:
 - a. Simple
 - b. Orientado a objetos
 - c. Distribuido
 - d. Robusto
 - e. Arquitectura neutral
 - f. Seguro
 - k. Recolector de basura
 - g. Portable o Independiente de plataforma
 - h. Interpretado
 - i. Multithreaded
 - j. Dinamico
4. Ventajas de Java
5. Limitantes:
6. Filosofia
7. Versiones
8. Java y DB
9. Java e Internet
10. Java y otros otros lenguajes de programación

Programación en Java

1. Tipos
 - a. Tipos primitivos
2. Variables
3. Literales
4. Operaciones sobre Tipos primitivos
5. Conversión entre tipos numéricos
6. Métodos
7. El termino void
8. Uso de métodos
9. Clases
10. Objetos, miembros y referencias
11. Conceptos básicos
12. Constructores
 - a. Constructor no-args.
 - b. Sobrecarga de constructores.
13. Datos estáticos
 - a. Métodos estáticos
 - b. El método main
 - c. Inicializadores estáticos
14. Inicialización de variables
 - a. Ambito de las variables
15. Recogida de basura
16. Sobrecarga de métodos
17. La referencia this
18. La referencia null
19. Ocultamiento de variables
20. Operadores
21. Funciones de Entrada y Salida

- a. Entrada por teclado en Java
 - b. Salida por pantalla en Java
 - c. Entrada por archivo
 - d. Salida por archivo en Java
22. La clase System
23. Stdout
24. Stderr
25. Clases comunes de Entrada/Salida
26. Streams de Entrada
27. Streams de Salida
28. Ejecución condicional
29. Iteraciones con for
30. Evaluación múltiple
31. Devolución de control
32. Expresiones
33. Arrays
- a. Arrays multidimensionales
34. Strings
35. Packages
36. Compilación y ejecución de programas
37. Modificadores
38. Definición de métodos. El uso de super.
39. La clase Object
40. Herencia simple
41. Gestión de Excepciones
42. Clases envoltorio (Wrapper)
43. Clases abstractas
44. Interfaces
45. Clases embebidas (Inner classes)
46. Anexos
- a. Ejemplo de como conectar java con el JDB
 - b. Applets
 - c. Cómo crear un applet
 - d. Requerimientos mínimos de instalación:
 - e. Manual de usuario. Instalación y configuración de Eclipse
47. Bibliografía

JAVA

11. Introducción

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 1990. Las aplicaciones Java están típicamente compiladas en un *bytecode*¹, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el *bytecode* es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del *bytecode* por un procesador Java también es posible.

El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel como punteros. JavaScript, un lenguaje interpretado², comparte un nombre similar y una sintaxis similar, pero no está directamente relacionado con Java.

Sun Microsystems proporciona una implementación GNU³ General Public License de un compilador Java y una máquina virtual Java, conforme a las especificaciones del Java Community Process, aunque la biblioteca de clases que se requiere para ejecutar los programas Java no es software libre⁴.

12. Generalidades:

Java Paradigma: Orientado a objetos

Apareció en: 1990s

Diseñado por: Sun Microsystems

Tipo de dato: Fuerte, Estático

Implementaciones: Numerosas Influido por: Objective-C⁵, C++, Smalltalk⁶, Eiffel⁷ Ha influido: C#⁸, J#⁹, JavaScript¹⁰

13. Historia:

Java comenzó como un proyecto llamado "Green" y su objetivo inicial era crear un lenguaje que fuera capaz de ejecutarse en electrodomésticos que tuvieran microprocesadores pero se dieron cuenta que ese tipo de tecnología estaba aun muy lejos de poder existir.

El proyecto dio como resultado un lenguaje muy parecido a C/C++ al cual le llamaron "Oak" (en referencia al roble que se encontraba en el exterior de las oficinas de Sun Microsystems) por James Gosling en junio de 1991 para usarse en un proyecto de receptor digital externo, pero descubrieron que ya existía un lenguaje con este nombre, luego alguien sugirió el nombre de Java (se cree que es por un tipo de café y otros piensan que son siglas) y fue ese nombre el que quedó.

La primera implementación pública fue Java 1.0 en 1995. Prometía "Escribir una vez, ejecutar en cualquier parte" ("*Write once, run anywhere*"), proporcionando ningún coste extra en el tiempo de ejecución en las plataformas populares. Era bastante seguro y su seguridad era configurable, permitiendo restringir el acceso a archivos o a una red.

Los principales navegadores web pronto incorporaron la capacidad de ejecutar "applets"¹¹ Java seguros dentro de páginas web. Java adquirió popularidad rápidamente. Con la llegada de "Java 2", las nuevas versiones tuvieron múltiples configuraciones pensadas para diferentes tipos de plataformas. Por ejemplo, J2EE¹² era para aplicaciones de empresa y la versión reducida J2ME¹³ era para aplicaciones para móviles. J2SE era la designación para la Edición Estándar. En 2006, las nuevas versiones "J2" fueron renombradas a Java EE, Java ME y Java SE, respectivamente.

En 1997, Sun se dirigió al cuerpo de estándares ISO/IEC JTC1 y más tarde a Ecma International para formalizar Java, pero pronto se retiró del proceso. Java permanece como un estándar de facto propietario que está controlado a través del Java Community Process. Sun hace disponibles la mayoría de sus implementaciones Java sin cargo alguno, generando los ingresos con productos especializados como el Java Enterprise System. Sun distingue entre su Software Development Kit (SDK) y su Java Runtime Environment (JRE) que es un subconjunto del SDK, siendo la principal distinción que en el JRE no está presente el compilador.

El 13 de noviembre de 2006, Sun liberó partes de Java como software libre/de código abierto, bajo la GNU General Public License (GPL). La publicación del código fuente completo bajo la GPL se espera que ocurra en la primera mitad de 2007.

14. Características:

Las características principales que nos ofrece Java respecto a cualquier otro lenguaje de programación, son:

a. Simple

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ es un lenguaje que adolece de falta de seguridad, pero C y C++ son lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el garbage collector (reciclador de memoria dinámica).

No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es un thread de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan:

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

Aunque, en realidad, lo que hace es eliminar las palabras reservadas (struct, typedef), ya que las clases son algo parecido.

Además, el intérprete completo de Java que hay en este momento es muy pequeño, solamente ocupa 215 Kb de RAM.

b. Orientado a objetos

Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. Las plantillas de objetos son llamadas, como en C++, clases y sus copias, instancias. Estas instancias, como en C++, necesitan ser construidas y destruidas en espacios de memoria.

Java incorpora funcionalidades inexistentes en C++ como por ejemplo, la resolución dinámica de métodos. Esta característica deriva del lenguaje Objective C, propietario del sistema operativo Next. En C++ se suele trabajar con librerías dinámicas (DLLs) que obligan a recompilar la aplicación cuando se retocan las funciones que se encuentran en su interior. Este inconveniente es resuelto por Java mediante una interfaz específica llamada RTTI (RunTime Type Identification) que define la interacción entre objetos excluyendo variables de instancias o implementación de métodos. Las clases en Java tienen una representación en el runtime que permite a los programadores interrogar por el tipo de clase y enlazar dinámicamente la clase con el resultado de la búsqueda.

c. Distribuido

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales. Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando.

d. Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

También implementa los arrays auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobreescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo de desarrollo de aplicaciones en Java.

Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los byte-codes, que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

e. Arquitectura neutral

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando en el porting a otras plataformas.

f. Seguro

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el casting implícito que hacen los compiladores de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El lenguaje C, por ejemplo, tiene lagunas de seguridad importantes, como son los errores de alineación. Los programadores de C utilizan punteros en conjunción con operaciones aritméticas. Esto le permite al programador que un puntero referencie a un lugar conocido de la memoria y pueda sumar (o restar) algún valor, para referirse a otro lugar de la memoria. Si otros programadores conocen nuestras estructuras de datos pueden extraer información confidencial de nuestro sistema.

Otra laguna de seguridad u otro tipo de ataque, es el Caballo de Troya. Se presenta un programa como una utilidad, resultando tener una funcionalidad destructiva. Por ejemplo, en UNIX se visualiza el contenido de un directorio con el comando ls. Si un programador deja un comando destructivo bajo esta referencia, se puede correr el riesgo de ejecutar código malicioso, aunque el comando siga haciendo la funcionalidad que se le supone, después de lanzar su carga destructiva.

Por ejemplo, después de que el caballo de Troya haya enviado por correo el /etc/shadow a su creador, ejecuta la funcionalidad de ls persentando el contenido del directorio. Se notará un retardo, pero nada inusual.

El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de byte-codes que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal -código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto-.

Si los byte-codes pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

- El código no produce desbordamiento de operandos en la pila
- El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
- No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros en punteros.
- El acceso a los campos de un objeto se sabe que es legal: public, private, protected.
- No hay ningún intento de violar las reglas de acceso y seguridad establecidas

El Cargador de Clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros local, del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya, ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método ultraseguro de autenticación por clave pública.

El Cargador de Clases puede verificar una firma digital antes de realizar una instancia de un objeto. Por tanto, ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso. Es decir, la seguridad se integra en el momento de compilación, con el nivel de detalle y de privilegio que sea necesario.

Dada, pues la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por Sun, no hay peligro. Java imposibilita, también, abrir ningún fichero de la máquina local (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el

applet), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra. Además, los intérpretes que incorporan los navegadores de la Web son aún más restrictivos.

Bajo estas condiciones (y dentro de la filosofía de que el único ordenador seguro es el que está apagado, desenchufado, dentro de una cámara acorazada en un bunker y rodeado por mil soldados de los cuerpos especiales del ejército), se puede considerar que Java es un lenguaje seguro y que los applets están libres de virus.

Respecto a la seguridad del código fuente, no ya del lenguaje, JDK proporciona un desensamblador de byte-code, que permite que cualquier programa pueda ser convertido a código fuente, lo que para el programador significa una vulnerabilidad total a su código. Utilizando javap no se obtiene el código fuente original, pero sí desmonta el programa mostrando el algoritmo que se utiliza, que es lo realmente interesante. La protección de los programadores ante esto es utilizar llamadas a programas nativos, externos (incluso en C o C++) de forma que no sea descompilable todo el código; aunque así se pierda portabilidad. Esta es otra de las cuestiones que Java tiene pendientes.

g. Portable o Independiente de plataforma

Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

h. Interpretado

El intérprete Java (sistema run-time) puede ejecutar directamente el código objeto. Enlazar (linkar) un programa, normalmente, consume menos recursos que compilarlo, por lo que los desarrolladores con Java pasarán más tiempo desarrollando y menos esperando por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, que todavía no hay compiladores específicos de Java para las diversas plataformas, Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional.

Se dice que Java es de 10 a 30 veces más lento que C, y que tampoco existen en Java proyectos de gran envergadura como en otros lenguajes. La verdad es que ya hay comparaciones ventajosas entre Java y el resto de los lenguajes de programación, y una ingente cantidad de folletos electrónicos que supuran fanatismo en favor y en contra de los distintos lenguajes contendientes con Java.

Lo que se suele dejar de lado en todo esto, es que primero habría que decidir hasta que punto Java, un lenguaje en pleno desarrollo y todavía sin definición definitiva, está maduro como lenguaje de programación para ser comparado con otros; como por ejemplo con Smalltalk, que lleva más de 20 años en cancha.

La verdad es que Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es ningún contrasentido, me explico, el código fuente escrito con cualquier editor se compila generando el byte-code. Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquina propia de cada plataforma y no tiene nada que ver con el p-code de Visual Basic. El byte-code corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello hace falta el run-time, que sí es completamente dependiente de la máquina y del sistema operativo, que interpreta dinámicamente el byte-code y añade el 20% de instrucciones que faltaban para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el run-time correspondiente al sistema operativo utilizado.

i. Multithreaded

Al ser multithreaded (multihilo), Java permite muchas actividades simultáneas en un programa. Los threads (a veces llamados, procesos ligeros), son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads contruidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++.

El beneficio de ser multithreaded consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.), aún supera a los entornos de flujo único de programa (single-threaded) tanto en facilidad de desarrollo como en rendimiento.

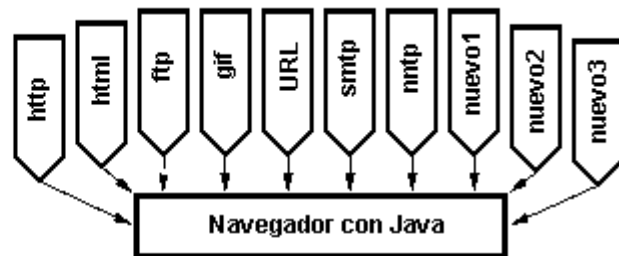
Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una gran imagen que se está trayendo. En Java, las imágenes se pueden ir trayendo en un thread independiente, permitiendo que el usuario pueda acceder a la información en la página sin tener que esperar por el navegador.

j. Dinamico

Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior).



Monolito: cada pieza de código se compacta dentro del código del navegador



Sistema Federado: el navegador es un coordinador de piezas, y cada pieza es responsable de una función. Las piezas se pueden añadir dinámicamente a través de la red

Java también simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, tal como se ha explicado en párrafos anteriores, Java es capaz de traer automáticamente cualquiera de esas piezas que el sistema necesita para funcionar.

Java, para evitar que los módulos de byte-codes o los objetos o nuevas clases, haya que estar trayéndolos de la red cada vez que se necesiten, implementa las opciones de persistencia, para que no se eliminen cuando se limpie la caché de la máquina.

k. Recolector de basura

Un argumento en contra de lenguajes como C++ es que los programadores se encuentran con la carga añadida de tener que administrar la memoria de forma manual. En C++, el desarrollador debe asignar memoria en una zona conocida como *heap* (montículo) para crear cualquier objeto, y posteriormente desalojar el espacio asignado cuando desea borrarlo. Un olvido a la hora de desalojar memoria previamente solicitada, o si no lo hace en el instante oportuno, puede

llevar a una *fuga de memoria*, ya que el sistema operativo piensa que esa zona de memoria está siendo usada por una aplicación cuando en realidad no es así. Así, un programa mal diseñado podría consumir una cantidad desproporcionada de memoria. Además, si una misma región de memoria es desalojada dos veces el programa puede volverse inestable y llevar a un eventual *cuelgue*.

En Java, este problema potencial es evitado en gran medida por el recolector automático de basura (o *automatic garbage collector*). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos.

El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste (que, desde un punto de vista de bajo nivel es una dirección de memoria). Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aún así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y frecuentemente más rápida que en C++.

La recolección de basura de Java es un proceso prácticamente invisible al desarrollador. Es decir, el programador no tiene conciencia de cuándo la recolección de basura tendrá lugar, ya que ésta no tiene necesariamente que guardar relación con las acciones que realiza el código fuente.

15. Ventajas de Java

- No se debe volver a escribir el código si quieres ejecutar el programa en otra máquina. Un solo código funciona para todos los browsers compatibles con Java o donde se tenga una Máquina Virtual de Java (Mac's, PC's, Sun's, etc).
- Java es un lenguaje de programación orientado a objetos, y tiene todos los beneficios que ofrece esta metodología de programación (más adelante doy una pequeña introducción a la filosofía de objetos).
- Un browser compatible con Java deberá ejecutar cualquier programa hecho en Java, esto ahorra a los usuarios tener que estar insertando "plug-ins" y demás programas que a veces nos quitan tiempo y espacio en disco.
- Java es un lenguaje y por lo tanto puede hacer todas las cosas que puede hacer un lenguaje de programación: Cálculos matemáticos, procesadores

de palabras, bases de datos, aplicaciones gráficas, animaciones, sonido, hojas de cálculo, etc.

- Si lo que me interesa son las páginas de Web, ya no tienen que ser estáticas, se le pueden poner toda clase de elementos multimedia y permiten un alto nivel de interactividad, sin tener que gastar en paquetes carísimos de multimedia.

16. Limitantes:

- La velocidad. Los programas hechos en Java no tienden a ser muy rápidos, supuestamente se está trabajando para mejorar esto. Como los programas de Java son interpretados nunca alcanzan la velocidad de un verdadero ejecutable.
- Java es un lenguaje de programación. Esta es otra gran limitante, por más que digan que es orientado a objetos y que es muy fácil de aprender sigue siendo un lenguaje y por lo tanto aprenderlo no es cosa fácil. Especialmente para los no programadores.
- Java es nuevo. En pocas palabras todavía no se conocen bien todas sus capacidades. Pero en general Java posee muchas ventajas y se pueden hacer cosas muy interesantes con esto. Hay que prestar especial atención a lo que está sucediendo en el mundo de la computación, a pesar de que Java es relativamente nuevo, posee mucha fuerza y es tema de moda en cualquier medio computacional.

17. Filosofía

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar la metodología de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.

4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

Para conseguir la ejecución de código remoto y el soporte de red, los programadores de Java a veces recurren a extensiones como CORBA (Common Object Request Broker Architecture), Internet Communications Engine o OSGi respectivamente.

8 Versiones

El proyecto Java ha visto muchas versiones publicadas. Desde 1997, estas son:

- JDK 1.1.4 (*Sparkler*) 12 de septiembre de 1997
 - JDK 1.1.5 (*Pumpkin*) 3 de diciembre de 1997
 - JDK 1.1.6 (*Abigail*) 24 de abril de 1998
 - JDK 1.1.7 (*Brutus*) 28 de septiembre de 1998
 - JDK 1.1.8 (*Chelsea*) 8 de abril de 1999
- J2SE 1.2 (*Playground*) 4 de diciembre de 1998
 - J2SE 1.2.1 (*ninguno*) 30 de marzo de 1999
 - J2SE 1.2.2 (*Cricket*) 8 de julio de 1999
- J2SE 1.3 (*Kestrel*) 8 de mayo de 2000
 - J2SE 1.3.1 (*Ladybird*) 17 de mayo de 2001
- J2SE 1.4.0 (*Merlin*) 13 de febrero de 2002
 - J2SE 1.4.1 (*Hopper*) 16 de septiembre de 2002
 - J2SE 1.4.2 (*Mantis*) 26 de junio de 2003
- J2SE 5.0 (1.5.0) (*Tiger*) 29 de septiembre de 2004
- Java SE 6 (1.6.0) (*Mustang*) 11 de diciembre de 2006
- Java SE 7 (1.7.0) (*Dolphin*) previsto para 2008

9. Java y DB

Muchos programadores quizás tengan mayor interés en realizar programación basada conjuntamente a Bases de Datos, pues Java no se queda atrás, Java no implementa Bases de Datos, ya que solo es un lenguaje de programación, pero implementa funciones que permiten al programador realizar conexiones entre la interfaz de usuario y el Gestor de Base de Datos. Java permite conectarse por medio de puentes JDBC o a través de Driver's a programas gestores de bases de datos, su independencia entre ambos permite al usuario mantener siempre un enfoque, separando el diseño de la Base de Datos y el de la interfaz en dos mundos de pensamientos diferentes el mundo de los datos y el mundo de las interfaces.

Java es orientado a objetos por ende da solidez a la aplicación evitando cortes bruscos del programa y permitiendo continuar de esta manera con la aplicación. Java permite Applets, lo que permite montar cualquier aplicación con Bases de Datos a través de la red de forma segura y sólida.

10. Java e Internet

Entre junio y julio de 1994, tras una sesión maratónica de tres días entre John Gaga, James Gosling, Joy Naughton, Wayne Rosing y Eric Schmidt, el equipo reorientó la plataforma hacia la Web. Sintieron que la llegada del navegador Web Mosaic, propiciaría que Internet se convirtiese en un medio interactivo, como el que pensaban era la televisión por cable. Naughton creó entonces un prototipo de navegador, WebRunner, que más tarde sería conocido como HotJava¹⁴.

Ese año renombraron el lenguaje como Java tras descubrir que "Oak" era ya una marca comercial registrada para adaptadores de tarjetas gráficas. El término Java fue acuñado en una cafetería frecuentada por algunos de los miembros del equipo. Pero no está claro si es un acrónimo o no, aunque algunas fuentes señalan que podría tratarse de las iniciales de sus creadores: **J**ames **G**osling, **A**rthur **V**an Hoff, y **A**ndy **B**echtolsheim. Otros abogan por el siguiente acrónimo, **J**ust **A**nother **V**ague **A**cronym ("sólo otro acrónimo ambiguo más"). La hipótesis que más fuerza tiene es la que Java debe su nombre a un tipo de café disponible en la cafetería cercana. Un pequeño signo que da fuerza a esta teoría es que los 4 primeros bytes (el *número mágico*) de los archivos .class que genera el compilador, son en hexadecimal, 0xCAFEBAE.

En octubre de 1994, se les hizo una demostración de HotJava y la plataforma Java a los ejecutivos de Sun. Java 1.0a pudo descargarse por primera vez en 1994, pero hubo que esperar al 23 de mayo de 1995, durante las conferencias de SunWorld, a que vieran la luz pública Java y HotJava, el navegador Web. El acontecimiento fue anunciado por John Gage, el Director Científico de Sun Microsystems. El acto estuvo acompañado por una pequeña sorpresa adicional, el anuncio por parte de Marc Andreessen, Vicepresidente Ejecutivo de Netscape, que Java sería soportado en sus navegadores. El 9 de enero del año siguiente, 1996, Sun fundó el grupo empresarial JavaSoft para que se encargase del desarrollo tecnológico. Dos semanas más tarde la primera versión de Java fue publicada.

11. Java y otros otros lenguajes de programación

Lenguaje	Paradigma	Tipos de Datos	Implementaciones	Influido por:
Java	Orientado a objetos	Fuerte, estático	Numerosas	Objective-C, C++, Smalltalk, Eiffel
C	Procedural	Débil, estático	Múltiples	B
C++	orientado a objetos, imperativo, programación genérica	Fuerte, estático	GNU Compiler Collection, Microsoft Visual C++, Borland C++ Builder, Dev-C++, C-Free	C, Simula
C#	Orientado a objetos	Fuerte, estático	Visual C#, Mono	Java, C++, Delphi, Eiffel
JavaScript	Basado en prototipos, Programación orientada a objetos	Débil, dinámico	Numerosas	Java
Smalltalk	Orientado a	Dinámico	Múltiples	Simula,

	objetos			Sketchpad, LISP
Lisp	orientado a objetos, funcional, declarativo	Fuerte, dinámico	Múltiples	
Python	Multiparadigma	Fuerte, dinámico	CPython, Jython, IronPython, PyPy	ABC, TCL, Perl, Modula-3, Smalltalk
PHP	multiparadigma	Dinámico	múltiples	AWK, BASIC-PLUS, C, C++, Lisp, Pascal, sed, Unix shell
Pascal	imperativo (estructurado)		múltiples	Algol
Perl	Multiparadigma	Dinámico	Perl mod_perl embperl	AWK, BASIC-PLUS, C, C++, Lisp, Pascal, sed, Unix shell
Objective-C	Orientado a objetos	Fuerte, dinámico	Numerosas	C, Smalltalk

Programación en Java

1. Tipos

Java es un lenguaje con control fuerte de Tipos (*Strongly Typed*). Esto significa que cada variable y cada expresión tiene un Tipo que es conocido en el momento de la compilación. El Tipo limita los valores que una variable puede contener, limita las operaciones soportadas sobre esos valores y determina el significado de la operaciones. El control fuerte de tipos ayuda a detectar errores en tiempo de compilación.

En Java existen dos categorías de Tipos:

- Tipos Primitivos
- Referencias

1.1 Tipos primitivos

Los tipos primitivos son los que permiten manipular valores numéricos (con distintos grados de precisión), caracteres y valores booleanos (verdadero / falso). Los Tipos Primitivos son:

- **boolean** : Puede contener los valores **true** o **false**.
- **byte** : Enteros. Tamaño 8-bits. Valores entre -128 y 127.
- **short** : Enteros. Tamaño 16-bits. Entre -32768 y 32767.
- **int** : Enteros. Tamaño 32-bits. Entre -2147483648 y 2147483647.
- **long** : Enteros. Tamaño 64-bits. Entre -9223372036854775808 y 9223372036854775807.
- **float** : Números en coma flotante. Tamaño 32-bits.
- **double** : Números en coma flotante. Tamaño 64-bits.
- **char** : Caracteres. Tamaño 16-bits. Unicode. Desde '\u0000' a '\uffff' inclusive. Esto es desde 0 a 65535

El tamaño de los tipos de datos no depende de la implementación de Java. Son siempre los mismos.

2. Variables

Una variable es un área en memoria que tiene un nombre y un Tipo asociado. El Tipo es o bien un Tipo primitivo o una Referencia.

Es obligatorio declarar las variables antes de usarlas. Para declararlas se indica su nombre y su Tipo, de la siguiente forma:

tipo_variable nombre ;

Ejemplos:

```
int i;      // Declaracion de un entero
char letra; // Declaracion de un caracter
boolean flag; // Declaracion de un booleano
```

- El ; es el separador de sentencias en Java.
- El símbolo // indica comentarios de línea.
- En Java las mayúsculas y minúsculas son significativas. No es lo mismo el nombre letra que Letra.

- Todas las palabras reservadas del lenguaje van en minúsculas.

Se pueden asignar valores a las variables mediante la instrucción de asignación.
Por ejemplo:

```
i = 5;           // a la variable i se le asigna el valor 5
letra = 'c';     // a la variable letra se le asigna el valor 'c'
flag = false;   // a la variable flag se le asigna el valor false
```

La declaración y la combinación se pueden combinar en una sola expresión:

```
int i = 5;
char letra = 'c';
boolean flag = false;
```

3. Literales

En los literales numéricos puede forzarse un tipo determinado con un sufijo:

- Entero largo: l ó L.
- Float: f ó F
- Double: d ó D.

Por ejemplo:

```
long l = 5L;
float numero = 5f;
```

En los literales numéricos para float y double puede usarse el exponente (10 elevado a...) de la forma: 1.5e3D (equivalente a 1500)

Literales hexadecimales: Al valor en hexadecimal se antepone el símbolo 0x. Por ejemplo: 0xf (valor 15)

Literales booleanos: **true** (verdadero) y **false** (falso)

Literales caracter: Un caracter entre apóstrofes (') o bien una secuencia de escape (también entre '). Las secuencias de escape están formadas por el símbolo \ y una letra o un número.

Algunas secuencias de escape útiles:

- \n: Salto de línea
- \t: Tabulador
- \b: Backspace.
- \r: Retorno de carro
- \uxxxx: donde xxxx es el código Unicode del carácter. Por ejemplo \u0027 es el apostrofe (')

4. Operaciones sobre Tipos primitivos

La siguiente tabla muestra un resumen de los operadores existentes para las distintas clases de tipos primitivos. El grupo 'Enteros' incluye byte, short, int, long y char. El grupo 'Coma flotante' incluye float and double.

Tipos	Grupo de operadores	Operadores
Enteros	Operadores de comparación que devuelven un valor boolean	< (menor) , <= (menor o igual) , > (mayor), >= (mayor o igual), == (igual), != (distinto)
	Operadores numéricos, que devuelven un valor int o long	+ (unario, positivo), - (unario, negativo), + (suma) , - (resta) , * (multiplicación), / (división), % (resto), ++ (incremento), -- (decremento), <<, >>, >>> (rotación) , ~ (complemento a nivel de bits), & (AND a nivel de bits), (OR a nivel de bits) ^ (XOR a nivel de bits)
Coma	Operadores de comparación que	< (menor) , <= (menor o igual)

flotante	devuelven un valor boolean	, > (mayor), >= (mayor o igual), == (igual), != (distinto)
	Operadores numéricos, que devuelven un valor float o double	+ (unario, positivo), - (unario, negativo), + (suma), - (resta), * (multiplicación), / (división), % (resto), ++ (incremento), -- (decremento).
Booleanos	Operadores booleanos	== (igual), != (distinto), ! (complemento), & (AND), (OR), ^ (XOR), && (AND condicional), (OR condicional)

5. Conversión entre tipos numéricos

Las normas de conversión entre tipos numéricos son las habituales en un lenguaje de programación: si en una operación se involucran varios datos numéricos de distintos tipos, todos ellos se convierten al tipo de dato que permite una mayor precisión y rango de representación numérica; así, por ejemplo:

- Si cualquier operando es double todos se convertirán en double.
- Si cualquier operando es float y no hay ningún double todos se convertirán a float.
- Si cualquier operando es long y no hay datos reales todos se convertirán en long.

Del mismo modo estas normas se extenderán para int, short y byte.

La "jerarquía" en las conversiones de mayor a menor es:

- double <- float <- long <- int <- short <- byte

Se deben tener en cuenta estas conversiones a la hora de mirar en que tipo de variable guardamos el resultado de la operación; ésta debe ser, al menos, de una jerarquía mayor o igual a la jerarquía de la máxima variable involucrada en la operación. Hay ciertas excepciones a la norma aquí descrita: Java solo tiene dos tipos de operadores enteros: uno que aplica para operar datos de tipo long, y otro que emplea para operar datos de tipo int. De este modo cuando operemos un byte con un byte, un short con un short o un short con un byte Java empleará para dicha operación el operador de los datos tipo int, por lo que el resultado de dicha operación será un int siempre.

Para indicar que una constante es de tipo long se debe indicar con una L: 23L.

Para indicar que una constante es flotante: ej: 2.3 se debe escribir: 2.3F, sino por defecto será double

Es posible convertir un dato de jerarquía "superior" a uno con jerarquía "inferior", arriesgándonos a perder información en el cambio. Este tipo de operación (almacenar el contenido de una variable de jerarquía superior en una de jerarquía inferior) se denomina cast.

Ejemplo:

```
public class ejemplo1 {  
    public static void main(String[] args) {  
        int i = 9,k;  
        float j = 47.9F;  
        System.out.println("i: " + i + " j: " + j);  
        k = (int)j; //empleo de un cast  
        System.out.println("j: " + j + " k: " + k);  
        j = k; //no necesita cast  
        System.out.println("j: " + j + " k: " + k);  
        float m = 2.3F;  
        //float m = 2.3; daría error al compilar.  
        System.out.println("m: " + m);  
    }  
}
```

6. Métodos

En Java toda la lógica de programación (Algoritmos) está agrupada en funciones o métodos.

Un método es:

- Un bloque de código que tiene un nombre,
- recibe unos parámetros o argumentos (opcionalmente),
- contiene sentencias o instrucciones para realizar algo (opcionalmente) y
- devuelve un valor de algún Tipo conocido (opcionalmente).

La sintaxis global es:

```
Tipo_Valor_devuelto nombre_método ( lista_argumentos ) {  
    bloque_de_codigo;  
}
```

Y la lista de argumentos se expresa declarando el tipo y nombre de los mismos (como en las declaraciones de variables). Si hay más de uno se separan por comas.

Por ejemplo:

```
int sumaEnteros ( int a, int b ) {  
    int c = a + b;  
    return c;  
}
```

- El método se llama sumaEnteros.
- Recibe dos parámetros también enteros. Sus nombres son a y b.
- Devuelve un entero.

En el ejemplo la cláusula **return** se usa para finalizar el método devolviendo el valor de la variable c.

7. El termino void

El hecho de que un método devuelva o no un valor es opcional. En caso de que devuelva un valor se declara el tipo que devuelve. Pero si no necesita ningún valor, se declara como tipo del valor devuelto, la palabra reservada **void**. Por ejemplo:

```
void haceAlgo()  
  
{  
    ...  
}
```

Cuando no se devuelve ningún valor, la cláusula **return** no es necesaria. Obsérvese que en el ejemplo el método haceAlgo tampoco recibe ningún parámetro. No obstante los paréntesis, son obligatorios.

8. Uso de métodos

Los métodos se invocan con su nombre, y pasando la lista de argumentos entre paréntesis. El conjunto se usa como si fuera una variable del Tipo devuelto por el método.

Por ejemplo:

```
int x;  
x = sumaEnteros(2,3);
```

Nota: Esta sintaxis no está completa, pero sirve para nuestros propósitos en este momento. La sintaxis completa se verá cuando se hable de objetos.

Aunque el método no reciba ningún argumento, los paréntesis en la llamada son obligatorios. Por ejemplo para llamar a la función `haceAlgo`, simplemente se pondría:

```
haceAlgo();
```

Observese que como la función tampoco devuelve ningún valor no se asigna a ninguna variable. (No hay nada que asignar).

9. Clases

Las clases son el mecanismo por el que se pueden crear nuevos Tipos en Java. Las clases son el punto central sobre el que giran la mayoría de los conceptos de la Orientación a Objetos.

Una clase es una agrupación de datos y de código que actúa sobre esos datos, a la que se le da un nombre.

Una clase contiene:

- Datos (se denominan Datos Miembro). Estos pueden ser de tipos primitivos o referencias.
- Métodos (se denominan Métodos Miembro).

La sintaxis general para la declaración de una clase es:

```
modificadores class nombre_clase {  
    declaraciones_de_miembros ;  
}
```

Por ejemplo:

```
class Punto {  
    int x;
```



```
int y;  
}
```

Los modificadores son palabras clave que afectan al comportamiento de la clase. Los iremos viendo progresivamente en los sucesivos capítulos.

10. Objetos, miembros y referencias

Un objeto es una instancia (ejemplar) de una clase. La clase es la definición general y el objeto es la materialización concreta (en la memoria del ordenador) de una clase.

El fenómeno de crear objetos de una clase se llama instanciación.

Los objetos se manipulan con referencias. Una referencia es una variable que apunta a un objeto. Las referencias se declaran igual que las variables de Tipos primitivos (tipo nombre). Los objetos se crean (se instancian) con el operador de instanciación **new**.

Ejemplo:

```
Punto p;  
p = new Punto();
```

La primera línea del ejemplo declara una referencia (p) que es de Tipo Punto. La referencia no apunta a ningún sitio. En la segunda línea se crea un objeto de Tipo Punto y se hace que la referencia p apunte a él. Se puede hacer ambas operaciones en la misma expresión:

```
Punto p = new Punto();
```

A los miembros de un objeto se accede a través de su referencia. La sintaxis es:

nombre_referencia.miembro

En el ejemplo, se puede poner:

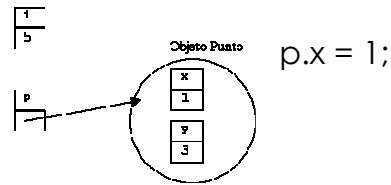
```
p.x = 1;  
p.y = 3;
```

Se puede visualizar gráficamente los datos primitivos, referencias y objetos de la siguiente forma:

- Datos primitivos: **int** i = 5;

- Referencias y objetos:

```
Punto p = new Punto();
p.y = 3;
```



Es importante señalar que en el ejemplo, p no es el objeto. Es una referencia que apunta al objeto.

Los métodos miembro se declaran dentro de la declaración de la clase, tal como se ha visto en el capítulo anterior. Por ejemplo:

```
class Circulo {
    Punto centro; // dato miembro. Referencia a un objeto punto
    int radio;     // dato miembro. Valor primitivo
    float superficie() { // método miembro.
        return 3.14 * radio * radio;
    }             // fin del método superficie
}                // fin de la clase Circulo
```

El acceso a métodos miembros es igual que el que ya se ha visto para datos miembro. En el ejemplo:

```
Circulo c = new Circulo();
c.centro.x = 2;
c.centro.y = 3;
c.radio = 5;
float s = c.superficie();
```

Es importante observar en el ejemplo:

- Los datos miembro pueden ser tanto primitivos como referencias. La clase Circulo contiene un dato miembro de tipo Punto (que es el centro del círculo).
- El acceso a los datos miembros del Punto centro se hace encadenando el operador . en la expresión c.centro.x que se podría leer como 'el miembro x del objeto (Punto) centro del objeto (Circulo) c'.
- Aunque el método superficie no recibe ningún argumento los paréntesis son obligatorios (Distinguen los datos de los métodos).

- Existe un Objeto Punto para cada instancia de la clase Circulo (que se crea cuando se crea el objeto Circulo).

11.1 Conceptos básicos

- Una Clase es una definición de un nuevo Tipo, al que se da un nombre.
- Una Clase contiene Datos Miembro y Métodos Miembro que configuran el estado y las operaciones que puede realizar.
- Un Objeto es la materialización (instanciación) de una clase. Puede haber tantos Objetos de una Clase como resulte necesario.
- Los Objetos se crean (se les asigna memoria) con el Operador **new**.
- Los Objetos se manipulan con Referencias.
- Una Referencia es una Variable que apunta a un Objeto.
- El acceso a los elementos de un Objeto (Datos o métodos) se hace con el operador . (punto) : *nombre_referencia.miembro*

12. Constructores

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando un método especial denominado constructor. Esta invocación es implícita y se realiza automáticamente cuando se utiliza el operador **new**. Los constructores tienen algunas características especiales:

- El nombre del constructor tiene que ser igual al de la clase.
- Puede recibir cualquier número de argumentos de cualquier tipo, como cualquier otro método.
- No devuelve ningún valor (en su declaración no se declara ni siquiera **void**).

El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto (con el operador **new**). No puede invocarse explícitamente en ningún otro momento.

Continuando con los ejemplos del capítulo anterior se podría escribir un constructor para la clase Punto, de la siguiente forma:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
}
```

```
}  
}
```

Con este constructor se crearía un objeto de la clase Punto de la siguiente forma:

```
Punto p = new Punto ( 1 , 2 );
```

12.1 Constructor no-args.

Si una clase no declara ningún constructor, Java incorpora un constructor por defecto (denominado constructor no-args) que no recibe ningún argumento y no hace nada.

Si se declara algún constructor, entonces ya no se puede usar el constructor no-args. Es necesario usar el constructor declarado en la clase.

En el ejemplo el constructor no-args sería:

```
class Punto {  
    int x , y ;  
    Punto ( ) { }  
}
```

12.2 Sobrecarga de constructores.

Una clase puede definir varios constructores (un objeto puede inicializarse de varias formas). Para cada instanciación se usa el que coincide en número y tipo de argumentos. Si no hay ninguno coincidente se produce un error en tiempo de compilación.

Por ejemplo:

```
class Punto {  
    int x , y ;  
    Punto ( int a , int b ) {  
        x = a ; y = b ;  
    }  
}
```

```

    }
    Punto () {
        x = 0 ; y = 0;
    }
}

```

En el ejemplo se definen dos constructores. El citado en el ejemplo anterior y un segundo que no recibe argumentos e inicializa las variables miembro a 0. (Nota: veremos más adelante que este tipo de inicialización es innecesario, pero para nuestro ejemplo sirve).

Desde un constructor puede invocarse explícitamente a otro constructor utilizando la palabra reservada **this** . **this** es una referencia al propio objeto. Cuando **this** es seguido por paréntesis se entiende que se está invocando al constructor del objeto en cuestión.

El ejemplo anterior puede reescribirse de la siguiente forma:

```

class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
    Punto () {
        this (0,0);
    }
}

```

Cuando se declaran varios constructores para una misma clase estos deben distinguirse en la lista de argumentos, bien en el número, bien en el tipo.

Esta característica de definir métodos con el mismo nombre se denomina sobrecarga y es aplicable a cualquier método miembro de una clase.

13. Datos estáticos

Un dato estático es una variable miembro que no se asocia a un objeto (instancia) de una clase, sino que se asocia a la clase misma; no hay una copia del dato para cada objeto sino una sola copia que es compartida por todos los objetos de la clase.

Por ejemplo:

```
class Punto {  
    int x , y ;  
    static int numPuntos = 0;  
  
    Punto ( int a , int b ) {  
        x = a ; y = b;  
        numPuntos ++ ;  
    }  
}
```

En el ejemplo numPuntos es un contador que se incrementa cada vez que se crea una instancia de la clase Punto.

Observe la forma en que se declara numPuntos, colocando el modificador **static** delante del tipo. La sintaxis general para declarar una variable es:

[modificadores] tipo_variable nombre;

static es un modificador. En los siguientes capítulos se irán viendo otros modificadores. Los [] en la expresión anterior quieren decir que los modificadores son opcionales.

El acceso a las variables estáticas desde fuera de la clase donde se definen se realiza a través del nombre de la clase y no del nombre del objeto como sucede con las variables miembro normales (no estáticas). En el ejemplo anterior puede escribirse:

int x = Punto.numPuntos;

No obstante también es posible acceder a las variables estáticas a través de una referencia a un objeto de la clase. Por ejemplo:

```
Punto p = new Punto();  
int x = p.numPuntos;
```

Las variables estáticas de una clase existen, se inicializan y pueden usarse antes de que se cree ningún objeto de la clase.

14. Métodos estáticos

Para los métodos, la idea es la misma que para los datos: los métodos estáticos se asocian a una clase, no a una instancia.

Por ejemplo:

```

class Punto {
    int x , y ;
    static int numPuntos = 0;

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }

    static int cuantosPuntos() {
        return numPuntos;
    }
}

```

La sintaxis general para la definición de los métodos es, por tanto, la siguiente:

```

[modificadores] Tipo_Valor_devuelto nombre_método ( lista_argumentos ) {
    bloque_de_codigo;
}

```

El acceso a los métodos estáticos se hace igual que a los datos estáticos, es decir, usando el nombre de la clase, en lugar de usar una referencia:

```

int totalPuntos = Punto.cuantosPuntos();

```

Dado que los métodos estáticos tienen sentido a nivel de clase y no a nivel de objeto (instancia) los métodos estáticos no pueden acceder a datos miembros que no sean estáticos.

14.1 El método main

Un programa Java se inicia proporcionando al intérprete Java un nombre de clase. La JVM carga en memoria la clase indicada e inicia su ejecución por un método estático que debe estar codificado en esa clase. El nombre de este método es main y debe declararse de la siguiente forma:

static void main (String [] args)

- Es un método estático. Se aplica por tanto a la clase y no a una instancia en particular, lo que es conveniente puesto que en el momento de iniciar la ejecución todavía no se ha creado ninguna instancia de ninguna clase.
- Recibe un argumento de tipo String []. String es una clase que representa una cadena de caracteres (se verá más adelante),
- Los corchetes [] indican que se trata de un array que se verán en un capítulo posterior.

No es obligatorio que todas las clases declaren un método main . Sólo aquellos métodos que vayan a ser invocados directamente desde la línea de comandos de la JVM necesitan tenerlo. En la práctica la mayor parte de las clases no lo tienen.

14.2 Inicializadores estáticos

En ocasiones es necesario escribir código para inicializar los datos estáticos, quizá creando algún otro objeto de alguna clase o realizando algún tipo de control. El fragmento de código que realiza esta tarea se llama inicializador estático. Es un bloque de sentencias que no tiene nombre, ni recibe argumentos, ni devuelve valor. Simplemente se declara con el modificador **static** .

La forma de declarar el inicializador estático es:

static { *bloque_codigo* }

Por ejemplo:

```
class Punto {  
    int x , y ;
```



```
static int numPuntos;
```

```
static {  
    numPuntos = 0;  
}
```

```
Punto ( int a , int b ) {  
    x = a ; y = b;  
    numPuntos ++ ;  
  
}
```

```
static int cuantosPuntos() {  
    return numPuntos;  
}  
}
```

Nota: El ejemplo, una vez más, muestra sólo la sintaxis y forma de codificación. Es innecesario inicializar la variable. Además podría inicializarse directamente con:
static int numPuntos = 0;

15. Inicialización de variables

Desde el punto de vista del lugar donde se declaran existen dos tipos de variables:

- Variables miembro: Se declaran en una clase, fuera de cualquier método.
- Variables locales: Se declaran y usan en un bloque de código dentro de un método.

Las variables miembro son inicializadas automáticamente, de la siguiente forma:

- Las numéricas a 0.
- Las booleanas a **false**.
- Las char al caracter nulo (hexadecimal 0).
- Las referencias a **null**.

Nota: **null** es un literal que indica referencia nula.

Las variables miembro pueden inicializarse con valores distintos de los anteriores en su declaración.

Las variables locales no se inicializan automáticamente. Se debe asignarles un valor antes de ser usadas. Si el compilador detecta una variable local que se usa antes de que se le asigne un valor produce un error.

Por ejemplo:

```
int p;  
int q = p;  // error
```

El compilador también produce un error si se intenta usar una variable local que podría no haberse inicializado, dependiendo del flujo de ejecución del programa. Por ejemplo:

```
int p;  
if ( . . . ) {  
    p = 5 ;  
}  
int q = p;  // error
```

El compilador produce un error del tipo 'La variable podría no haber sido inicializada', independientemente de la condición del if.

15.1 Ambito de las variables

El ámbito de una variable es el área del programa donde la variable existe y puede ser utilizada. Fuera de ese ámbito la variable, o bien no existe o no puede ser usada (que viene a ser lo mismo).

El ámbito de una variable miembro (que pertenece a un objeto) es el de la usabilidad de un objeto. Un objeto es utilizable desde el momento en que se crea y mientras existe una referencia que apunte a él.

Cuando la última referencia que lo apunta sale de su ámbito el objeto queda 'perdido' y el espacio de memoria ocupado por el objeto puede ser recuperado por la JVM cuando lo considere oportuno. Esta recuperación de espacio en memoria se denomina 'recogida de basura'.

El ámbito de las variables locales es el bloque de código donde se declaran. Fuera de ese bloque la variable es desconocida.

Ejemplo:

```

{
    int x;    // empieza el ámbito de x. (x es conocida y utilizable)
    {
        int q; // empieza el ámbito de q. x sigue siendo conocida.
        ...
    }        // finaliza el ámbito de q (termina el bloque de código)
    ...      // q ya no es utilizable
}           // finaliza el ámbito de x

```

16. Recogida de basura

Cuando ya no se necesita un objeto simplemente puede dejar de referenciarse. No existe una operación explícita para 'destruir' un objeto o liberar el área de memoria usada por él.

La liberación de memoria la realiza el recolector de basura (*garbage collector*) que es una función de la JVM. El recolector revisa toda el área de memoria del programa y determina que objetos pueden ser borrados porque ya no tienen referencias activas que los apunten. El recolector de basura actúa cuando la JVM lo determina (tiene un mecanismo de actuación no trivial).

En ocasiones es necesario realizar alguna acción asociada a la acción de liberar la memoria asignada al objeto (como por ejemplo liberar otros recursos del sistema, como descriptores de ficheros). Esto puede hacerse codificando un método `finalize` que debe declararse como:

```
protected void finalize() throws Throwable { }
```

El método `finalize` es invocado por la JVM antes de liberar la memoria por el recolector de basura, o antes de terminar la JVM. No existe un momento concreto en que las áreas de memoria son liberadas, sino que lo determina en cada momento la JVM en función de sus necesidades de espacio.

17. Sobrecarga de métodos

Una misma clase puede tener varios métodos con el mismo nombre siempre que se diferencien en el tipo o número de los argumentos. Cuando esto sucede se dice que el método está sobrecargado. Por ejemplo, una misma clase podría tener los métodos:

```

int metodoSobrecargado() { ... }
int metodoSobrecargado(int x) { ... }

```

Sin embargo no se puede sobrecargar cambiando sólo el tipo del valor devuelto. Por ejemplo:

```
int metodoSobrecargado() { . . . }  
void metodoSobrecargado() { . . . } // error en compilación
```

con esta definición, en la expresión `y.metodoSobrecargado()` la JVM no sabría que método invocar.

Se puede sobrecargar cualquier método miembro de una clase, así como el constructor.

18. La referencia **this**

En ocasiones es conveniente disponer de una referencia que apunte al propio objeto que se está manipulando. Esto se consigue con la palabra reservada **this**. **this** es una referencia implícita que tienen todos los objetos y que apunta a sí mismo. Por ejemplo:

```
class Circulo {  
    Punto centro;  
    int radio;  
    . . .  
    Circulo elMayor(Circulo c) {  
        if (radio > c.radio) return this;  
        else return c;  
    }  
}
```

El método `elMayor` devuelve una referencia al círculo que tiene mayor radio, comparando los radios del `Circulo c` que se recibe como argumento y el propio. En caso de que el propio resulte mayor el método debe devolver una referencia a sí mismo. Esto se consigue con la expresión **return this**.

19. La referencia null

Para asignar a una referencia el valor nulo se utiliza la constante null. El ejemplo del caso anterior se podría completar con:

```
class Circulo {  
    Punto centro;  
    int radio;  
    ...  
    Circulo elMayor(Circulo c) {  
        if (radio > c.radio) return this;  
        else if (c.radio > radio) return c;  
        else return null;  
    }  
}
```

20. Ocultamiento de variables

Puede ocurrir que una variable local y una variable miembro reciban el mismo nombre (en muchos casos por error). Cuando se produce esto la variable miembro queda oculta por la variable local, durante el bloque de código en que la variable local existe y es accesible. Cuando se sale fuera del ámbito de la variable local, entonces la variable miembro queda accesible.

Observe esto en el ejemplo siguiente:

```
...  
String x = "Variable miembro";  
...  
void variableOculta() {  
    System.out.println(x);  
    {  
        String x = "Variable local";  
        System.out.println(x);  
    }  
    System.out.println(x);  
}
```

La llamada al método variableOculta() producirá la siguiente salida:

```
Variable miembro  
Variable local  
Variable miembro
```

Se puede acceder a la variable miembro oculta usando la referencia this. En el ejemplo anterior la expresión:

```
System.out.println(this.x);
```

siempre producirá la salida 'Variable miembro', puesto que this.x se refiere siempre a la variable miembro.

21. Operadores

La siguiente tabla muestra un resumen de operadores clasificados por grupos:

Grupo de operador	Operador	Significado
Aritméticos	+	Suma
	-	Resta
	*	Multiplicación
	/	División
	%	Resto
Relacionales	>	Mayor
	>=	Mayor o igual
	<	Menor
	<=	Menor o igual
	==	Igual
	!=	Distinto
Logicos	&&	AND
		OR
	!	NOT
A nivel de bits	&	AND
		OR
	^	NOT
	<<	Desplazamiento a la izquierda
	>>	Desplazamiento a la derecha rellenando con 1
	>>>	Desplazamiento a la derecha rellenando con 0
Otros	+	Concatenación de

++	cadenas
--	Autoincremento (actua
=	como prefijo o sufijo)
+=	Autodecremento (actua
-=	como preficjo o sufijo)
*=	Asignación
/=	Suma y asignación
?:	Resta y asignación
	Multiplicación y asignación
	División y asignación
	Condicional

21. Funciones de Entrada y Salida

21.1 Entrada por teclado en Java

Se utiliza System.in. Es habitual introducido en un InputStreamReader seguido de un BufferedReader por razones de eficiencia. Las cadenas de caracteres se pueden leer con el método `readLine()`. Para leer números se debe analizar sintácticamente la cadena leída, por ejemplo con `parseInt()` o `parseDouble()`.

```
BufferedReader ent = new BufferedReader(new InputStreamReader(System.in));
String cadReal = ent.readLine();
double real = Double.parseDouble(cadReal);
```

Se pueden leer varios valores de una línea de entrada poniendo la línea en un objeto de tipo String y usando a continuació

21.2 Salida por pantalla en Java

Se usan los métodos `print()` y `println()` de System.out para imprimir cadenas de caracteres; suponen la existencia de un método `toString()` para convertir objetos en cadenas de caracteres.

"\n" avanza a una nueva línea; `println()` también produce un salto de línea después de producir la salida.

```
System.out.print("Hola");
System.out.println(" amigo");
```

Dar formato a la salida requiere clases especiales como `DecimalFormat`.

21.3 Entrada por archivo

Es habitual introducir un `FileReader` en un `BufferedReader` para crear objetos de lectura a través de archivo. Entonces se pueden leer cadenas de caracteres con `readLine()`; y analizarlas sintácticamente para obtener valores numéricos de forma análoga a la descrita en la entrada por teclado. Esto debe hacerse en un bloque `try`, ya que se pueden lanzar excepciones.

```
try{
BufferedReader fent =
new BufferedReader(new FileReader(nombreFich));
String cadReal = fent.readLine();
double real = Double.parseDouble(cadReal);
}
catch {
//
```

21.4 Salida por archivo en Java

Es habitual introducir un objeto `FileWriter` en un `BufferedWriter` que a su vez se introduce en un `PrintWriter`. Se usan los métodos `print()` y `println()` para escribir en la salida:

```
PrintWriter fsal = new PrintWriter(
new BufferedWriter(new FileWriter(nombreFich)));
fsal.println("Sueldo" + sueldo);
```

21.5 La clase System

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

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

21.6 Stdout

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

21.7 Stderr

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

Ejemplo de entrada/salida en Java. El código siguiente, miType.java[∞], reproduce, o funciona como la utilidad cat de Unix o type de DOS:

```
import java.io.*; class miType { public static void main( String args[] ) throws
IOException { int c; int contador = 0; while( (c = System.in.read() ) != '\n' ) {
contador; System.out.print( (char)c ); } System.out.println(); Línea en blanco
System.err.println( "Contados "+ contador +" bytes en total." ); }}
```

21.8 Clases comunes de Entrada/Salida

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

FileInputStream DataInputStream FileOutputStream DataOutputStream

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

21.9 PipedInputStream

21.12 StreamTokenizer

21.10 BufferedInputStream

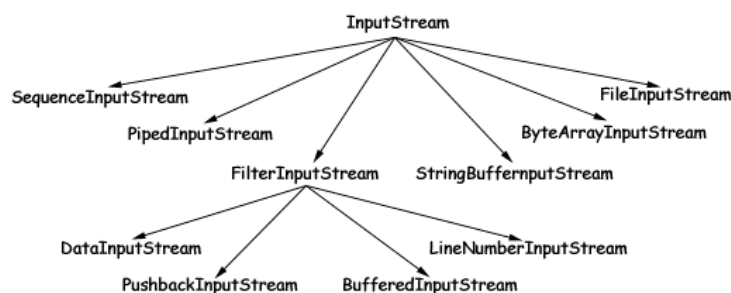
21.13 PipedOutputStream

21.11 PushBackInputStream

21.14 BufferedOutputStream

21.9 Streams de Entrada

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



▪ **Objetos *FileInputStream***

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

▪ **Apertura de un *FileInputStream***

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

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

También se puede utilizar:

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

▪ **Lectura de un *FileInputStream***

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

```
int read();
```

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

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

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

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

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

▪ Cierre de `FileInputStream`

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

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

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

Ejemplo: Visualización de un fichero

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

```
FileInputStream fis;
TextArea ta;

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

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

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

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

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

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

▪ **Objetos DataInputStream**

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

▪ **Apertura y cierre de DataInputStream**

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

```
DataInputStream miDStream;  
FileInputStream miFStream;  
  
// Obtiene un controlador de fichero  
miFStream = new FileInputStream("/etc/ejemplo.dbf");  
//Encadena un fichero de entrada de datos  
miDStream = new DataInputStream( miFStream );  
  
// Ahora se pueden utilizar los dos streams de entrada para  
// acceder al fichero (si se quiere...)  
miFStream.read( b );  
i = miDStream.readInt();  
  
// Cierra el fichero de datos explícitamente  
//Siempre se cierra primero el fichero stream de mayor nivel  
miDStream.close();  
miFStream.close();
```

▪ **Lectura de un DataInputStream**

Al acceder a un fichero como DataInputStream, se pueden utilizar los mismos métodos *read()* de los objetos FileInputStream. No obstante, también se tiene acceso a otros métodos diseñados para leer cada uno de los tipos de datos:

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

Cada método leerá un objeto del tipo pedido.

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

Para leer un *long*, por ejemplo:

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

▪ Streams de entrada de URLs

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

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

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

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

▪ Apertura de un Stream de entrada de URL

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

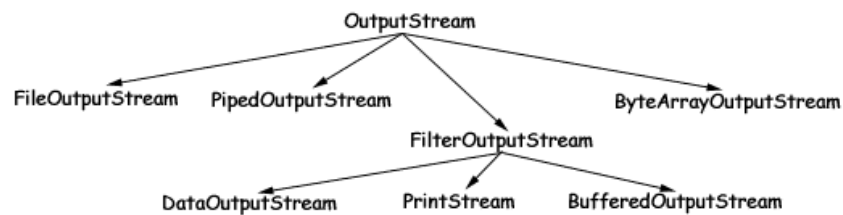
```
InputStream is;
byte buffer[] = new byte[24];
is = new URL( getDocumentBase(),datos).openStream();
```

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

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

21.10 Streams de Salida

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



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

▪ Objetos `FileOutputStream`

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

▪ Apertura de un `FileOutputStream`

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

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

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

```
File miFichero FileOutputStream miFicheroSt;
```

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

▪ **Escritura en un *FileOutputStream***

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

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

Escribe un byte.

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

Escribe todo el array, si es posible.

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

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

▪ **Cierre de *FileOutputStream***

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

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

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

Ejemplo: Almacenamiento de Información. Este programa, *Telefonos.java*, pregunta al usuario una lista de nombres y números de teléfono. Cada nombre y número se añade a un fichero situado en una localización fija. Para indicar que se ha introducido toda la lista, el usuario especifica "Fin" ante la solicitud de entrada del nombre. Una vez que el usuario ha terminado de teclear la lista, el programa creará un fichero de salida que se mostrará en pantalla o se imprimirá. Por ejemplo:

95-4751232,Juanito
564878,Luisa
123456,Pepe

347698,Antonio
91-3547621,Maria

El código fuente del programa es el siguiente:

```
import java.io.*;

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

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

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

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

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

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


- **Streams de salida con buffer**

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

- **Creación de Streams de salida con buffer**

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

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

- **Volcado y Cierre de Streams de salida con buffer**

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

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

- **Streams DataOutput**

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

- **Apertura y cierre de objetos `DataOutputStream`**

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

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

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

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

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

- **Escritura en un objeto `DataOutputStream`**

Cada uno de los métodos `write()` accesibles por los `FileOutputStream` también lo son a través de los `DataOutputStream`. También encontrará métodos complementarios a los de `DataInputStream`:

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

Para las cadenas, se tienen dos posibilidades: bytes y caracteres. Hay que recordar que los bytes son objetos de 8 bits y los caracteres lo son de 16 bits. Si

nuestras cadenas utilizan caracteres Unicode, debemos escribirlas con `writeChars()`.

- **Contabilidad de la salida**

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

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

22. Ejecución condicional

El formato general es:

```
if (expresion_booleana)
    sentencia
[else
    sentencia]
```

sentencia (a todo lo largo de este capítulo) puede ser una sola sentencia o un bloque de sentencias separadas por ; y enmarcadas por llaves { y }. Es decir

```
if (expresion_booleana) {
    sentencia;
    sentencia;
    ...
}
else {
    sentencia;
    sentencia;
    ...
}
```

expresion_booleana es una expresión que se evalúa como **true** o **false** (es decir, de tipo booleano). Si el resultado es true la ejecución bifurca a la sentencia que sigue al **if**. En caso contrario se bifurca a la sentencia que sigue al **else**.

Los corchetes en el formato anterior indican que la cláusula **else** es opcional.

23. Iteraciones con while

Sintaxis formato 1:

```
while (expresion_booleana)  
    sentencia
```

Sintaxis formato 2:

```
do  
    sentencia  
while (expresion_booleana)
```

La sentencia o bloque se sentencias (se aplica la misma idea que para el if-else) se ejecuta mientras que la *expresion_booleana* se evalúe como **true**

La diferencia entre ambos formatos es que en el primero la expresión se evalúa al principio del bloque de sentencias y en el segundo se evalúa al final.

24. Iteraciones con for

El formato es:

```
for ( inicializacion ; expresion_booleana ; step )  
    sentencia
```

inicializacion es una sentencia que se ejecuta la primera vez que se entra en el bucle **for** . Normalmente es una asignación. Es opcional.

expresion_booleana es una expresión que se evalúa antes de la ejecución de la sentencia, o bloque de sentencias, para cada iteración. La sentencia o bloque de sentencias se ejecutan mientras que la *expresion_booleana* se evalúe como cierta. Es opcional.

step es una sentencia que se ejecuta cada vez que se llega al final de la sentencia o bloque de sentencias. Es opcional.

Una utilización clásica de un bucle de tipo for se muestra a continuación para evaluar un contador un número fijo de veces:

```
for ( int i = 1 ; i <= 10 ; i++ )  
    sentencia
```

La sentencia (o bloque de sentencias) se evaluará 10 veces. En cada ejecución (pasada) el valor de la variable *i* irá variando desde 1 hasta 10 (inclusive). Cuando salga del bloque de sentencias *i* estará fuera de su ámbito (porque se define en el bloque **for**).

Si se omiten las tres clausulas del bucle se obtiene un bucle infinito:

```
for ( ; ; )  
    sentencia
```

Obsérvese que se pueden omitir las clausulas pero no los separadores (;).

25. Evaluación múltiple

El formato es:

```
switch ( expresion_entera ) {  
    case valor_entero:  
        sentencia;  
        break;  
    case valor_entero:  
        sentencia;  
        break;  
    ...  
    default:  
        sentencia;  
}
```

En el **switch** la expresión que se evalúa no es una expresión booleana como en el if-else, sino una expresión entera.

Se ejecuta el bloque **case** cuyo valor coincida con el resultado de la expresión entera de la clausula **switch** . Se ejecuta hasta que se encuentra una sentencia **break** o se llega al final del **switch** .

Si ningún valor de **case** coincide con el resultado de la expresión entera se ejecuta el bloque **default**(si está presente).

default y **break** son opcionales.

26. Devolución de control

El formato es:

return *valor*

Se utiliza en los métodos para terminar la ejecución y devolver un valor a quien lo llamó.

valor debe ser del tipo declarado en el método.

valor es opcional. No debe existir cuando el método se declara de tipo **void**. En este caso, la cláusula **return** al final del método es opcional, pero puede usarse para devolver el control al llamador en cualquier momento.

27. Expresiones

La mayor parte del trabajo en un programa se hace mediante la evaluación de expresiones, bien por sus efectos tales como asignaciones a variables, bien por sus valores, que pueden ser usados como argumentos u operandos en expresiones mayores, o afectar a la secuencia de ejecución de instrucciones.

Cuando se evalúa una expresión en un programa el resultado puede denotar una de tres cosas:

- Una variable. (Si por ejemplo es una asignación)
- Un valor. (Por ejemplo una expresión aritmética, booleana, una llamada a un método, etc.)
- Nada. (Por ejemplo una llamada a un método declarado void)

Si la expresión denota una variable o un valor, entonces la expresión tiene siempre un tipo conocido en el momento de la compilación. Las reglas para determinar el tipo de la expresión varían dependiendo de la forma de las expresiones pero resultan bastante naturales. Por ejemplo, en una expresión aritmética con operandos de diversas precisiones el resultado es de un tipo tal que no se produzca pérdida de información, realizándose internamente las conversiones necesarias. El análisis pormenorizado de las conversiones de tipos, evaluaciones de expresiones, etc, queda fuera del ámbito de estos apuntes. En general puede decirse que es bastante similar a otros lenguajes, en particular C, teniendo en

cuenta la característica primordial de Java de tratarse de un lenguaje con control fuerte de tipos.

28. Arrays

Un array es una colección ordenada de elementos del mismo tipo, que son accesibles a través de un índice.

Un array puede contener datos primitivos o referencias a objetos.

Los arrays se declaran:

[modificadores] tipo_variable [] nombre;

Por ejemplo:

```
int [ ] a;  
Punto [ ] p;
```

La declaración dice que a es un array de enteros y p un array de objetos de tipo Punto. Más exactamente a es una referencia a una colección de enteros, aunque todavía no se sabe cuantos elementos tiene el array. p es una referencia a una colección de referencias que apuntarán objetos Punto.

Un array se crea como si se tratara de un objeto (de hecho las variables de tipo array son referencias):

```
a = new int [5];  
p = new Punto[3];
```

En el momento de la creación del array se dimensiona el mismo y se reserva la memoria necesaria.

También puede crearse de forma explícita asignando valores a todos los elementos del array en el momento de la declaración, de la siguiente forma:

```
int [ ] a = { 5 , 3 , 2 };
```

El acceso a los elementos del array se realiza indicando entre corchetes el elemento del array que se desea, teniendo en cuenta que siempre el primer

elemento del array es el índice 0. Por ejemplo `a[1]`. En este ejemplo los índices del array de tres elementos son 0, 1 y 2. Si se intenta usar un índice que está fuera del rango válido para ese array se produce un error (en realidad una excepción. En el ejemplo anterior se produce esta excepción si el índice es menor que 0 o mayor que 2.

Se puede conocer el número de elementos de un array usando la variable `length`. En el ejemplo `a.length` contiene el valor 3.

Un array, como cualquier otra referencia puede formar parte de la lista de parámetros o constituir el valor de retorno de un método. En ambos casos se indica que se trata de un array con los corchetes que siguen al tipo. Por ejemplo:

```
String [ ] metodoConArrays ( Punto [ ] ) { . . }
```

El método `metodoConArrays` recibe como parámetro un array de Puntos y devuelve un array de Strings. El método podría invocarse de la siguiente forma:

```
Punto p [ ] = new Punto [3];  
...  
String [ ] resultado = metodoConArrays(p);
```

29. Arrays multidimensionales

Es posible declarar arrays de más de una dimensión. Los conceptos son los mismos que para los arrays monodimensionales.

Por ejemplo:

```
int [ ] [ ] a = { { 1 , 2 } , { 3 , 4 } , { 5 , 6 } };  
int x = a[1][0]; // contiene 3  
int y = a[2][1]; // contiene 6
```

Se pueden recorrer los elementos de un array multidimensional, de la siguiente forma:

```
int [ ] [ ] a = new int [3][2];  
for ( int i = 0 ; i < a.length ; i++ )  
    for ( int j = 0 ; j < a[i].length ; j++ )  
        a[i][j] = i * j;
```

Obsérvese en el ejemplo la forma de acceder al tamaño de cada dimensión del array.

30. Strings

En Java no existe un tipo de datos primitivo que sirva para la manipulación de cadenas de caracteres. En su lugar se utiliza una clase definida en la API que es la clase `String`. Esto significa que en Java las cadenas de caracteres son, a todos los efectos, objetos que se manipulan como tales, aunque existen ciertas operaciones, como la creación de `Strings`, para los que el lenguaje tiene soporte directo, con lo que se simplifican algunas operaciones.

La clase `String` forma parte del package `java.lang` y se describe completamente en la documentación del API del JDK.

a. Creación de Strings

Un `String` puede crearse como se crea cualquier otro objeto de cualquier clase; mediante el operador `new`:

```
String s = new String("Esto es una cadena de caracteres");
```

Observese que los literales de cadena de caracteres se indican entre comillas dobles (`"`), a diferencia de los caracteres, que utilizan comillas simples (`'`).

Sin embargo también es posible crear un `String` directamente, sin usar el operador `new`, haciendo una asignación simple (como si se tratara de un dato primitivo):

```
String s = "Esto es una cadena de caracteres";
```

Ambas expresiones conducen al mismo objeto.

Los `Strings` no se modifican una vez que se les ha asignado valor. Si se produce una reasignación se crea un nuevo objeto `String` con el nuevo contenido.

Además la clase `String` proporciona constructores para crear `Strings` a partir de arrays de caracteres y arrays de bytes. Consultar la documentación del API del JDK para más detalles.

b. Concatenación de Strings

Java define el operador `+` (suma) con un significado especial cuando los operandos son de tipo `String`. En este caso el operador suma significa concatenación. El resultado de la concatenación es un nuevo `String` compuesto por las dos cadenas, una tras otra. Por ejemplo:

```
String x = "Concatenar" + "Cadenas";
```

da como resultado el String "ConcatenarCadenas".

También es posible concatenar a un String datos primitivos, tanto numéricos como booleanos y char. Por ejemplo, se puede usar:

```
int i = 5;  
String x = "El valor de i es " + i;
```

Cuando se usa el operador + y una de las variables de la expresión es un String, Java transforma la otra variable (si es de tipo primitivo) en un String y las concatena. Si la otra variable es una referencia a un objeto entonces invoca el método toString() que existe en todas las clases (es un método de la clase Object).

c. Otros métodos de la clase String

La clase String dispone de una amplia gama de métodos para la manipulación de las cadenas de caracteres. Para una referencia completa consultar la documentación del API del JDK. El siguiente cuadro muestra un resumen con algunos de los métodos más significativos:

Método	Descripción
char charAt(int index)	Devuelve el carácter en la posición indicada por index. El rango de index va de 0 a length() - 1.
boolean equals(Object obj)	Compara el String con el objeto especificado. El resultado es true si y solo si el argumento es no nulo y es un objeto String que contiene la misma secuencia de caracteres.
boolean equalsIgnoreCase(String s)	Compara el String con otro, ignorando consideraciones de mayúsculas y minúsculas. Los dos Strings se consideran iguales si tienen la misma longitud y, los caracteres correspondientes en ambos Strings son iguales sin tener en cuenta mayúsculas y minúsculas.

<code>int indexOf(char c)</code>	Devuelve el índice donde se produce la primera aparición de c. Devuelve -1 si c no está en el string.
<code>int indexOf(String s)</code>	Igual que el anterior pero buscando la subcadena representada por s.
<code>int length()</code>	Devuelve la longitud del String (número de caracteres)
<code>String substring(int begin, int end)</code>	Devuelve un substring desde el índice begin hasta el end
<code>static String valueOf(int i)</code>	Devuelve un string que es la representación del entero i. Obsérvese que este método es estático. Hay métodos equivalentes donde el argumento es un float, double, etc.
<code>char[] toCharArray()</code> <code>String toLowerCase()</code> <code>String toUpperCase()</code>	Transforman el string en un array de caracteres, o a mayúsculas o a minúsculas.

d. La clase StringBuffer

Dado que la clase String sólo manipula cadenas de caracteres constantes resulta poco conveniente cuando se precisa manipular intensivamente cadenas (reemplazando caracteres, añadiendo o suprimiendo, etc.). Cuando esto es necesario puede usarse la clase StringBuffer definida también en el package `java.lang.` del API. Esta clase implanta un buffer dinámico y tiene métodos que permiten su manipulación comodamente. Ver la documentación del API.

31. Packages

Un package es una agrupación de clases afines. Equivale al concepto de librería existente en otros lenguajes o sistemas. Una clase puede definirse como perteneciente a un package y puede usar otras clases definidas en ese o en otros packages.

Los packages delimitan el espacio de nombres (space name). El nombre de una clase debe ser único dentro del package donde se define. Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser

usadas en el mismo programa. Una clase se declara perteneciente a un package con la cláusula `package`, cuya sintaxis es:

```
package nombre_package;
```

La cláusula **package** debe ser la primera sentencia del archivo fuente. Cualquier clase declarada en ese archivo pertenece al package indicado.

Por ejemplo, un archivo que contenga las sentencias:

```
package miPackage;
```

```
...
```

```
class miClase {
```

```
...
```

declara que la clase `miClase` pertenece al package `miPackage`.

La cláusula `package` es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto sin nombre.

La agrupación de clases en packages es conveniente desde el punto de vista organizativo, para mantener bajo una ubicación común clases relacionadas que cooperan desde algún punto de vista. También resulta importante por la implicación que los packages tienen en los modificadores de acceso.

a. Cláusula `import`

Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
package Geometria;
```

```
...
```

```
class Circulo {  
    Punto centro;
```

```
    ...
```

```
}
```

En esta declaración definimos la clase `Circulo` perteneciente al package `Geometria`. Esta clase usa la clase `Punto`. El compilador y la JVM asumen que `Punto` pertenece también al package `Geometria`, y tal como está hecha la definición, para que la clase `Punto` sea accesible (conocida) por el compilador, es necesario que esté definida en el mismo package.

Si esto no es así, es necesario hacer accesible el espacio de nombres donde está definida la clase Punto a nuestra nueva clase. Esto se hace con la cláusula `import`. Supongamos que la clase Punto estuviera definida de esta forma:

```
package GeometriaBase;  
class Punto {  
    int x , y;  
}
```

Entonces, para usar la clase Punto en nuestra clase Circulo deberíamos poner:

```
package GeometriaAmpliada;  
  
import GeometriaBase.*;  
  
class Circulo {  
    Punto centro;  
    ... }  
}
```

Con la cláusula `import GeometriaBase.*;` se hacen accesibles todos los nombres (todas las clases) declaradas en el package GeometriaBase. Si sólo se quisiera tener accesible la clase Punto se podría declarar: `import GeometriaBase.Punto;`

También es posible hacer accesibles los nombres de un package sin usar la cláusula `import` calificando completamente los nombres de aquellas clases pertenecientes a otros packages. Por ejemplo:

```
package GeometriaAmpliada;  
  
class Circulo {  
    GeometriaBase.Punto centro;  
    ...  
}
```

Sin embargo si no se usa `import` es necesario especificar el nombre del package cada vez que se usa el nombre Punto.

La cláusula `import` simplemente indica al compilador donde debe buscar clases adicionales, cuando no pueda encontrarlas en el package actual y delimita los espacios de nombres y modificadores de acceso. Sin embargo, no tiene la implicación de 'importar' o copiar código fuente u objeto alguno. En una clase puede haber tantas sentencias `import` como sean necesarias. Las cláusulas `import` se colocan después de la cláusula `package` (si es que existe) y antes de las definiciones de las clases.

b. Nombres de los packages

Los packages se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las direcciones URL de Internet. Por ejemplo se puede tener un package de nombre `misPackages.Geometria.Base`. Cuando se utiliza esta estructura se habla de packages y subpackages. En el ejemplo `misPackages` es el Package base, `Geometria` es un subpackage de `misPackages` y `Base` es un subpackage de `Geometria`.

De esta forma se pueden tener los packages ordenados según una jerarquía equivalente a un sistema de archivos jerárquico.

El API de java está estructurado de esta forma, con un primer calificador (`java` o `javax`) que indica la base, un segundo calificador (`awt`, `util`, `swing`, etc.) que indica el grupo funcional de clases y opcionalmente subpackages en un tercer nivel, dependiendo de la amplitud del grupo. Cuando se crean packages de usuario no es recomendable usar nombres de packages que empiecen por `java` o `javax`.

c. Ubicación de packages en el sistema de archivos

Además del significado lógico descrito hasta ahora, los packages también tienen un significado físico que sirve para almacenar los módulos ejecutables (ficheros con extensión `.class`) en el sistema de archivos del ordenador.

Supongamos que definimos una clase de nombre `miClase` que pertenece a un package de nombre `misPackages.Geometria.Base`. Cuando la JVM vaya a cargar en memoria `miClase` buscará el módulo ejecutable (de nombre `miClase.class`) en un directorio en la ruta de acceso `misPackages/Geometria/Base`. Esta ruta deberá existir y estar accesible a la JVM para que encuentre las clases.

Si una clase no pertenece a ningún package (no existe cláusula **package**) se asume que pertenece a un package por defecto sin nombre, y la JVM buscará el archivo `.class` en el directorio actual.

Para que una clase pueda ser usada fuera del package donde se definió debe ser declarada con el modificador de acceso `public`, de la siguiente forma:

```
package GeometriaBase;
```

```
public class Punto {  
    int x , y;  
}
```

Si una clase no se declara **public** sólo puede ser usada por clases que pertenezcan al mismo package.

32. Compilación y ejecución de programas

En este apartado se asume que se ha instalado el JDK (J2SE) distribuido por SUN Microsystems y que tanto el compilador (javac) como la JVM (java) están accesibles. Asumiremos que los comandos se emitirán desde una ventana DOS en un sistema Windows, siendo la sintaxis en un entorno UNIX muy parecida. En este capítulo se verán todos los pasos necesarios para crear, compilar y ejecutar un programa Java.

PASO 1: Con un editor de texto simple (incluso notepad sirve, aunque resulta poco aconsejable) creamos un archivo con el contenido siguiente:

```
package Programas.Ejemplo1;

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

Guardamos el fichero fuente con nombre HolaMundo.java en la carpeta:
C:\ApuntesJava\Programas\Ejemplo1.

PASO 2: Abrimos una ventana DOS y en ella:

```
C:> cd C:\ApuntesJava
C:\ApuntesJava>javac Programas\Ejemplo1\HolaMundo.java
```

Si no hay ningún error en el programa se producirá la compilación y el compilador almacenará en el directorio C:\ApuntesJava\Programas\Ejemplo1 un fichero de

nombre `HolaMundo.class`, con el código ejecutable correspondiente a la clase `HolaMundo`.

Recuerda que en Java las mayúsculas y minúsculas son significativas. No es lo mismo la clase `ejemplo1` que la clase `Ejemplo1`. Esto suele ser fuente de errores, sobre todo al principio. Sin embargo, ten en cuenta que en algunos sistemas operativos como Windows, o más concretamente en una ventana DOS, esta distinción no existe. Puedes poner `cd C:\ApuntesJava` o `cd C:\APUNTESJAVA` indistintamente: el resultado será el mismo (no así en cualquier UNIX, que sí distingue unas y otras). Asegurate por tanto, de que las palabras están correctamente escritas.

Cuando pones `javac Programas\Ejemplo1\HolaMundo.java` estás indicando al compilador que busque un archivo de nombre `HolaMundo.java` en la ruta `Programas\Ejemplo1`, a partir del directorio actual; es decir, estás especificando la ruta de un archivo.

En el ejemplo se utiliza la clase del API de Java `System`. Sin embargo el programa no tiene ningún **import**. No obstante el compilador no detecta ningún error y genera el código ejecutable directamente. Esto se debe a que la clase `System` está definida en el package `java.lang`, que es el único del que no es necesario hacer el **import**, que es hecho implícitamente por el compilador. Para cualquier clase del API, definida fuera de este package es necesario hacer el `import` correspondiente.

PASO 3: Ejecutar el programa: Desde la ventana DOS.

```
C:\ApuntesJava>java Programas.Ejemplo1.HolaMundo
```

Se cargará la JVM, cargará la clase `HolaMundo` y llamará a su método `main` que producirá en la ventana DOS la salida:

```
Hola a todos
```

Los archivos `.class` son invocables directamente desde la línea de comandos (con la sintaxis `java nombreDeClase`) si tienen un método `main` definido.

Se puede indicar a la JVM que busque las clases en rutas alternativas al directorio actual. Esto se hace con el parámetro `-classpath` (abreviadamente `-cp`) en la línea de comandos. Por ejemplo si el directorio actual es otro, podemos invocar el programa de ejemplo de la forma:

```
C:\Windows>java -cp C:\ApuntesJava Programas.Ejemplo1.HolaMundo
```

Con el parámetro `-cp` se puede especificar diversas rutas alternativas para la búsqueda de clases separadas por ;

Cuando pones `java Programas.Ejemplo1.HolaMundo` estás indicando a la JVM que cargue y ejecute la clase `HolaMundo` del `Package Programas`, subpackage `Ejemplo1`. Para cumplir está orden, expresada en términos Java de clases y packages la JVM buscará el archivo `HolaMundo.class` en la ruta `Programas\Ejemplo1` que es algo expresado en términos del sistema de archivos, y por tanto del Sistema Operativo.

a. Archivos fuente (.java) y ejecutables (.class)

El esquema habitual es tener un archivo fuente por clase y asignar al archivo fuente el mismo nombre que la clase con la extensión `.java` (el nombre `.java` para la extensión es obligatorio). Esto generará al compilar un archivo `.class` con el mismo nombre que el fuente (y que la clase). Fuentes y módulos residirán en el mismo directorio.

Lo habitual es tener uno o varios packages que compartan un esquema jerárquico de directorios en función de nuestras necesidades (packages por aplicaciones, temas, etc.)

Es posible definir más de una clase en un archivo fuente, pero sólo una de ellas podrá ser declarada `public` (es decir podrá ser utilizada fuera del package donde se define). Todas las demás clases declaradas en el fuente serán internas al package. Si hay una clase `public` entonces, obligatoriamente, el nombre del fuente tiene que coincidir con el de la clase declarada como `public`

33. Modificadores

Los modificadores son elementos del lenguaje que se colocan delante de la definición de variables locales, datos miembro, métodos o clases y que alteran o condicionan el significado del elemento. En capítulos anteriores se ha descrito alguno, como es el modificador **static** que se usa para definir datos miembros o métodos como pertenecientes a una clase, en lugar de pertenecer a una instancia. En capítulos posteriores se tratarán otros modificadores como **final**, **abstract** o **synchronized**. En este capítulo se presentan los modificadores de acceso, que son aquellos que permiten limitar o generalizar el acceso a los componentes de una clase o a la clase en si misma.

a. Modificadores de acceso

Los modificadores de acceso permiten al diseñador de una clase determinar quien accede a los datos y métodos miembros de una clase.

Los modificadores de acceso preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

[modificadores] tipo_variable nombre;

[modificadores] tipo_devuelto nombre_Metodo (lista_Argumentos);

Existen los siguientes modificadores de acceso:

- **public** - Todo el mundo puede acceder al elemento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.
- **private** - Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede invocarse el método desde otro método de la clase.
- **protected** - Se explicará en el capítulo dedicado a la herencia.
- sin modificador - Se puede acceder al elemento desde cualquier clase del package donde se define la clase.

Pueden utilizarse estos modificadores para cualquier tipo de miembros de la clase, incluidos los constructores (con lo que se puede limitar quien puede crear instancias de la clase).

En el ejemplo los datos miembros de la clase Punto se declaran como private, y se incluyen métodos que devuelven las coordenadas del punto. De esta forma el diseñador de la clase controla el contenido de los datos que representan la clase e independiza la implementación de la interface.

```
class Punto {  
    private int x , y ;  
    static private int numPuntos = 0;  
  
    Punto ( int a , int b ) {  
        x = a ; y = b;  
        numPuntos ++ ;  
    }  
  
    int getX() {  
        return x;  
    }  
  
    int getY() {  
        return y;  
    }  
  
    static int cuantosPuntos() {  
        return numPuntos;  
    }  
}
```

```
}
```

Si alguien, desde una clase externa a Punto, intenta:

```
...  
Punto p = new Punto(0,0);  
p.x = 5;  
...
```

obtendrá un error del compilador.

b. Modificadores de acceso para clases

Las clases en si mismas pueden declararse:

- **public** - Todo el mundo puede usar la clase. Se pueden crear instancias de esa clase, siempre y cuando alguno de sus constructores sea accesible.
- sin modificador - La clase puede ser usada e instanciada por clases dentro del package donde se define.

Las clases no pueden declararse ni **protected** , ni **private** .

c. Importancia de los modificadores de acceso

Los modificadores de acceso permiten al diseñador de clases delimitar la frontera entre lo que es accesible para los usuarios de la clase, lo que es estrictamente privado y 'no importa' a nadie más que al diseñador de la clase e incluso lo que podría llegar a importar a otros diseñadores de clases que quisieran alterar, completar o especializar el comportamiento de la clase.

Con el uso de estos modificadores se consigue uno de los principios básicos de la Programación Orientada a Objetos, que es la encapsulación: Las clases tienen un comportamiento definido para quienes las usan conformado por los elementos que tienen un acceso público, y una implementación oculta formada por los elementos privados, de la que no tienen que preocuparse los usuarios de la clase.

Los otros dos modificadores, **protected** y el acceso por defecto (**package**) complementan a los otros dos. El primero es muy importante cuando se utilizan relaciones de herencia entre las clases y el segundo establece relaciones de 'confianza' entre clases afines dentro del mismo package. Así, la pertenencia de las clases a un mismo package es algo más que una clasificación de clases por cuestiones de orden.

Cuando se diseñan clases, es importante pararse a pensar en términos de quien debe tener acceso a que. Qué cosas son parte de la implantación y deberían ocultarse (y en que grado) y que cosas forman parte de la interface y deberían ser públicas.

34. Herencia

En anteriores ejemplos se ha visto que una clase tiene datos miembro que son instancias de otras clases. Por ejemplo:

```
class Circulo {  
    Punto centro;  
    int radio;  
    float superficie() {  
        return 3.14 * radio * radio;  
    }  
}
```

Esta técnica en la que una clase se compone o contiene instancias de otras clases se denomina composición. Es una técnica muy habitual cuando se diseñan clases. En el ejemplo diríamos que un Circulo tiene un Punto (centro) y un radio.

Pero además de esta técnica de composición es posible pensar en casos en los que una clase es una extensión de otra. Es decir una clase es como otra y además tiene algún tipo de característica propia que la distingue.

Por ejemplo podríamos pensar en la clase Empleado y definirla como:

```
class Empleado {  
    String nombre;  
    int numEmpleado , sueldo;  
  
    static private int contador = 0;  
  
    Empleado(String nombre, int sueldo) {  
        this.nombre = nombre;  
        this.sueldo = sueldo;  
        numEmpleado = ++contador;  
    }  
}
```

```

public void aumentarSueldo(int porcentaje) {
    sueldo += (int)(sueldo * aumento / 100);
}

public String toString() {
    return "Num. empleado " + numEmpleado + " Nombre: " + nombre +
        " Sueldo: " + sueldo;
}
}

```

En el ejemplo el Empleado se caracteriza por un nombre (String) y por un número de empleado y sueldo (enteros). La clase define un constructor que asigna los valores de nombre y sueldo y calcula el número de empleado a partir de un contador (variable estática que siempre irá aumentando), y dos métodos, uno para calcular el nuevo sueldo cuando se produce un aumento de sueldo (método `aumentarSueldo`) y un segundo que devuelve una representación de los datos del empleado en un String. (método `toString`).

Con esta representación podemos pensar en otra clase que reúna todas las características de Empleado y añada alguna propia. Por ejemplo, la clase Ejecutivo. A los objetos de esta clase se les podría aplicar todos los datos y métodos de la clase Empleado y añadir algunos, como por ejemplo el hecho de que un Ejecutivo tiene un presupuesto.

Así diríamos que la clase Ejecutivo extiende o hereda la clase Empleado. Esto en Java se hace con la cláusula **extends** que se incorpora en la definición de la clase, de la siguiente forma:

```

class Ejecutivo extends Empleado {
    int presupuesto;
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
}

```

Con esta definición un Ejecutivo es un Empleado que además tiene algún rasgo distintivo propio. El cuerpo de la clase Ejecutivo incorpora sólo los miembros que son específicos de esta clase, pero implícitamente tiene todo lo que tiene la clase Empleado.

A Empleado se le llama clase base o superclase y a Ejecutivo clase derivada o subclase.

Los objetos de las clases derivadas se crean igual que los de la clase base y pueden acceder tanto sus datos y métodos como a los de la clase base. Por ejemplo:

```
Ejecutivo jefe = new Ejecutivo( "Armando Mucho", 1000);  
jefe.asignarPresupuesto(1500);  
jefe.aumentarSueldo(5);
```

Nota: La discusión acerca de los constructores la veremos un poco más adelante.

Atención!: Un Ejecutivo ES un Empleado, pero lo contrario no es cierto. Si escribimos:

```
Empleado curri = new Empleado ( "Esteban Comex Plota" , 100) ;  
curri.asignarPresupuesto(5000); // error
```

se producirá un error de compilación pues en la clase Empleado no existe ningún método llamado asignarPresupuesto.

35. Definición de métodos. El uso de super.

Además se podría pensar en redefinir algunos métodos de la clase base pero haciendo que métodos con el mismo nombre y características se comporten de forma distinta. Por ejemplo podríamos pensar en rediseñar el método toString de la clase Empleado añadiendo las características propias de la clase Ejecutivo. Así se podría poner:

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
  
    public String toString() {  
        String s = super.toString();  
        s = s + " Presupuesto: " + presupuesto;  
        return s;  
    }  
}
```

```
}  
}
```

De esta forma cuando se invoque `jefe.toString()` se usará el método `toString` de la clase `Ejecutivo` en lugar del existente en la clase `Empleado`.

Observese en el ejemplo el uso de **super**, que representa referencia interna implícita a la clase base (superclase). Mediante **super.toString()** se invoca el método `toString` de la clase `Empleado`

a. Inicialización de clases derivadas

Cuando se crea un objeto de una clase derivada se crea implícitamente un objeto de la clase base que se inicializa con su constructor correspondiente. Si en la creación del objeto se usa el constructor no-args, entonces se produce una llamada implícita al constructor no-args para la clase base. Pero si se usan otros constructores es necesario invocarlos explícitamente.

En nuestro ejemplo dado que la clase `método` define un constructor, necesitaremos también un constructor para la clase `Ejecutivo`, que podemos completar así:

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
    Ejecutivo (String n, int s) {  
        super(n,s);  
    }  
  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
  
    public String toString() {  
        String s = super.toString();  
        s = s + " Presupuesto: " + presupuesto;  
        return s;  
    }  
}
```

Observese que el constructor de `Ejecutivo` invoca directamente al constructor de `Empleado` mediante `super(argumentos)`. En caso de resultar necesaria la

invocación al constructor de la superclase debe ser la primera sentencia del constructor de la subclase.

b. El modificador de acceso **protected**

El modificador de acceso **protected** es una combinación de los accesos que proporcionan los modificadores **public** y **private**. **protected** proporciona acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.

Por ejemplo, si en la clase **Empleado** definimos:

```
class Empleado {  
    protected int sueldo;  
    ...  
}
```

entonces desde la clase **Ejecutivo** se puede acceder al dato miembro **sueldo**, mientras que si se declara **private** no.

c. Up-casting y Down-casting

Siguiendo con el ejemplo de los apartados anteriores, dado que un **Ejecutivo** ES un **Empleado** se puede escribir la sentencia:

```
Empleado emp = new Ejecutivo("Máximo Dueño" , 2000);
```

Aquí se crea un objeto de la clase **Ejecutivo** que se asigna a una referencia de tipo **Empleado**. Esto es posible y no da error ni al compilar ni al ejecutar porque **Ejecutivo** es una clase derivada de **Empleado**. A esta operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases se denomina 'upcasting'.

Cuando se realiza este tipo de operaciones, hay que tener cuidado porque para la referencia **emp** no existen los miembros de la clase **Ejecutivo**, aunque la referencia apunte a un objeto de este tipo. Así, las expresiones:

```
emp.aumentarSueldo(3); // 1. ok. aumentarSueldo es de Empleado  
emp.asignarPresupuesto(1500); // 2. error de compilación
```


En la primera expresión no hay error porque el método `aumentarSueldo` está definido en la clase `Empleado`. En la segunda expresión se produce un error de compilación porque el método `asignarPresupuesto` no existe para la clase `Empleado`.

Por último, la situación para el método `toString` es algo más compleja. Si se invoca el método:

```
emp.toString(); // se invoca el metodo toString de Ejecutivo
```

El método que resultará llamado es el de la clase `Ejecutivo`. `toString` existe tanto para `Empleado` como para `Ejecutivo`, por lo que el compilador Java no determina en el momento de la compilación que método va a usarse. Sintácticamente la expresión es correcta. El compilador retrasa la decisión de invocar a un método o a otro al momento de la ejecución. Esta técnica se conoce con el nombre de *dynamic binding* o *late binding*. En el momento de la ejecución la JVM comprueba el contenido de la referencia `emp`. Si apunta a un objeto de la clase `Empleado` invocará al método `toString` de esta clase. Si apunta a un objeto `Ejecutivo` invocará por el contrario al método `toString` de `Ejecutivo`.

d. Operador cast

Si se desea acceder a los métodos de la clase derivada teniendo una referencia de una clase base, como en el ejemplo del apartado anterior hay que convertir explícitamente la referencia de un tipo a otro. Esto se hace con el operador de `cast` de la siguiente forma:

```
Empleado emp = new Ejecutivo("Máximo Dueño" , 2000);  
Ejecutivo ej = (Ejecutivo)emp; // se convierte la referencia de tipo  
ej.asignarPresupuesto(1500);
```

La expresión de la segunda línea convierte la referencia de tipo `Empleado` asignándola a una referencia de tipo `Ejecutivo`. Para el compilador es correcto porque `Ejecutivo` es una clase derivada de `Empleado`. En tiempo de ejecución la JVM convertirá la referencia si efectivamente `emp` apunta a un objeto de la clase `Ejecutivo`. Si se intenta:

```
Empleado emp = new Empleado("Javier Todudas" , 2000);  
Ejecutivo ej = (Ejecutivo)emp;
```

no dará problemas al compilar, pero al ejecutar se producirá un error porque la referencia emp apunta a un objeto de clase Empleado y no a uno de clase Ejecutivo.

36. La clase Object

En Java existe una clase base que es la raíz de la jerarquía y de la cual heredan todas aunque no se diga explícitamente mediante la cláusula extends. Esta clase base se llama Object y contiene algunos métodos básicos. La mayor parte de ellos no hacen nada pero pueden ser redefinidos por las clases derivadas para implementar comportamientos específicos.

Los métodos declarados por la clase Object son los siguientes:

- **public class** Object { ... }
- **public final** Class getClass() { ... }
- **public** String toString() { ... }
- **public boolean** equals(Object obj) { ... }
- **public int** hashCode() { ... }
- **protected** Object clone() **throws** CloneNotSupportedException { ... }
- **public final void** wait() **throws** IllegalMonitorStateException, InterruptedException { ... }
- **public final void** wait(**long** millis) **throws** IllegalMonitorStateException, InterruptedException { ... }
- **public final void** wait(**long** millis, **int** nanos) **throws** IllegalMonitorStateException, InterruptedException { ... }
- **public final void** notify() **throws** IllegalMonitorStateException { ... }
- **public final void** notifyAll() **throws** IllegalMonitorStateException { ... }
- **protected void** finalize() **throws** Throwable { ... }

Las cláusulas final y throws se verán más adelante. Como puede verse toString es un método de Object, que puede ser redefinido en las clases derivadas. Los métodos wait, notify y notifyAll tienen que ver con la gestión de threads de la JVM. El método finalize ya se ha comentado al hablar del recolector de basura.

Para una descripción exhaustiva de los métodos de Object se puede consultar la documentación de la API del JDK.

a. La cláusula final

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida. Para esto está la cláusula final, que tiene significados levemente distintos según se aplique a un dato miembro, a un método o a una clase.

Para una clase, final significa que la clase no puede extenderse. Es, por tanto el punto final de la cadena de clases derivadas. Por ejemplo si se quisiera impedir la extensión de la clase Ejecutivo, se pondría:

```
final class Ejecutivo {  
    ...  
}
```

Para un método, final significa que no puede redefinirse en una clase derivada. Por ejemplo si declaramos:

```
class Empleado {  
    ...  
    public final void aumentarSueldo(int porcentaje) {  
        ...  
    }  
    ...  
}
```

Entonces la clase Ejecutivo, clase derivada de Empleado no podría reescribir el método aumentarSueldo, y por tanto cambiar su comportamiento.

Para un dato miembro, final significa también que no puede ser redefinido en una clase derivada, como para los métodos, pero además significa que su valor no puede ser cambiado en ningún sitio; es decir el modificador final sirve también para definir valores constantes. Por ejemplo:

```
class Circulo {  
    ...  
    public final static float PI = 3.141592;  
    ...  
}
```

En el ejemplo se define el valor de PI como de tipo float, estático (es igual para todas las instancias), constante (modificador final) y de acceso público.

37. Herencia simple

Java incorpora un mecanismo de herencia simple. Es decir, una clase sólo puede tener una superclase directa de la cual hereda todos los datos y métodos. Puede existir una cadena de clases derivadas en que la clase A herede de B y B herede de C, pero no es posible escribir algo como:

```
class A extends B , C .... // error
```

Este mecanismo de herencia múltiple no existe en Java.

Java implanta otro mecanismo que resulta parecido al de herencia múltiple que es el de las interfaces que se verá más adelante.

38. Gestión de Excepciones

Las excepciones son el mecanismo por el cual pueden controlarse en un programa Java las condiciones de error que se producen. Estas condiciones de error pueden ser errores en la lógica del programa como un índice de un array fuera de su rango, una división por cero o errores disparados por los propios objetos que denuncian algún tipo de estado no previsto, o condición que no pueden manejar.

La idea general es que cuando un objeto encuentra una condición que no sabe manejar crea y dispara una excepción que deberá ser capturada por el que le llamó o por alguien más arriba en la pila de llamadas. Las excepciones son objetos que contienen información del error que se ha producido y que heredan de la clase Throwable o de la clase Exception. Si nadie captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar quién produjo la excepción.

Existen dos categorías de excepciones:

- Excepciones verificadas: El compilador obliga a verificarlas. Son todas las que son lanzadas explícitamente por objetos de usuario.
- Excepciones no verificadas: El compilador no obliga a su verificación. Son excepciones como divisiones por cero, excepciones de puntero nulo, o índices fuera de rango.

a. Generación de excepciones

Supongamos que tenemos una clase Empresa que tiene un array de objetos Empleado (clase vista en capítulos anteriores). En esta clase podríamos tener métodos para contratar un Empleado (añadir un nuevo objeto al array), despedirlo (quitarlo del array) u obtener el nombre a partir del número de empleado. La clase podría ser algo así como lo siguiente:

```
public class Empresa {
    String nombre;
    Empleado [] listaEmpleados;
    int totalEmpleados = 0;
    ...
    Empresa(String n, int maxEmp) {
        nombre = n;
        listaEmpleados = new Empleado [maxEmp];
    }
    ...
    void nuevoEmpleado(String nombre, int sueldo) {
        if (totalEmpleados < listaEmpleados.length ) {
            listaEmpleados[totalEmpleados++] = new Empleado(nombre,sueldo);
        }
    }
}
```

Observe en el método nuevoEmpleado que se comprueba que hay sitio en el array para almacenar la referencia al nuevo empleado. Si lo hay se crea el objeto. Pero si no lo hay el método no hace nada más. No da ninguna indicación de si la operación ha tenido éxito o no. Se podría hacer una modificación para que, por ejemplo el método devolviera un valor booleano true si la operación se ha completado con éxito y false si ha habido algún problema.

Otra posibilidad es generar una excepción verificada (Una excepción no verificada se produciría si no se comprobara si el nuevo empleado va a caber o no en el array). Vamos a ver como se haría esto.

Las excepciones son clases, que heredan de la clase genérica Exception. Es necesario por tanto asignar un nombre a nuestra excepción. Se suelen asignar nombres que den alguna idea del tipo de error que controlan. En nuestro ejemplo le vamos a llamar CapacidadEmpresaExcedida.

Para que un método lance una excepción:

- Debe declarar el tipo de excepción que lanza con la cláusula throws, en su declaración.
- Debe lanzar la excepción, en el punto del código adecuado con la sentencia throw.

En nuestro ejemplo:

```
void nuevoEmpleado(String nombre, int sueldo) throws
CapacidadEmpresaExcedida {
    if (totalEmpleados < listaEmpleados.length) {
        listaEmpleados[totalEmpleados++] = new Empleado(nombre,sueldo);
    }
    else throw new CapacidadEmpresaExcedida(nombre);
}
```

Además, necesitamos escribir la clase CapacidadEmpresaExcedida. Sería algo así:

```
public class CapacidadEmpresaExcedida extends Exception {
    CapacidadEmpresaExcedida(String nombre) {
        super("No es posible añadir el empleado " + nombre);
    }
    ...
}
```

La sentencia `throw` crea un objeto de la clase CapacidadEmpresaExcedida . El constructor tiene un argumento (el nombre del empleado). El constructor simplemente llama al constructor de la superclase pasándole como argumento un texto explicativo del error (y el nombre del empleado que no se ha podido añadir).

La clase de la excepción puede declarar otros métodos o guardar datos de depuración que se consideren oportunos. El único requisito es que extienda la clase Exception. Consultar la documentación del API para ver una descripción completa de la clase Exception.

De esta forma se pueden construir métodos que generen excepciones.

b. Captura de excepciones

Con la primera versión del método nuevoEmpleado (sin excepción) se invocaría este método de la siguiente forma:

```
Empresa em = new Empresa("La Mundial");
em.nuevoEmpleado("Adán Primero",500);
```

Si se utilizara este formato en el segundo caso (con excepción) el compilador produciría un error indicando que no se ha capturado la excepción verificada lanzada por el método nuevoEmpleado.

Para capturar la excepción se utiliza la construcción try / catch, de la siguiente forma:

```
Empresa em = new Empresa("La Mundial");  
try {  
    em.nuevoEmpleado("Adán Primero",500);  
} catch (CapacidadEmpresaExcedida exc) {  
    System.out.println(exc.toString());  
    System.exit(1);  
}
```

- Se encierra el código que puede lanzar la excepción en un bloque try / catch.
- A continuación del catch se indica que tipo de excepción se va a capturar.
- Después del catch se escribe el código que se ejecutará si se lanza la excepción.
- Si no se lanza la excepción el bloque catch no se ejecuta.

El formato general del bloque try / catch es:

```
try {  
    ...  
} catch (Clase_Excepcion nombre) { ... }  
    catch (Clase_Excepcion nombre) { ... }  
    ...
```

Observe que se puede capturar más de un tipo de excepción declarando más de una sentencia catch. También se puede capturar una excepción genérica (clase Exception) que engloba a todas las demás.

En ocasiones el código que llama a un método que dispara una excepción tampoco puede (o sabe) manejar esa excepción. Si no sabe que hacer con ella puede de nuevo lanzarla hacia arriba en la pila de llamada para que la gestione quien le llamo (que a su vez puede capturarla o reenviarla). Cuando un método no tiene intención de capturar la excepción debe declararla mediante la cláusula throws, tal como hemos visto en el método que genera la excepción.

Supongamos que, en nuestro ejemplo es el método main de una clase el que invoca el método nuevoEmpleado. Si no quiere capturar la excepción debe hacer lo siguiente:

```
public static void main(String [] args) throws CapacidadEmpresaExcedida {  
    Empresa em = new Empresa("La Mundial");
```

```
em.nuevoEmpleado("Adán Primero",500);  
}
```

c. Cláusula finally

La cláusula finally forma parte del bloque **try / catch** y sirve para especificar un bloque de código que se ejecutará tanto si se lanza la excepción como si no. Puede servir para limpieza del estado interno de los objetos afectados o para liberar recursos externos (descriptores de fichero, por ejemplo). La sintaxis global del bloque **try / catch / finally** es:

```
try {  
    ...  
} catch (Clase_Excepcion nombre) { ... }  
    catch (Clase_Excepcion nombre) { ... }  
    ...  
finally { ... }
```

39. Clases envoltorio (Wrapper)

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Por ejemplo, los contenedores definidos por el API en el package java.util (Arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) utilizan como unidad de almacenamiento la clase Object. Dado que Object es la raíz de toda la jerarquía de objetos en Java, estos contenedores pueden almacenar cualquier tipo de objetos. Pero los datos primitivos no son objetos, con lo que quedan en principio excluidos de estas posibilidades.

Para resolver esta situación el API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Por ejemplo podríamos definir una clase envoltorio para los enteros, de forma bastante sencilla, con:

```
public class Entero {  
    private int valor;  
  
    Entero(int valor) {  
        this.valor = valor;  
    }  
  
    int intValue() {  
        return valor;  
    }  
}
```



```
}  
}
```

La API de Java hace innecesario esta tarea al proporcionar un conjunto completo de clases envoltorio para todos los tipos primitivos. Adicionalmente a la funcionalidad básica que se muestra en el ejemplo las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc.)

Las clases envoltorio existentes son:

- Byte para byte.
- Short para short.
- Integer para int.
- Long para long.
- Boolean para boolean
- Float para float.
- Double para double y
- Character para char.

Observe que las clases envoltorio tienen siempre la primera letra en mayúsculas.

Las clases envoltura se usan como cualquier otra:

```
Integer i = new Integer(5);  
int x = i.intValue();
```

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación ...) están definidas solo para los datos primitivos por lo que las clases envoltura no sirven para este fin.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.

a. Resumen de métodos de Integer

Las clases envoltorio proporcionan también métodos de utilidad para la manipulación de datos primitivos. La siguiente tabla muestra un resumen de los métodos disponibles para la clase Integer

Método	Descripción
Integer(int valor) Integer(String valor)	Constructores a partir de int y String
int intValue() / byte byteValue() / float floatValue() . . .	Devuelve el valor en distintos formatos, int, long, float, etc.
boolean equals(Object obj)	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo.
static Integer getInteger(String s)	Devuelve un Integer a partir de una cadena de caracteres. Estático
static int parseInt(String s)	Devuelve un int a partir de un String. Estático.
static String toBinaryString(int i) static String toOctalString(int i) static String toHexString(int i) static String toString(int i)	Convierte un entero a su representación en String en binario, octal, hexadecimal, etc. Estáticos
String toString()	
static Integer valueOf(String s)	Devuelve un Integer a partir de un String. Estático.

El API de Java contiene una descripción completa de todas las clases envoltorio en el package java.lang.

40. Clases abstractas

Hay ocasiones, cuando se desarrolla una jerarquía de clases en que algún comportamiento está presente en todas ellas pero se materializa de forma distinta para cada una. Por ejemplo, pensemos en una estructura de clases para manipular figuras geométricas. Podríamos pensar en tener una clase genérica,

que podría llamarse `FiguraGeometrica` y una serie de clases que extienden a la anterior que podrían ser `Circulo`, `Poligono`, etc.

Podría haber un método `dibujar` dado que sobre todas las figuras puede llevarse a cabo esta acción, pero las operaciones concretas para llevarla a cabo dependen del tipo de figura en concreto (de su clase). Por otra parte la acción `dibujar` no tiene sentido para la clase genérica `FiguraGeometrica`, porque esta clase representa una abstracción del conjunto de figuras posibles.

Para resolver esta problemática Java proporciona las clases y métodos abstractos. Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código). Una clase abstracta es una clase que tiene al menos un método abstracto. Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

a. Declaración e implementación de métodos abstractos

Siguiendo con el ejemplo del apartado anterior, se puede escribir:

```
abstract class FiguraGeometrica {  
    ...  
    abstract void dibujar();  
    ...  
}  
  
class Circulo extends FiguraGeometrica {  
    ...  
    void dibujar() {  
        // codigo para dibujar Circulo  
        ...  
    }  
}
```

La clase abstracta se declara simplemente con el modificador **abstract** en su declaración. Los métodos abstractos se declaran también con el mismo modificador, declarando el método pero sin implementarlo (sin el bloque de código encerrado entre {}). La clase derivada se declara e implementa de forma normal, como cualquier otra. Sin embargo si no declara e implementa los métodos abstractos de la clase base (en el ejemplo el método `dibujar`) el compilador genera un error indicando que no se han implementado todos los

métodos abstractos y que, o bien, se implementan, o bien se declara la clase abstracta.

b. Referencias y objetos abstractos

Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner:

```
FiguraGeometrica figura;
```

Sin embargo una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta. El compilador producirá un error si se intenta:

```
FiguraGeometrica figura = new FiguraGeometrica();
```

Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

Sin embargo utilizando el up-casting visto en el capítulo dedicado a la Herencia si se puede escribir:

```
FiguraGeometrica figura = new Circulo(. . .);  
figura.dibujar();
```

La invocación al método dibujarse resolverá en tiempo de ejecución y la JVM llamará al método de la clase adecuada. En nuestro ejemplo se llamará al método dibujar de la clase Circulo.

41. Interfaces

El concepto de Interface lleva un paso más adelante la idea de las clases abstractas. En Java una interface es una clase abstracta pura, es decir una clase donde todos los métodos son abstractos (no se implementa ninguno). Permite al diseñador de clases establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código). Una interface puede también contener datos miembro, pero estos son siempre static y final. Una interface sirve para establecer un 'protocolo' entre clases.

Para crear una interface, se utiliza la palabra clave `interface` en lugar de `class`. La interface puede definirse `public` o sin modificador de acceso, y tiene el mismo significado que para las clases. Todos los métodos que declara una interface son siempre `public`.

Para indicar que una clase implementa los métodos de una interface se utiliza la palabra clave `implements`. El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interface. Una clase puede implementar más de una interface.

a. Declaración y uso

Una interface se declara:

```
interface nombre_interface {  
    tipo_retorno nombre_metodo ( lista_argumentos ) ;  
    ...  
}
```

Por ejemplo:

```
interface InstrumentoMusical {  
    void tocar();  
    void afinar();  
    String tipoInstrumento();  
}
```

Y una clase que implementa la interface:

```
class InstrumentoViento extends Object implements InstrumentoMusical {  
    void tocar() { ... };  
    void afinar() { ... };  
    String tipoInstrumento() {}  
}  
  
class Guitarra extends InstrumentoViento {  
    String tipoInstrumento() {  
        return "Guitarra";  
    }  
}
```

La clase `InstrumentoViento` implementa la interface, declarando los métodos y escribiendo el código correspondiente. Una clase derivada puede también redefinir si es necesario alguno de los métodos de la interface.

b. Referencias a Interfaces

Es posible crear referencias a interfaces, pero las interfaces no pueden ser instanciadas. Una referencia a una interface puede ser asignada a cualquier objeto que implemente la interface.

Por ejemplo:

```
InstrumentoMusical instrumento = new Guitarra();  
instrumento.play();  
System.out.println(instrumento.tipoInstrumento());
```

```
InstrumentoMusical i2 = new InstrumentoMusical(); //error.No se puede instanciar
```

c. Extensión de interfaces

Las interfaces pueden extender otras interfaces y, a diferencia de las clases, una interface puede extender más de una interface. La sintaxis es:

```
interface nombre_interface extends nombre_interface , ... {  
    tipo_retorno nombre_metodo ( lista_argumentos ) ;  
    ...  
}
```

d. Agrupaciones de constantes

Dado que, por definición, todos los datos miembros que se definen en una interface son static y final, y dado que las interfaces no pueden instanciarse resultan una buena herramienta para implantar grupos de constantes. Por ejemplo:

```
public interface Meses {  
    int ENERO = 1 , FEBRERO = 2 ... ;  
    String [] NOMBRES_MESES = { " " , "Enero" , "Febrero" , ... } ;  
}
```

Esto puede usarse simplemente:

```
System.out.println(Meses.NOMBRES_MESES[ENERO]);
```

e. Un ejemplo casi real

El ejemplo mostrado a continuación es una simplificación de como funciona realmente la gestión de eventos en el sistema gráfico de usuario soportado por el API de Java (AWT o swing). Se han cambiado los nombres y se ha simplificado para mostrar un caso real en que el uso de interfaces resuelve un problema concreto.

Supongamos que tenemos una clase que representa un botón de acción en un entorno gráfico de usuario (el típico botón de confirmación de una acción o de cancelación). Esta clase pertenecerá a una amplia jerarquía de clases y tendrá mecanismos complejos de definición y uso que no son objeto del ejemplo. Sin embargo podríamos pensar que la clase Boton tiene miembros como los siguientes.

```
class Boton extends ... {  
    protected int x , y, ancho, alto; // posicion del boton  
    protected String texto; // texto del boton  
    Boton(. . .) {  
        ...  
    }  
    void dibujar() { . . .}  
    public void asignarTexto(String t) { . . .}  
    public String obtenerTexto() { . . .}  
    ...  
}
```

Lo que aquí nos interesa es ver lo que sucede cuando el usuario, utilizando el ratón pulsa sobre el botón. Supongamos que la clase Boton tiene un método, de nombre por ejemplo click(), que es invocado por el gestor de ventanas cuando ha detectado que el usuario ha pulsado el botón del ratón sobre él.

El botón deberá realizar alguna acción como dibujarse en posición 'pulsado' (si tiene efectos de tres dimensiones) y además, probablemente, querrá informar a alguien de que se ha producido la acción del usuario. Es en este mecanismo de 'notificación' donde entra el concepto de interface. Para ello definimos una interface Oyente de la siguiente forma:

```
interface Oyente {  
    void botonPulsado(Boton b);  
}
```

La interface define un único método botonPulsado. La idea es que este método sea invocado por la clase Boton cuando el usuario pulse el botón. Para que esto sea posible en algún momento hay que notificar al Boton quien es el Oyente que debe ser notificado. La clase Boton quedaría:

```
class Boton extends ... {  
    ...  
    private Oyente oyente;  
    void registrarOyente(Oyente o) {  
        oyente = o;  
    }  
    void click() {  
        ...  
        oyente.botonPulsado(this);  
    }  
}
```

El método registrarOyente sirve para que alguien pueda 'apuntarse' como receptor de las acciones del usuario. Obsérvese que existe una referencia de tipo Oyente. A Boton no le importa que clase va a recibir su notificación. Simplemente le importa que implante la interface Oyente para poder invocar el método botonPulsado. En el método click se invoca este método. En el ejemplo se le pasa como parámetro una referencia al propio objeto Boton. En la realidad lo que se pasa es un objeto 'Evento' con información detallada de lo que ha ocurrido.

Con todo esto la clase que utiliza este mecanismo podría tener el siguiente aspecto:

```
class miAplicacion extends ... implements Oyente {  
    public static main(String [] args) {  
        new miAplicacion(. . .);  
        ...  
    }  
    ...  
    miAplicacion(. . .) {
```



```

    ...
    Boton b = new Boton(. . .);
    b.registrarOyente(this);
}

...
void botonPulsado(Boton x) {
    // procesar click
    ...
}
}

```

Obsérvese en el método registrarOyente que se pasa la referencia **this** que en el lado de la clase Boton es recogido como una referencia a la interface Oyente. Esto es posible porque la clase miAplicacion implementa la interface Oyente . En términos clásicos de herencia miAplicacion ES un Oyente .

42. Clases embebidas (Inner classes)

Una clase embebida es una clase que se define dentro de otra. Es una característica de Java que permite agrupar clases lógicamente relacionadas y controlar la 'visibilidad' de una clase. El uso de las clases embebidas no es obvio y contienen detalles algo más complejos que escapan del ámbito de esta introducción.

Se puede definir una clase embebida de la siguiente forma:

```

class Externa {
    ...
    class Interna {
        ...
    }
}

```

La clase Externa puede instanciar y usar la clase Interna como cualquier otra, sin limitación ni cambio en la sintaxis de acceso:

```

class Externa {
    ...
    class Interna {
        ...
    }
    void metodo() {
        Interna i = new Interna(. . .);
        ...
    }
}

```

Una diferencia importante es que un objeto de la clase embebida está relacionado siempre con un objeto de la clase que la envuelve, de tal forma que las instancias de la clase embebida deben ser creadas por una instancia de la clase que la envuelve. Desde el exterior estas referencias pueden manejarse, pero calificandolas completamente, es decir nombrando la clase externa y luego la interna.

Además una instancia de la clase embebida tiene acceso a todos los datos miembros de la clase que la envuelve sin usar ningún calificador de acceso especial (como si le pertenecieran). Todo esto se ve en el ejemplo siguiente.

a. Ejemplo

Un ejemplo donde puede apreciarse fácilmente el uso de clases embebidas es el concepto de iterador. Un iterador es una clase diseñada para recorrer y devolver ordenadamente los elementos de algún tipo de contenedor de objetos. En el ejemplo se hace una implementación muy elemental que trabaja sobre un array.

```
class Almacen {
    private Object [] listaObjetos;
    private numElementos = 0;
    Almacen (int maxElementos) {
        listaObjetos = new Object[maxElementos];
    }
    public Object get(int i) {
        return listaObject[i];
    }
    public void add(Object obj) {
        listaObjetos[numElementos++] = obj;
    }
    public Iterador getIterador() {
        new Iterador();
    }
}

class Iterador {
    int indice = 0;
    Object siguiente() {
        if (indice < numElementos) return listaObjetos[indice++];
        else return null;
    }
}
```

```

    }
}
}

```

Y la forma de usarse, sería:

```

Almacen alm = new Almacen(10); // se crea un nuevo almacen
...
alm.add(...); // se añaden objetos
...
// para recorrerlo
Almacen.Iterator i = alm.getIterator(); // obtengo un iterador para alm
Object o;
while ( (o = i.siguiente()) != null) {
    ...
}

```

43. Comentarios, documentación y convenciones de nombres

En Java existen comentarios de línea con // y bloques de comentario que comienzan con /* y terminan con */. Por ejemplo:

```

// Comentario de una línea
/* comienzo de comentario
   continua comentario
   fin de comentario */

```

a. Comentarios para documentación

El JDK proporciona una herramienta para generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. El nombre de la herramienta es javadoc. Para que javadoc pueda generar los textos HTML es necesario que se sigan unas normas de documentación en el fuente, que son las siguientes:

- Los comentarios de documentación deben empezar con /** y terminar con */.
- Se pueden incorporar comentarios de documentación a nivel de clase, a nivel de variable (dato miembro) y a nivel de método.
- Se genera la documentación para miembros public y protected.
- Se pueden usar tags para documentar ciertos aspectos concretos como listas de parámetros o valores devueltos. Los tags se indican a continuación.

Tipo de tag	Formato	Descripción
-------------	---------	-------------

Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.
Clases	@since	Fecha desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

Toda la documentación del API de Java está creada usando esta técnica y la herramienta javadoc.

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

```
/** Un programa Java simple.
 * Envía un saludo y dice que día es hoy.
 * @author Antonio Bel
 * @version 1
 */
public class HolaATodos {

    /** Unico punto de entrada.
     * @param args Array de Strings.
     * @return No devuelve ningun valor.
     * @throws No dispara ninguna excepcion.
     */
    public static void main(String [ ] args) {
        System.out.println("Hola a todos");
        System.out.println(new Date());
    }

}
```

b. Convenciones de nombres

SUN recomienda un estilo de codificación que es seguido en el API de Java y en estos apuntes que consiste en:

- Utilizar nombres descriptivos para las clases, evitando los nombres muy largos.
- Para los nombres de clases poner la primera letra en mayúsculas y las demás en minúsculas. Por ejemplo: Empleado
- Si el nombre tiene varias palabras ponerlas todas juntas (sin separar con - o _) y poner la primera letra de cada palabra en mayúsculas. Por ejemplo: InstrumentoMusical.
- Para los nombres de miembros (datos y métodos) seguir la misma norma, pero con la primera letra de la primera palabra en minúsculas. Por ejemplo: registrarOyente.
- Para las constantes (datos con el modificador final) usar nombres en mayúsculas, separando las palabras con _

Anexos

Ejemplo de como conectar java con el JDBC

```
public boolean Conectar(String DBName) {
    boolean val = true;
    //Creación de la URL
    String url = "jdbc:odbc:" + DBName;
    try {
        //Seleccionar y cargar el driver a ser usado.
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        //Conección con JDBC
        con = DriverManager.getConnection(url);
    } catch (SQLException jdbc) {
        //En caso de error con la conexión con JDBC-Server
        con = null;
        val = false;
        JOptionPane
            .showMessageDialog(
                new JFrame(),
```

```

        jdbc.toString()
            .substring(
                49,
                jdbc.toString()
                    .length())
            + "\n\nCausas probables del error:\n1- El usuario no
tiene los permisos para acceder a "
            + DBName
            + ".\nVerifique he inténtelo nuevamente.\nGracias!",
            "Error!.- No hubo conexión con la base de datos.",
            0);
    } catch (ClassNotFoundException cnfe) {
        //En caso de error con el driver.
        con = null;
        val = false;
        JOptionPane.showMessageDialog(new JFrame(), cnfe.toString()
            .substring(34, cnfe.toString().length()),
            "Error!.- Falta Driver.", 0);
    }
    return val; }

```

Applets

Las applets de Java son programas incrustados en otras aplicaciones, normalmente una página Web que se muestra en un navegador.

```

// Hola.java
import java.applet. Applet;
import java.awt. Graphics;

public class Hola extends Applet {
    public void paint(Graphics gc) {
        gc.drawString("Hola, mundo!", 65, 95);
    }
}

<!-- Hola.html -->
<html>
<head>
    <title>Applet Hola Mundo</title>
</head>
<body>
    <applet code="Hola" width="200" height="200">
    </applet>
</body>
</html>

```

La sentencia **import** indica al compilador de Java que incluya las clases **java.applet. Applet** y **java.awt. Graphics**, para poder referenciarlas por sus nombres, sin tener que anteponer la ruta completa cada vez que se quieran usar en el código fuente.

La clase **Hola** extiende (extends) a la clase **Applet**, es decir, es una subclase de ésta. La clase **Applet** permite a la aplicación mostrar y controlar el estado del applet. La clase **Applet** es un componente del AWT (Abstract Windowing Toolkit), que permite al applet mostrar una interfaz gráfica de usuario o GUI (Graphical User Interface), y responder a eventos generados por el usuario.

La clase **Hola** sobrecarga el método **paint(Graphics)** heredado de la superclase contenedora (**Applet** en este caso), para acceder al código encargado de dibujar. El método **paint()** recibe un objeto **Graphics** que contiene el contexto gráfico para dibujar el applet. El método **paint()** llama al método **drawString(String, int, int)** del objeto **Graphics** para mostrar la cadena de caracteres **Hola, mundo!** en la posición (65, 96) del espacio de dibujo asignado al applet.

La referencia al applet es colocada en un documento HTML usando la etiqueta **<applet>**. Esta etiqueta o tag tiene tres atributos: **code="Hola"** indica el nombre del applet, y **width="200" height="200"** establece la anchura y altura, respectivamente, del applet. Un applet también pueden alojarse dentro de un documento HTML usando los elementos **object**, o **embed**, aunque el soporte que ofrecen los navegadores Web no es uniforme.

Cómo crear un applet

Para crear un *applet* necesitamos escribir una clase de la clase *Applet* del paquete **java.applet.***;

```
import java.applet.*;
public class MiApplet extends Applet
{
    //Cuerpo del "applet".
}
```

El código anterior declara una nueva clase **MiApplet** que hereda todas las capacidades de la clase **Applet** de Java. El resultado es un fichero **MiApplet.java**. Una vez creada la clase que compone el *applet*, escribimos el resto del código y después lo compilamos, obteniendo el fichero **MiApplet.class**. Para poder crear el *applet* se necesita compilar el código Java en un intérprete.

```
import java.applet.*;
import java.awt.*;
import java.util.*;
import java.text.DateFormat;
```

```

public class MiApplet extends Applet implements Runnable
{
    private Thread hilo = null;
    private Font fuente;
    private String horaActual = "00:00:00";

    public void init()
    {
        fuente = new Font("Verdana", Font.BOLD, 24);
    }
    public void start()
    {
        if (hilo == null)
        {
            hilo = new Thread(this, "Reloj");
            hilo.start();
        }
    }
    public void run()
    {
        Thread hiloActual = Thread.currentThread();
        while (hilo == hiloActual)
        {
            //obtener la hora actual
            Calendar cal = Calendar.getInstance();

            Date hora = cal.getTime();
            DateFormat df = DateFormat.getTimeInstance();
            horaActual = df.format(hora);
            repaint();
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e){}
        }
    }
    public void paint(Graphics g)
    {
        //Dibujar un rectangulo alrededor del contenedor
        g.draw3DRect(1, 1, getSize().width-3, getSize().height-3, false);
        //Establecer la Fuente
        g.setFont(fuente);
        //mostrar la Hora
        g.drawString(horaActual,14,40);
    }
    public void stop()
    {
        hilo = null;
    }
}

```



```
}  
}
```

Finalmente, para ejecutar el *applet* hay que crear una página Web que haga referencia al mismo. La etiqueta HTML que permite hacer eso es <applet>:

```
<html>  
<applet code="MiApplet.class" width="370" height="270">  
</applet>  
</html>
```

Obviamente, los parámetros "MiApplet.class", "370" y "270" se pueden modificar.

Requerimientos mínimos de instalación:

- Sistema operativo Windows'95 o '98.
- 16 MB de memoria RAM.
- 50 MB disponibles en el disco duro.

Si se posee un computador con estas características se deben instalar los siguientes componentes:

- [el kit de desarrollo para Java 1.1](#): contiene los comandos javac y java para compilar y ejecutar sus programas. Para instalar el kit, ejecute el archivo. Se recomienda elegir como destino el directorio c:\jdk1.1.8.
- [La API de Java 1.1](#): contiene la documentación de las clases estándares de Java. Para instalar la API, desempaquete el archivo ".zip" con un programa como WinZip o ZipCentral. Le recomendamos elegir como destino c:\. Esto dejará la API junto con el kit de desarrollo.
- [FreeJava](#): es un editor de programas escritos en Java mucho más avanzado que el Notepad que viene con Windows. Desempaquete este archivo ".zip" en algún directorio temporal y luego ejecute el archivo setup.exe.
- [Biblioteca de CC10A](#): es una biblioteca especial para CC10A. En particular incluye la consola y otras clases. Copie este archivo a su PC sin desempaquetarlo.

Observaciones:

La instalación del kit de desarrollo requiere que Ud. agregue la siguiente línea en el archivo c:\autoexec.bat :

```
set PATH=C:\"... dir de jdk ..."bin;"%PATH%"
```

en donde "dir. de jdk" es el directorio en donde Ud. instaló el kit de desarrollo para Java 1.1.

Para que el compilador utilice la biblioteca de CC10A, se debe agregar la siguiente línea al archivo c:\autoexec.bat:

```
set CLASSPATH=;"...directorio de cc10a-lib.zip ..."cc10a-lib.zip
```

Esta biblioteca está en permanente desarrollo.

Por último, se recomienda agregar la siguiente línea al archivo c:\autoexec.bat:

```
set JAVA_COMPILER=NONE
```

Esto hará que Java consuma menos memoria y hará más fácil la la búsqueda de errores en sus programas. Para que las modificaciones hechas en c:\autoexec.bat tengan validez, se debe reiniciar el computador.

BIBLIOGRAFIA

<http://www.dcc.uchile.cl/~cc10a01b/2000/java/>

<http://www.desarrolloweb.com/manuales/57/>

http://www.java.com/es/download/windows_xpi.jsp?locale=es&host=www.java.com

<http://www.arrakis.es/~abelp/ApuntesJava/indice.htm>

http://www.fi-b.unam.mx/pp/profesores/carlos/java/java_basico1_5_2.html

¹ El **bytecode** es un código intermedio más abstracto que el código máquina. Habitualmente se lo ajtrata como a un fichero binario que contiene un programa ejecutable similar a un módulo objeto, que es un fichero binario que contiene código máquina producido por el compilador. El bytecode recibe su nombre porque generalmente cada código de operación tiene una longitud de un byte, si bien la longitud del código de las instrucciones varía. Cada instrucción tiene un código de operación entre 0 y 255 seguido de parámetros tales como los registros o las direcciones de memoria. Esta sería la descripción de un caso típico, si bien la especificación del bytecode depende ampliamente del lenguaje. Como código intermedio, se trata de una forma de salida utilizada por los implementadores de lenguajes para reducir la dependencia respecto del hardware específico y facilitar la interpretación. Menos frecuentemente se utiliza el bytecode como código intermedio en un compilador. Algunos sistemas, llamados traductores dinámicos o *compiladores just-in-time* traducen el bytecode a código máquina inmediatamente antes de su ejecución para mejorar la velocidad de ejecución.

² **Lenguaje interpretado**, es un lenguaje de programación que fue diseñado para ser ejecutado por medio de un intérprete, en contraste con los lenguajes compilados. También se les conoce como lenguajes de **script**.

³ La **GNU GPL** (*General Public License* o licencia pública general) es una licencia creada por la Free Software Foundation a mediados de los 80, y está orientada principalmente a proteger la libre distribución, modificación y uso de software. Su propósito es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios.

⁴ **Software libre** (en inglés *free software*) es el software que, una vez obtenido, puede ser usado, copiado, estudiado, modificado y redistribuido libremente. El software libre suele estar disponible gratuitamente, pero no hay que asociar software libre a software gratuito, o a precio del coste de la distribución a través de otros medios; sin embargo no es obligatorio que sea así y, aunque conserve su carácter de libre, puede ser vendido comercialmente.

⁵ **Objective-C** es un lenguaje de programación orientado a objetos creado como un superconjunto de C pero que implementase un modelo de objetos parecido al de Smalltalk. Originalmente fue creado por Brad Cox y la corporación StepStone en 1980. En 1988 fue adoptado como lenguaje de programación de NEXTSTEP y en 1992 fue liberado bajo licencia GNU para el compilador GCC. Actualmente se usa como lenguaje principal de programación en Mac OS X y GNUstep.

⁶ **Smalltalk** es un sistema informático que permite realizar tareas de computación mediante la interacción con un entorno de objetos virtuales. Metafóricamente, se puede considerar que un Smalltalk es un mundo virtual donde viven objetos que se comunican mediante el envío de mensajes.

⁷ **Eiffel** fue ideado en 1985 por Bertrand Meyer. Es un lenguaje de programación orientado a objetos centrado en la construcción de software robusto. Su sintaxis es parecida a la del lenguaje de programación Pascal. Una característica que lo distingue del resto de los lenguajes es que permite el diseño por contrato desde la base, con precondiciones, postcondiciones, invariantes y variantes de bucle, invariantes de clase y asertos. Eiffel es un lenguaje con tipos fuertes, pero relajado por herencia. Implementa administración automática de memoria, generalmente mediante algoritmos de recolección de basura. Las claves de este lenguaje están recogidas en el libro de Meyer, *Construcción de Software Orientado a Objetos*.

⁸ **C#** es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA e ISO. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET el cual es similar al de Java aunque incluye mejoras derivadas de otros lenguajes (más notablemente de Delphi y Java). C# fue diseñado para combinar el control a bajo nivel de lenguajes como C y la velocidad de programación de lenguajes como Visual Basic.

⁹ **J#** es un lenguaje transicional para programadores del lenguaje de programación Java y del lenguaje J++ de Microsoft, creado con la intención de que ambos puedan usar sus conocimientos actuales para crear aplicaciones en la plataforma .NET de Microsoft. J# se supone compatible con Java, tanto a nivel código fuente, como binario.

¹⁰ **JavaScript** es un lenguaje interpretado, es decir, que no requiere compilación, utilizado principalmente en páginas web, con una sintaxis semejante a la del lenguaje Java y el lenguaje C. Al contrario que Java, JavaScript no es un lenguaje orientado a objetos propiamente dicho, ya que no dispone de Herencia, es más bien un lenguaje basado en prototipos, ya que las nuevas clases se generan clonando las clases base (prototipos) y extendiendo su funcionalidad.

¹¹ Un **applet** es un componente de software que corre en el contexto de otro programa, por ejemplo un navegador web. El applet debe correr en un contenedor, que lo proporciona un programa anfitrión, mediante un plugin, o en aplicaciones como teléfonos móviles que soportan el modelo de programación por applets. A diferencia de un programa, un applet no puede correr de manera independiente, ofrece información gráfica y a veces interactúa con el usuario, típicamente carece de sesión y tiene privilegios de seguridad restringidos. Un applet normalmente lleva a cabo una función muy específica que carece de uso independiente. El término fue introducido en [AppleScript](#) en 1993.

Ejemplos comunes de *applets* son las Java applets y las animaciones Flash. Otro ejemplo es el Windows Media Player utilizado para desplegar archivos de video incrustados en los navegadores como el Internet Explorer. Otros plugins permiten mostrar modelos 3D que funcionan con una applet.

Un applet es un código JAVA que carece de un método main, por eso se utiliza principalmente para el trabajo de páginas web, ya que es un pequeño programa que es utilizado en una página HTML y representado por una pequeña pantalla gráfica dentro de ésta.

Por otra parte, la diferencia entre una aplicación JAVA y un applet radica en cómo se ejecutan. Para cargar una aplicación JAVA se utiliza el intérprete de JAVA (pcGRASP de Auburn University, Visual J++ de Microsoft, Forte de Sun de Visual Café). En cambio, un applet se puede cargar y ejecutar desde cualquier explorador que soporte JAVA (Netscape y Explorador de Windows).

¹² **Java Platform, Enterprise Edition o Java EE** (anteriormente conocido como Java 2 Platform, Enterprise Edition o J2EE hasta la versión 1.4), es una plataforma de programación—parte de la Plataforma Java—para desarrollar y ejecutar software de aplicaciones en Lenguaje de programación Java con arquitectura de n niveles distribuida, basándose ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones. La plataforma Java EE está definida por una *especificación*. Similar a otras especificaciones del Java Community Process, Java EE es también considerada informalmente como un estándar debido a que los suministradores deben cumplir ciertos requisitos de conformidad para declarar que sus productos son *conformes a Java EE*; no obstante sin un estándar de ISO o ECMA.

Java EE incluye varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc y define cómo coordinarlos. Java EE también configura algunas especificaciones únicas para Java EE para componentes. Estas incluyen Enterprise JavaBeans, servlets, portlets (siguiendo la especificación de Portlets Java), JavaServer Pages y varias tecnologías de servicios web. Esto permite al desarrollador crear una Aplicación de Empresa portable entre plataformas y escalable, a la vez que integrable con tecnologías anteriores. Otros beneficios añadidos son, por ejemplo, que el servidor de aplicaciones puede manejar transacciones, la seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de en tareas de mantenimiento de bajo nivel.

¹³ La plataforma **Java 2, Micro Edition, o Java ME** (anteriormente **J2ME**), es una colección de APIs en Java orientadas a productos de consumo como PDAs, teléfonos móviles o electrodomésticos. Java ME se ha convertido en una buena opción para crear juegos en teléfonos móviles debido a que se puede emular en un PC durante la fase de desarrollo y luego subirlos fácilmente al teléfono. Al utilizar tecnologías Java el desarrollo de aplicaciones o videojuegos con estas APIs resulta bastante económico de portar a otros dispositivos.

¹⁴ **HotJava** es un navegador web modular y extensible de Sun Microsystems que puede ejecutar applets Java. Fue el primer navegador en soportar applets Java y fue la plataforma de demostración de Sun para la entonces nueva tecnología. Desde entonces ha sido descontinuado y ya no es soportado. Como cualquier navegador de Web, HotJava puede decodificar [HTML](#) estándar y URLs estándares, aunque no soporta completamente el estándar HTML 3.0. La ventaja sobre el resto de navegadores, sin soporte Java, es que puede ejecutar programas Java sobre la red. La diferencia con Netscape, es que tiene implementado completamente los sistemas de seguridad que propone Java, esto significa que puede escribir y leer en el disco local, aunque esto hace disminuir la seguridad, ya que se pueden grabar en nuestro disco programas que contengan código malicioso e introducirnos un virus, por ejemplo. No obstante, el utilizar esta característica de HotJava es decisión del usuario.