1. INTRODUCCIÓN A JAVA

1.1 Origen de Java

Sun Microsystems, líder en servidores para Internet, uno de cuyos lemas desde hace mucho tiempo es "the network is the computer" (lo que quiere dar a entender que el verdadero ordenador es la red en su conjunto y no cada máquina individual), es quien ha desarrollado el lenguaje Java, en un intento de resolver simultáneamente todos los problemas que se le plantean a los desarrolladores de software por la proliferación de arquitecturas incompatibles, tanto entre las diferentes máquinas como entre los diversos sistemas operativos y sistemas de ventanas que funcionaban sobre una misma máquina, añadiendo la dificultad de crear aplicaciones distribuidas en una red como Internet.

He podido leer más de cinco versiones distintas sobre el origen, concepción y desarrollo de Java, desde la que dice que este fue un proyecto que rebotó durante mucho tiempo por distintos departamentos de Sun sin que nadie le prestara ninguna atención, hasta que finalmente encontró su nicho de mercado en la aldea global que es Internet; hasta la más difundida, que justifica a Java como lenguaje de pequeños electrodomésticos.

Hace algunos años, Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Tras unos comienzos dudosos, Sun decidió crear una filial, denominada FirstPerson Inc., para dar margen de maniobra al equipo responsable del proyecto.

El mercado inicialmente previsto para los programas de FirstPerson eran los equipos domésticos: microondas, tostadoras y, fundamentalmente, televisión interactiva. Este mercado, dada la falta de pericia de los usuarios para el manejo de estos dispositivos, requería unos interfaces mucho más cómodos e intuitivos que los sistemas de ventanas que proliferaban en el momento.

Otros requisitos importantes a tener en cuenta eran la fiabilidad del código y la facilidad de desarrollo. James Gosling, el miembro del equipo con más experiencia en lenguajes de programación, decidió que las ventajas aportadas por la eficiencia de C++ no compensaban el gran coste de pruebas y depuración. Gosling había estado trabajando en su tiempo libre en un lenguaje de programación que él había llamado Oak, el cual, aún partiendo de la sintaxis de C++, intentaba remediar las deficiencias que iba observando.

Los lenguajes al uso, como C o C++, deben ser compilados para un chip, y si se cambia el chip, todo el software debe compilarse de nuevo. Esto encarece mucho los desarrollos y el problema es especialmente acusado en el campo de la electrónica de consumo. La aparición de un chip más barato y, generalmente, más eficiente, conduce inmediatamente a los fabricantes a incluirlo en las nuevas series de sus cadenas de producción, por pequeña que sea la diferencia en precio ya que,
multiplicada por la tirada masiva de los aparatos, supone un ahorro considerable. Por tanto, Gosling decidió mejorar las características de Oak y utilizarlo.

El primer proyecto en que se aplicó este lenguaje recibió el nombre de proyecto Green y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar. Para ello se construyó un ordenador experimental denominado *7 (Star Seven). El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. En el sistema aparecía Duke, la actual mascota de Java.

Posteriormente se aplicó a otro proyecto denominado VOD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo y fueron como su bautismo de fuego.

Una vez que en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, urgieron a FirstPerson a desarrollar con rapidez nuevas estrategias que produjeran beneficios. No lo consiguieron y FirstPerson cerró en la primavera de 1994.

Pese a lo que parecía ya un olvido definitivo, Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podría llegar a ser el campo de juego adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes. Tras un cambio de nombre y modificaciones de diseño, el lenguaje Java fue presentado en sociedad en agosto de 1995.

Lo mejor será hacer caso omiso de las historias que pretenden dar carta de naturaleza a la clarividencia industrial de sus protagonistas; porque la cuestión es si independientemente de su origen y entorno comercial, Java ofrece soluciones a nuestras expectativas. Porque tampoco vamos a desechar la penicilina aunque haya sido su origen fruto de la casualidad.

1.2 Características de Java

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

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.

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.

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.

La verdad es que 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.

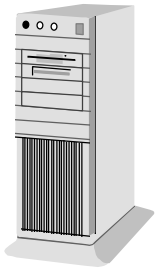
Robusto

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.

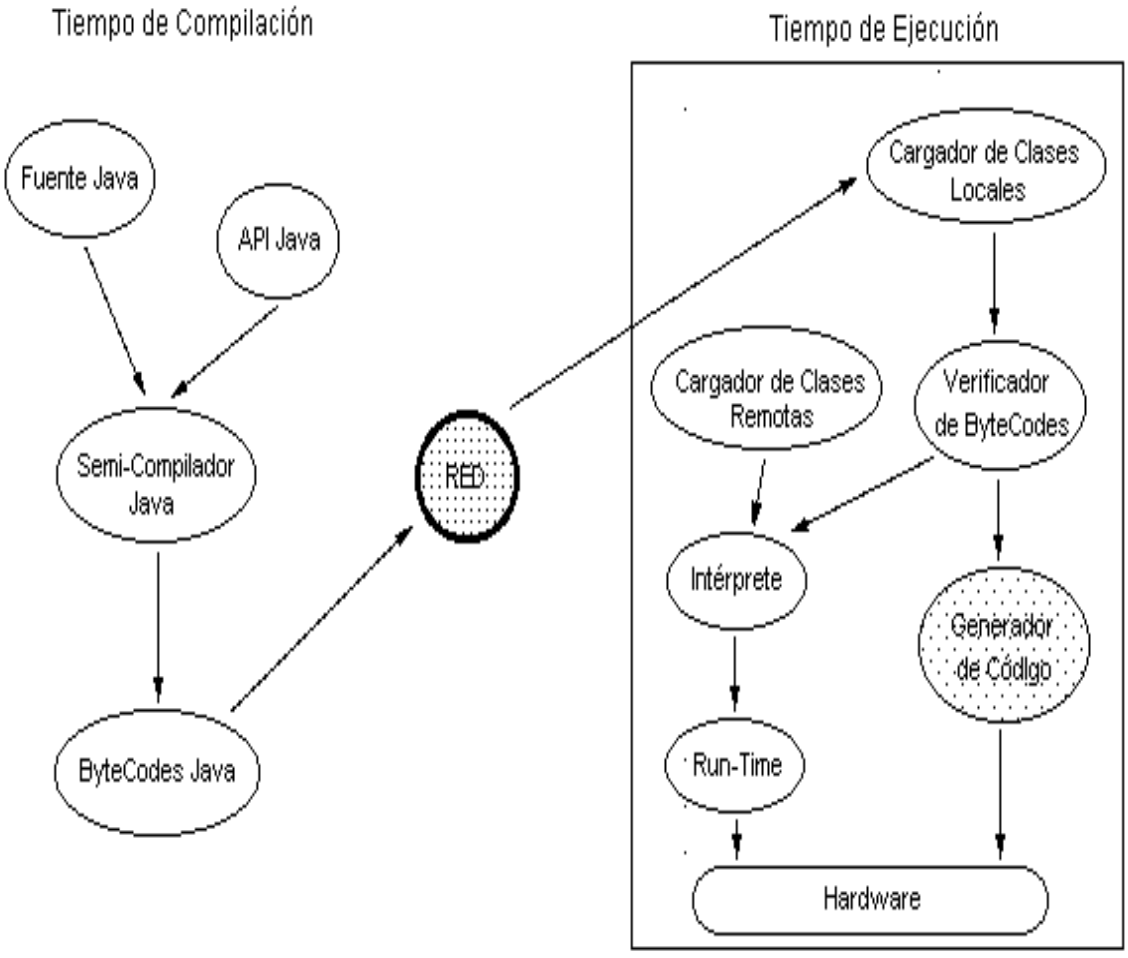
Java proporciona, pues:



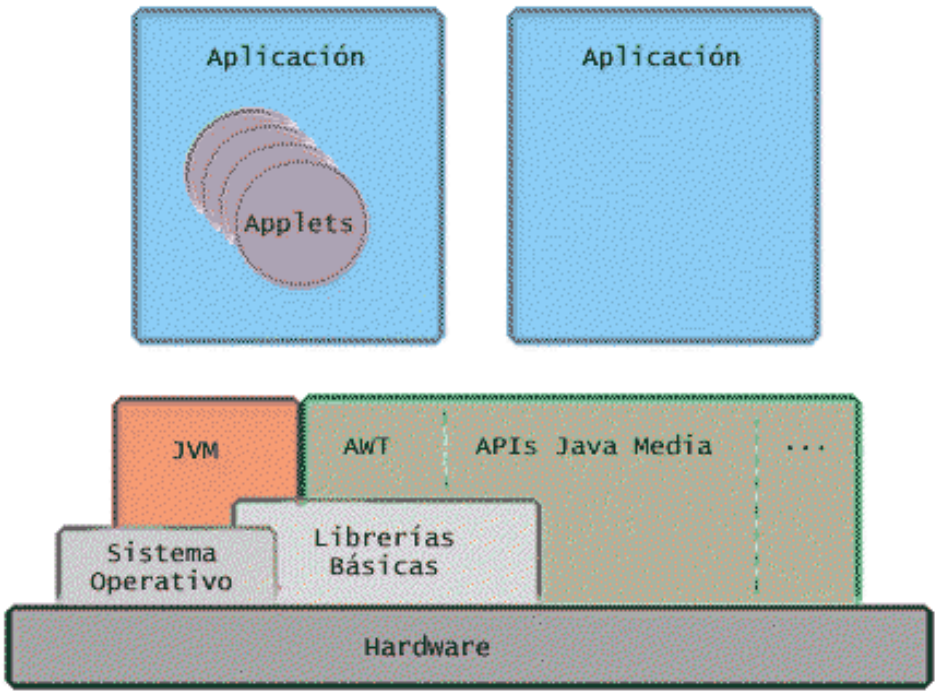
Comprobación de punteros
Comprobación de límites de arrays
Excepciones
Verificación de byte-codes

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.



El código fuente Java se "compila" a un código de bytes de alto nivel independiente de la máquina. Este código (byte-codes) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema run-time, que sí es dependiente de la máquina.
En una representación en que tuviésemos que indicar todos los elementos que forman parte de la arquitectura de Java sobre una plataforma genérica, obtendríamos una figura como la siguiente:



En ella podemos ver que lo verdaderamente dependiente del sistema es la Máquina Virtual Java (JVM) y las librerías fundamentales, que también nos permitirían acceder directamente al hardware de la máquina. Además, habrá APIs de Java que también entren en contacto directo con el hardware y serán dependientes de la máquina, como ejemplo de este tipo de APIs podemos citar:

Java 2D: gráficos 2D y manipulación de imágenes
Java Media Framework : Elementos críticos en el tiempo: audio, video...
Java Animation: Animación de objetos en 2D
Java Telephony: Integración con telefonía
Java Share: Interacción entre aplicaciones multiusuario
Java 3D: Gráficos 3D y su manipulación

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. Con un lenguaje como C, se pueden tomar números enteros aleatorios y convertirlos en punteros para luego acceder a la memoria:

```
printf( "Escribe un valor entero: " );
scanf( "%u",&puntero );
printf( "Cadena de memoria: %s\n",puntero );
```

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:

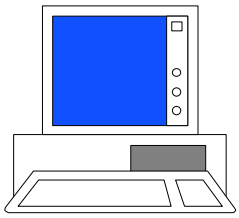
- El código no produce desbordamiento de operandos en la pila
- El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
- No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros en punteros.

- 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 autentificación 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 desemsamblador 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.

Portable

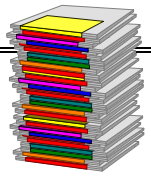
Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema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

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 propias 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.

Multithreaded

Al ser multithreaded (multihilvanado, en mala traducción), 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++.

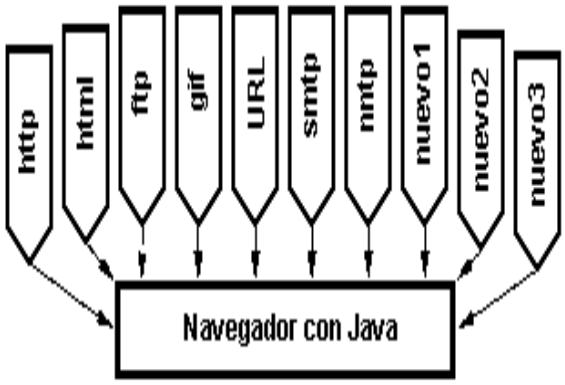
El beneficio de ser miltithreaded consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.), aún supera a los entornos de flujo único de programa (single-threaded)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

Dinamico

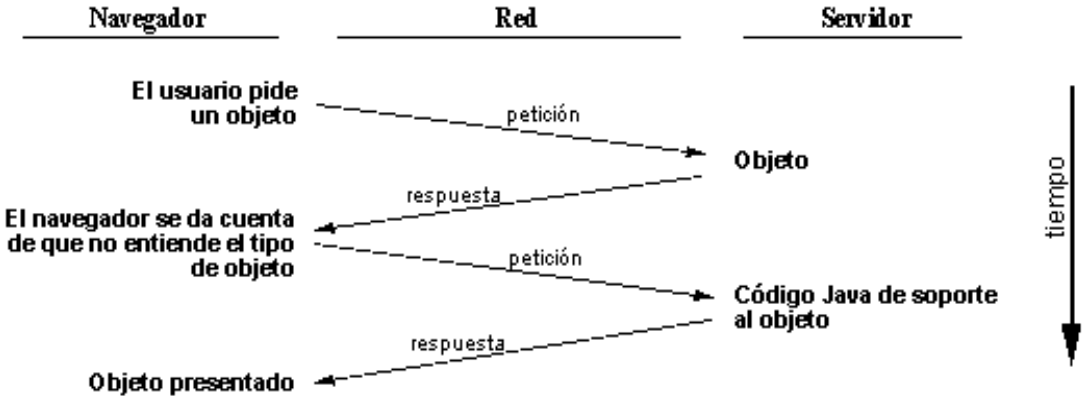
Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librería nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior).

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 de limpie la caché de la máquina.

¿Cuál es la ventaja de todo esto?¿Qué gano con Java?

- Primero: No debes 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).

- Segundo: Java es un lenguaje de programación orientado a objetos, y tiene todos los beneficios que ofrece esta metodología de programacion (más adelante doy una pequeña introducción a la filosofía de objetos).
- Tercero: 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.
- Cuarto: 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.
- Quinto: 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.

Todo esto suena muy bonito pero tambien se tienen algunas limitantes:

⇒ La velocidad.
⇒ Los programas hechos en Java no tienden a ser muy rápidos, supuestamente se está trabajando en 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. Muchas personas apuestan a futuro y piensan en Java. La pregunta es : ¿Estarán en lo correcto? La verdad es que no se, pero este manual no es para filosofar sobre el futuro del lenguaje sino para aprender a programarlo.

1.3 HotJava

HotJava, en pocas palabras, es un navegador con soporte Java (Java-enabled), desarrollado en Java. Como cualquier navegador de Web, HotJava puede decodificar HTML estándar y URLs estándares, aunque no soporta

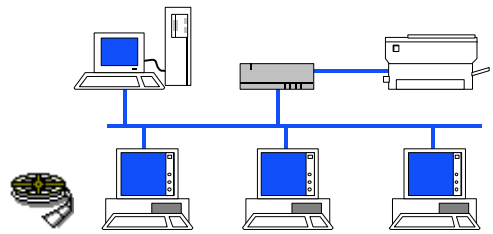
completamente el estándar HTML 3.0. La ventaja sobre el resto de navegadores, sin soporte Java, es que puede ejecutar programas Java sobre la red. La diferencia con Netscape, es que tiene implementado completamente los sistemas de seguridad que propone Java, esto significa que puede escribir y leer en 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.

1.4 Java para aplicaciones corporativas

Java actualmente está en boca de todos, Java e Intranet son las palabras de moda. Pero, surge la pregunta de si esta es una buena tecnología para desarrollar aplicaciones corporativas. Y la respuesta es afirmativa y voy a proponer argumentos para esa afirmación. En donde la red sea algo crítico, Java facilita tremendamente la vida de la programación corporativa.

Durante años, las grandes empresas se han convencido de que la "red" corporativa es la arteria por donde fluye la sangre que mantiene vivo su negocio. Desde el gran servidor de sus oficinas centrales, hasta los servidores de las delegaciones, las estaciones de trabajo de los programadores y la marabunta de PCs, la información va fluyendo de unos a otros. Para muchas compañías, la Red es la Empresa.

Si esta red no se mantiene sana, los pedidos no llegan, el inventario no se actualiza, el software no se desarrolla adecuadamente, los clientes no están satisfechos y, fundamentalmente, el dinero no entra. La necesidad de diagnosticar y reducir la arterioesclerosis de la red, hace que se estén inyectando continuamente nuevas metodologías que subsanen este grave problema.



¿Es Java la medicina? Está claro que cuando vemos un cepillo animado limpiando los dientes, cubos moviéndose en 3-D, o una banda de gatos locos en applets de Java, nos convencemos de que es el lenguaje idóneo para Internet. Pero, qué pasa con las aplicaciones corporativas, ¿sería una buena tecnología allí donde la red es el punto crítico? Vamos a intentar responder comparando las capacidades de Java contra la lista de necesidades de la red corporativa.

Desarrollo rápido de aplicaciones

Hace años, se decía que los programadores pronto desaparecerían. Los generadores automáticos de programas, eliminarían a los generadores humanos y el mundo sería un lugar mejor para vivir. Desafortunadamente, quienes decían

esto no tuvieron en cuenta una acelerada demanda de software de calidad para muy diferentes aplicaciones. Sin embargo, la tecnología de objetos pronto vino a intentar facilitar la tarea, adoptando el modelo de "generar parte de un programa", así, generando la parte básica de un programa (los objetos), se podría conectar con otras partes para proporcionar diferentes utilidades al usuario.

El lenguaje C++ es una buena herramienta, pero no cumple totalmente la premisa. Visual Basic y NextStep, se acercan cada vez más al poder de los objetos. Java facilita la creación de entornos de desarrollo-aplicaciones de modo similar, pero además es flexible, poderoso y efectivo. Los programadores ahora disponen de herramientas de programación de calidad beta, que apuntan hacia esa meta, como son el Java WorkShop de SunSoft, el entorno Java de Borland, el Café de Symantec, y pronto, herramientas más sofisticadas como Netcode o FutureTense. Esto proporciona una gran progresión a los entornos de desarrollo Java.

Aplicaciones efectivas y eficientes

Las aplicaciones que se crean en grandes empresas deben ser más efectivas que eficientes; es decir, conseguir que el programa funcione y el trabajo salga adelante es más importante que el que lo haga eficientemente. Esto no es una crítica, es una realidad de la programación corporativa. Al ser un lenguaje más simple que cualquiera de los que ahora están en el cajón de los programadores, Java permite a éstos concentrarse en la mecánica de la aplicación, en vez de pasarse horas y horas incorporando APIs para el control de las ventanas, controlando minuciosamente la memoria, sincronizando los ficheros de cabecera y corrigiendo los agónicos mensajes del linker. Java tiene su propio toolkit para interfaces, maneja por sí mismo la memoria que utilice la aplicación, no permite ficheros de cabecera separados (en aplicaciones puramente Java) y solamente usa enlace dinámico.
Muchas de las implementaciones de Java actuales son puros intérpretes. Los byte-codes son interpretados por el sistema run-time de Java, la Máquina Virtual Java (JVM), sobre el ordenador del usuario. Aunque ya hay ciertos proveedores que ofrecen compiladores nativos Just-In-Time (JIT). Si la Máquina Virtual Java dispone de un compilador instalado, las secciones (clases) del byte-code de la aplicación se compilarán hacia la arquitectura nativa del ordenador del usuario.

Los programas Java en ese momento rivalizarán con el rendimiento de programas en C++. Los compiladores JIT no se utilizan en la forma tradicional de un compilador; los programadores no compilan y distribuyen binarios Java a los usuarios. La compilación JIT tiene lugar a partir del byte-code Java, en el sistema del usuario, como una parte (opcional) del entorno run-time local de Java.

Muchas veces, los programadores corporativos, ansiosos por exprimir al máximo la eficiencia de su aplicación, empiezan a hacerlo demasiado pronto en el ciclo de vida de la aplicación. Java permite algunas técnicas innovadoras de optimización.

Por ejemplo, Java es inherentemente multithreaded, a la vez que ofrece posibilidades de multithread como la clase Thread y mecanismos muy sencillos de usar de sincronización; Java en sí utiliza threads. Los desarrolladores de compiladores inteligentes pueden utilizar esta característica de Java para lanzar un thread que compruebe la forma en que se está utilizando la aplicación. Más específicamente, este thread podría detectar qué métodos de una clase se están usando con más frecuencia e invocar a sucesivos niveles de optimización en tiempo de ejecución de la aplicación. Cuanto más tiempo esté corriendo la aplicación o el applet, los métodos estarán cada vez más optimizados (Guava de Softway es de este tipo).

Si un compilador JIT está embebido en el entorno run-time de Java, el programador no se preocupa de hacer que la aplicación se ejecute óptimamente. Siempre he pensado que en los Sistemas Operativos tendría que aplicarse esta filosofía; un optimizador progresivo es un paso más hacia esta idea.

Portabilidad para programador y programa

En una empresa de relativo tamaño hay una pléyade diferente de ordenadores. Probablemente nos encontremos con estaciones de trabajo Sun para el desarrollo de software, hordas de PCs para cada empleado, algún Mac en el departamento de documentación, una estación de trabajo HP en administración y una estación SGI en la sala de demos. Desarrollar aplicaciones corporativas para un grupo tan diferente de plataformas en excesivamente complejo y caro. Hasta ahora era complicado convencer a los programadores de cada arquitectura que utilizasen un API común para reducir el coste de las aplicaciones.

Con un entorno run-time de Java portado a cada una de las arquitecturas de las plataformas presentes en la empresa y una buena librería de clases ("packages" en Java), los programadores pueden entenderse y encontrar muy interesante trabajar con Java. Esta posibilidad hará tender a los programadores hacia Java, justo donde otros intentos anteriores con entornos universales (como Galaxy o XVT) han fracasado. Estos APIs eran simplemente inadecuados, no orientados a redes y, verdaderamente, pesados.

Una vez que los programas estén escritos en Java, otro lado interesante del asunto es que los programadores también son portables. El grupo de programadores de la empresa puede ahora enfrentarse a un desarrollo para cualquiera de las plataformas. La parte del cliente y del servidor de una aplicación estarán ahora escritas en el mismo lenguaje. Ya no será necesario tener un grupo

que desarrolle en Solaris en del departamento de I+D, programadores trabajando sobre Visual Basic en el departamento de documentación y programadores sobre GNU en proyectos especiales; ahora todos ellos podrán estar juntos y formar el grupo de software de la empresa.

Costes de desarrollo

En contraste con el alto coste de los desarrollos realizados sobre estaciones de trabajo, el coste de creación de una aplicación Java es similar al de desarrollar sobre un PC.

Desarrollar utilizando un software caro para una estación de trabajo (ahora barata) es un problema en muchas empresas. La eficiencia del hardware y el poco coste de mantenimiento de una estación de trabajo Sun, por ejemplo, resulta muy atractivo para las empresas; pero el coste adicional del entorno de desarrollo con C++ es prohibitivo para la gran mayoría de ellas. La llegada de Java e Intranet reducen considerablemente estos costes. Las herramientas Java ya no están en el entorno de precios de millones de pesetas, sino a los niveles confortables de precio de las herramientas de PCs. Y con el crecimiento cada día mayor de la comunidad de desarrolladores de software freeware y shareware que incluso proporcionan el código fuente, los programadores corporativos tienen un amplio campo donde moverse y muchas oportunidades de aprender y muchos recursos a su disposición.

El éxito que Internet ha proporcionado a los equipos de software corporativos es un regalo. El precio del software es ahora el mismo para un poderoso equipo corriendo Unix que para un PC. Incluso Netscape tiene al mismo precio la versión Unix de su servidor Web SuiteSpot que la versión PC/NT. Esta es la filosofía de precios que parece ser será la que se siga con las herramientas basadas en Java.

Mantenimiento y soporte

Un problema bien conocido que ocurre con el software corporativo es la demanda de cuidados y realimentación. Java no es, ciertamente, la cura para la enfermedad del mantenimiento, pero tiene varias características que harán la vida del enfermero más fácil.

Uno de los componentes del JDK es javadoc. Si se usan ciertas convenciones en el código fuente Java (como comenzar un comentario con /** y terminarlo con */), javadoc se puede fácilmente generar páginas HTML con el contenido de esos comentarios, que pueden visualizarse en cualquier navegador. La documentación del API de Java ha sido creada de este modo. Esto hace que el trabajo de documentar el código de nuevas clases Java sea trivial.

Otro gran problema del desarrollador corporativo es la creación y control de makefiles. Leerse un makefile es como estar leyendo la historia de empresa. Normalmente se pasan de programador a programador, quitando la información que no es esencial, siempre que se puede. Esto hace que muchos de los makefiles de las aplicaciones contengan docenas de librerías, una miríada de ficheros de cabecera y ultra-confusos macros. Es como mirar en el estómago de la ballena de Jonás.

Java reduce las dependencia de complejos makefiles drásticamente. Primero, no hay ficheros de cabecera. Java necesita que todo el código fuente de una clase se encuentre en un solo fichero. Java tiene la inteligencia de make en el propio lenguaje para simplificar la compilación de byte-codes.
Por ejemplo:

```
public class pepe {                  // Fichero: pepe.java
   Guitarra flamenca ;
   }
public class guitarra {              // Fichero: guitarra.java
   }
```

```
% javac -verbose pepe.java
[parsed pepe.java in 720ms]
[loaded C:\JAVA\BIN\..\classes\java\lang\Object.class in 220ms]
[checking class pepe]
[parsed .\\Guitarra.java in 50ms]
[wrote pepe.class]
[checking class Guitarra]
[wrote .\\Guitarra.class]
[done in 2300ms]
```

El compilador Java se da cuenta de que necesita compilar el fichero guitarra.java. Ahora vamos a forzarlo a que recompile pepe.java sin cambiar guitarra.java, podremos comprobar que el compilador de byte-code Java no recompila innecesariamente el fichero guitarra.java.

```
% javac -verbose pepe.java
[parsed pepe.java in 440ms]
[loaded C:\JAVA\BIN\..\classes\java\lang\Object.class in 160ms]
[checking class pepe]
[loaded .\\Guitarra.java in 0ms]
[wrote pepe.class]
[done in 1860ms]
```

Ahora, si modificamos guitarra.java (añadiendo, por ejemplo, otro miembro a la clase) y compilamos pepe.java, el compilador Java se dará cuenta de que debe recompilar tanto pepe.java como guitarra.java

```
% javac -verbose pepe.java
[parsed pepe.java in 710ms]
[loaded C:\JAVA\BIN\..\classes\java\lang\Object.class in 220ms]
[checking class pepe]
[parsed .\\Guitarra.java in 0ms]
[wrote pepe.class]
[checking class Guitarra]
[wrote .\\Guitarra.class]
[done in 2640ms]
```

En el libro Just Java de Peter van der Linden hay un capítulo excelente acerca del compilador de Java, si tienes oportunidad, no dejes de leerlo.

Aprendizaje

Si la empresa está llena de programadores de C++ con alguna experiencia en el manejo de librería gráficas, aprenderán rápidamente lo esencial de Java. Si el equipo de ingenieros no conoce C++, pero maneja cualquier otro lenguaje de programación orientada a objetos, les llevará pocas semanas dominar la base de Java. Lo que sí que no es cierto es que haya que aprender C++ antes de aprender Java.

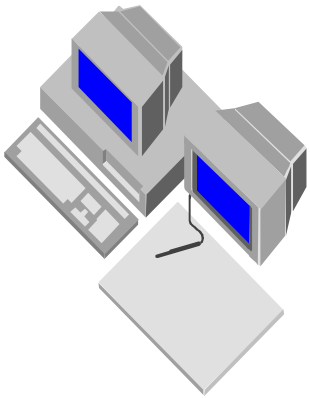
Si los ingenieros de la empresa no conocen ningún lenguaje orientado a objetos, sí que tienen que aprender los fundamentos de esta tecnología antes de nada, y luego aplicarlos a la programación con Java. El análisis y diseño orientado a objetos debe ser comprendido antes de intentar nada con Java. Los programadores de Java sin un fondo de conocimientos de OOA/D producirán código pobre. Además, los libros sobre Java crecen como la espuma, ya hay más de 25 publicados, y si buscas "Progamming in Java" en la Red, encontrarás 312 Web sites, y 30 más dedicados a "Learning Java". Y si esto, evidentemente, no es el sustituto de un instructor humano, hay ya varias empresas que ofrecen enseñanza de Java, entre ellas, Sun.

2. INSTALACIÓN DEL JDK

Actualmente ya hay entornos de desarrollo integrados completos para Java, diferentes del JDK de Sun. Symantec dispone de un compilador de Java para Windows 95 y Windows NT, con las ventajas del aumento de velocidad de proceso y capacidades multimedia que esto proporciona, Symantec Café. Borland también está trabajando en ello y la nueva versión de su entorno de desarrollo soporta Java. Sun ha lanzado la versión comercial de su propio entorno de desarrollo para Java, el Java Workshop, enteramente escrito en Java. Y Microsoft ha puesto en el mercado Visual J++, que sigue el estilo de todas sus herramientas de desarrollo.

No obstante, trataremos solamente el JDK, que hasta el momento es lo más conocido. El entorno básico del JDK de Java que proporciona Sun está formado por herramientas en modo texto, que son: java, intérprete que ejecuta programas en byte-code. javac, compilador de Java que convierte el código fuente en byte-code. javah, crea ficheros de cabecera para implementar métodos para cualquier clase. javap, es un descompilador de byte-code a código fuente Java. javadoc, es un generador automático de documentos HTML a partir del código fuente Java. javaprof, es un profiler para aplicaciones de un solo thread. HotJava, es un navegador Web escrito completamente en Java.

El entorno habitual pues, consiste en un navegador que pueda ejecutar applets, un compilador que convierta el código fuente Java a byte-code y el intérprete Java para ejecutar los programas. Estos son los componenetes básicos para desarrollar algo en Java. No obstante se necesita un editor para escribir el código fuente, y no son estrictamente necesarias otras herramientas como el debugger, un entorno visual, la documentación o un visualizador de jerarquía de clases. Tan es así, que disponiendo del navegador Netscape 2.0 no se necesita ni tan siquiera el JDK (a petición de varios amigos que disfrutan del uso de Linux pero no disponen de soporte ELF para poder utilizar el JDK portado por Randy Chapman, les indicaré como conseguir utilizar el compilador embebido en Netscape).



2.1 Windows

La versión del JDK para Windows es un archivo autoextraible. Se necesitan alrededor de 6 Mb de espacio libre en disco. Ejecutar el fichero, que desempaquetará el contenido del archivo. El directorio donde se instale no es importante, pero supondremos que se instala en el raiz del disco C:, en cuyo caso los archivos colgarán de c:\java. Es necesario añadir c:\java\bin a la variable de entorno PATH.

Además de los ficheros java, el JDK incluye dos librerías dinámicas, MSVCRT20.DLL y MFC30.DLL, que se instalarán en el directorio de Java. Si tienes ninguna copia de estos ficheros en tu ordenador (probablemente en el directorio system de Windows) copia estos ficheros en el directorio c:\java\bin. Si estos ficheros ya están en tu ordenador, elimina las copias extra que instala el JDK.

2.2 Solaris

La versión del JDK para Solaris es un fichero tar comprimido. Se necesitan alrededor de 9 Mb de disco para descomprimir el JDK, aunque el doble de espacio sería una cifra más cómoda. Ejecutar los siguientes comandos:

```
% uncompress JDK-beta-solaris2-sparc.tar.Z
% tar xvf JDK-beta-solaris2-sparc-tar
```

Puedes descomprimir el archivo en tu directorio home, o, si tienes privilegios de supervisor, en algún sitio más conveniente de /usr/local para que todos los usuarios tengan acceso a los ficheros. Sin embargo, los privilegios del supervisor no son necesarios para instalar y ejecutar Java. Por simplicidad, supondré que has descomprimido el JDK en /usr/local, aunque el path completo donde se haga no tiene relevancia (también es posible colocarlo en /opt que es donde residen todas las aplicaciones de Solaris). Si lo has colocado en un sitio diferente, simplemente sustituye /usr/local por ese directorio (si lo has descomprimido en tu home, puedes utilizar ~/java y ~/hotjava, en vez del path completo).

Es necesario añadir /usr/local/java/bin a la variable de entorno PATH. Utiliza el siguiente comando (suponiendo que tengas el shell csh o tcsh):

```
 set path=($PATH /usr/local/java/bin)
```

También puedes añadir esta línea al final del fichero .profile y .cshrc, y ya tienes el sistema listo para ejecutar applets. Si quieres desembarazarte de la ventana que aparece cada vez que lances el appletviewer con la licencia de Sun, crea un directorio que se llame .hotjava en el directorio java/bin y ya no volverás a verla.

2.3 Linux

Necesitas un kernel que soporte binarios ELF, por lo tanto tu Linux debe ser la versión 1.2.13 o superior, las anteriores tienen un bug que hacen que javac no funcione. Necesitas también Netscape, versión 2.0b4 o posterior. Sobre la versión 1.2.13 del kernel de Linux, hay que seguir los pasos que indico para conseguir que JDK funcione:

➥Bajarse el JDK, linux.jdk-1.0-try4.static-motif.tar.gz y

linux.jdk-1.0 try1.common.tar.gz a
/usr/local, descomprimirlo y hacer 'tar xvf'

➪En el fichero .java_wrapper (si no existe, crearlo) cambiar las variable J_HOME y PRG, para que queden como:

J_HOME=/usr/local/java
PRG=/usr/local/java/bin

➪Bajarse la librería libc.5.2.18.bin.tar.gz a /, descomprimirla, hacer 'tar xvf'.

➪Asegurarse de que /lib/libc.so.5 es un link simbólico a este nuevo fichero.Si no lo es, hacer el /lib 'ln -s libc.so.5.2.18 libc.so.5'

➪Bajarse ld-so.1.7.14.tar.gz a un directorio temporal, descomprimirlo y hacer 'tar xvf'.

➪Ejecutar 'instldso.sh' y eliminar el directorio temporal.

➪Añadir /usr/local/java a la variable de entorno PATH. Si se desea que esté fijada para todos los usuarios, incorporar el directorio a la varible PATH que se fija en el fichero /etc/profile.

➪Bajarse netscape-v202-export.i486-unknown-linux.tar.z a usr/local/netscape, descomprimirlo y hacer 'tar xvf'

➪Crear un link en /usr/local/bin a /usr/local/netscape/netscape

Esto debería ser suficiente para compilar cualquier cosa en Java/Linux. En caso de tener problemas, es el momento de recurrir a las FAQ.

Siguiendo los pasos indicados ya se puede ejecutar el ejemplo del Tic-Tac-Toe que propone la hoja de instalación que Sun ha incluido en todas sus versiones y que en Linux consistiría en cambiarse al directorio de la demo:

% cd /usr/local/java/demo/TicTacToe

ejecutar el visualizador de applets sobre la página html:

% appletviewer example1.html

y a jugar a las tres en raya. Por cierto, que el algoritmo que usa el ordenador está falseado por lo que es posible ganarle.

2.4 Compilación sin JDK

Parece raro, pero se puede conseguir. Lo único necesario es el navegador Netscape 2.0. Este navegador, junto con la máquina virtual Java (JVM) y el sistema run-time, tiene un compilador Java.
Si no se dispone del Java Development Kit (JDK), que no está disponible para todas las plataformas, pero sí de la versión de Netscape para nuestra plataforma, aquí van los pasos a seguir para utilizar el compilador de Java embebido en Netscape.

Como necesito partir de algún punto para tomarlo como referencia, voy a suponer que estamos sobre Linux y que vamos a prescindir del JDK de Randy Chapman. Lo que habría que hacer sería lo siguiente.

- Primero. Instalar Netscape en el ordenador. Asegurarse de entender perfectamente y leerse hasta el final el fichero README, para seguir las instrucciones específicas de la instalación de Netscape en la plataforma y que Netscape funcione perfectamente. En nuestro caso, en que vamos a intentar compilar código Java con Netscape sobre Linux, la pieza clave es la situación del fichero moz2_0.zip, que en mi máquina está en /usr/local/netscape/java/classes.
- Segundo. Extraer de una copia cualquiera del JDK (aunque sea de otra plataforma), el fichero java/lib/classes.zip y guardarlo en el mismo sitio que el fichero moz2_0.zip; esta localización no es necesaria, pero simplifica la estructura.

- Tercero. Fijar la variable de entorno CLASSPATH para que Netscape pueda encontrar sus propias clases además de las clases del Java de Sun. Asegurarse de incluir el "directorio actual", para poder compilar a la vez que se usan los ficheros .zip de Netscape y Sun. Por ejemplo:

  ```
  setenv CLASSPATH
  .:/usr/local/netscape/java/classes/moz2_0.zip :
  /usr/local/netscape/java/classes/classes.zip
  ```

- Cuarto. Compilar el código Java (applet o aplicación) con el comando: netscape -java sun.tools.javac.Main [fichero].java (sustituir el nombre del fichero con el código Java en vez de [fichero]). Esto convertirá el código fuente Java en byte-code, generándose el archivo [fichero].class.

- Quinto. Comprobar si se puede ejecutar la aplicación con el comando:
  netscape -java [clase]
  (sustituir el nombre de la clase de la aplicación -la que contiene la rutina main- en vez de [clase]).

↗ Sexto. Si se ha compilado un applet Java, construir una página html que lo utilice para visualizarlo con el navegador en su forma normal. O también se puede visualizar utilizando el appletviewer, ejecutando:

netscape -java sun.applet.AppletViewer [clase]

Desgraciadamente, la sentencia anterior no parece funcionar en todos los sistemas. Hay amigos míos que no han sido capaces de visualizar applets con este método.

Para aprovechar el tiempo, se puede crear un script que recoja los pasos 3, 4 y 6. Si estamos utilizando el csh, el contenido del script sería:

```
#/bin/csh -f setenv CLASSPATH
   .:/usr/local/netscape/java/classes/moz2_0.zip:
   /usr/local/netscape/java/classes/classes.zip
   netscape -java sun.tools.javac.Main $1
```

y lo almacenaríamos como javac. Se ha de hacer el script ejecutable y cambiar /bin/csh por el path completo donde esté situado el csh. De forma semejante podemos definir el intérprete java y el appletviewer, sustituyendo la línea adecuada de llamada a Netscape.

3. CONCEPTOS BÁSICOS DE JAVA

3.1 Programación en Java

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

Comentarios

En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea

/* comentarios de una o
   más líneas
*/

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

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

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

Identificadores

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

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

Serían identificadores válidos:

identificador
nombre_usuario
Nombre_Usuario
_variable_del_sistema
$transaccion

y su uso sería, por ejemplo:

```
int contador_principal;
char _lista_de_ficheros;
float $cantidad_en_Ptas;
```

Palabras clave

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como indentificadores:

abstract	continue	for	new	switch	
boolean	default	goto	null	synchronized	
break	do	if	package	this	
byte	double	implements	private	threadsafe	
byvalue	else	import	protected	throw	
case	extends	instanceof	public	transient	
catch	false	int	return	true	
char	final	interface	short	try	
class	finally	long	static	void	

const float native super while

Palabras Reservadas
Además, el lenguaje se reserva unas cuantas palabras más, pero que hasta ahora no tienen un cometido específico. Son:

cast future generic inner
operator outer rest var

Literales
Un valor constante en Java se crea utilizando una representación literal de él. Java utiliza cinco tipos de elementos: enteros, reales en coma flotante, booleanos, caracteres y cadenas, que se pueden poner en cualquier lugar del código fuente de Java. Cada uno de estos literales tiene un tipo correspondiente asociado con él.

Enteros:

byte	8 bits	complemento a dos
short	16 bits	complemento a dos
int	32 bits	complemento a dos
long	64 bits	complemento a dos

Por ejemplo: 21 077 0xDC00

Reales en coma flotante:

float	32 bits	IEEE 754
double	64 bits	IEEE 754

Por ejemplo: 3.14 2e12 3.1E12

Booleanos:
true
false

Caracteres:
Por ejemplo: a \t \u???? [????] es un número unicode
Cadenas:
Por ejemplo: "Esto es una cadena literal"

Arrays
Se pueden declarar en Java arrays de cualquier tipo:

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

Incluso se pueden construir arrays de arrays:

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

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

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

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

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

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

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

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

Esto que es equivalente a:

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

No se pueden crear arrays estáticos en tiempo de compilación:

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

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

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

Es decir, todos los arrays en Java son estáticos. Para convertir un array en el equivalente a un array dinámico en C/C++, se usa la clase vector, que permite operaciones de inserción, borrado, etc. en el array.

Operadores

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

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

,

Los operadores numéricos se comportan como esperamos:

int + int = int

Los operadores relacionales devuelven un valor booleano.

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

String nombre = "nombre" + "Apellido";

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

Separadores

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

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

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

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

; - punto y coma. Separa sentencias.

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

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

3.2 Control de Flujo

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

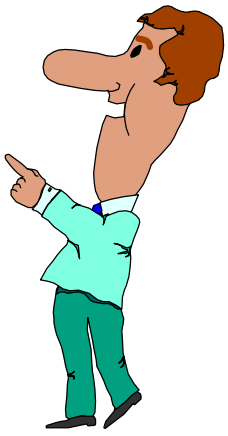
Sentencias de Salto

if/else

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

switch

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



Sentencias de Bucle

Bucles for

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

El siguiente trocito de código Java que dibuja varias líneas en pantalla alternando sus colores entre rojo, azul y verde. Este fragmento sería parte de una función Java (método):

```
int contador;
for( contador=1; contador <= 12; contador++ ) {
   switch( contador % 3 ) {
      case 0:
         setColor( Color.red );
         break;
      case 1:
         setColor( Color.blue );
         break;
      case 2:
         setColor( Color.green );
         break;
      }
   g.drawLine( 10,contador*10,80,contador*10 );
   }
```

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

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

Bucles while

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

Bucles do/while

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

Excepciones

try-catch-throw

```
try {
   sentencias;
} catch( Exception ) {
   sentencias;
}
```

Java implementa excepciones para facilitar la construcción de código robusto. Cuando ocurre un error en un programa, el código que encuentra el error lanza una excepción, que se puede capturar y recuperarse de ella. Java proporciona muchas excepciones predefinidas.

Control General del Flujo

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

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

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

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

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

```
}
```

3.3 Clases

Las clases son lo más simple de Java. Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas Java.

Todas las acciones de los programas Java se colocan dentro del bloque de una clase o un objeto. Todos los métodos se definen dentro del bloque de la clase, Java no soporta funciones o variables globales. Esto puede despistar a los programadores de C++, que pueden definir métodos fuera del bloque de la clase, pero esta posibilidad es más un intento de no separarse mucho y ser compatible con C, que un buen diseño orientado a objetos. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos. En C la unidad fundamental son los ficheros con código fuente, en Java son las clases. De hecho son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave import (equivalente al #include) puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

Tipos de Clases

Hasta ahora sólo se ha utilizado la palabra clave public para calificar el nombre de las clases que hemos visto, pero hay tres modificadores más. Los tipos de clases que podemos definir son:

abstract

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

final

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

public

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

synchronizable

Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintos threads; el sistema se encarga de colocar los flags necesarios para

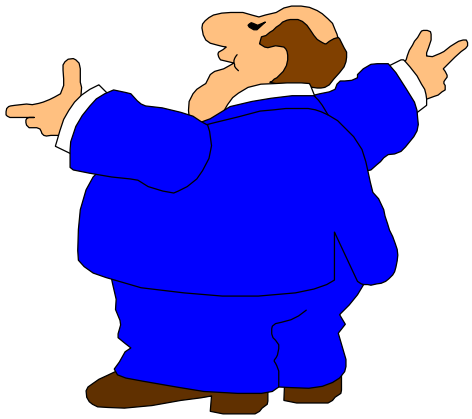
evitarlo. Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobreescriban.

3.4 Variables y Métodos de Instancia

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

Por ejemplo, en el siguiente trozo de código podemos observarlo:

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



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

Ambito de una variable

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

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

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

Métodos y Constructores

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

Cuando se declara una clase en Java, se pueden declarar uno o más constructores opcionales que realizan la inicialización cuando se instancia (se crea una ocurrencia) un objeto de dicha clase.
Utilizando el código de ejemplo anterior, cuando se crea una nueva instancia de MiClase, se crean (instancian) todos los métodos y variables, y se llama al constructor de la clase:

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

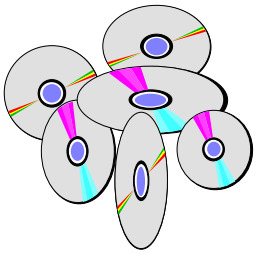
La palabra clave new se usa para crear una instancia de la clase. Antes de ser instanciada con new no consume memoria, simplemente es una declaración de tipo. Después de ser instanciado un nuevo objeto mc, el valor de i en el objeto mc será igual a 10. Se puede referenciar la variable (de instancia)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

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

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

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

y ahora la variable mc.i vale 21.



Finalizadores

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

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

3.5 Alcance de Objetos y Reciclado de Memoria

Los objetos tienen un tiempo de vida y consumen recursos durante el mismo. Cuando un objeto no se va a utilizar más, debería liberar el espacio que ocupaba

en la memoria de forma que las aplicaciones no la agoten (especialmente las grandes).
En Java, la recolección y liberación de memoria es responsabilidad de un thread llamado automatic garbage collector (recolector automático de basura). Este thread monitoriza el alcance de los objetos y marca los objetos que se han salido de alcance. Veamos un ejemplo:

```
String s;               // no se ha asignado todavia
s = new String( "abc" );  // memoria asignada
s = "def";              // se ha asignado nueva memoria
                    // (nuevo objeto)
```

Más adelante veremos en detalle la clase String, pero una breve descripción de lo que hace esto es; crear un objeto String y rellenarlo con los caracteres "abc" y crear otro (nuevo) String y colocarle los caracteres "def".

En esencia se crean dos objetos:

Objeto String "abc"
Objeto String "def"

Al final de la tercera sentencia, el primer objeto creado de nombre s que contiene "abc" se ha salido de alcance. No hay forma de acceder a él. Ahora se tiene un nuevo objeto llamado s y contiene "def". Es marcado y eliminado en la siguiente iteración del thread reciclador de memoria.

3.6 Herencia

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

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

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

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

```
import MiClase;
public class MiNuevaClase extends MiClase {
```

```
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        }
    }
```

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

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

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

3.7 Control de Acceso

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

public
    public void CualquieraPuedeAcceder(){}

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

protected
    protected void SoloSubClases(){}

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

private
    private String NumeroDelCarnetDeIdentidad;

Las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases.
friendly (sin declaración específica)

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

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

Los métodos protegidos (protected) pueden ser vistos por las clases derivadas, como en C++, y también, en Java, por los paquetes (packages). Todas las clases de un paquete pueden ver los métodos protegidos de ese paquete. Para evitarlo, se deben declarar como private protected, lo que hace que ya funcione como en C++ en donde sólo se puede acceder a las variables y métodos protegidos de las clases derivadas.

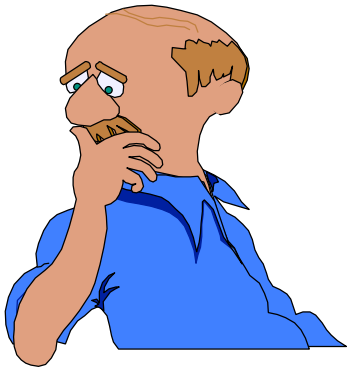
3.8 Variables y Métodos Estáticos

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

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

El valor de la variable version será el mismo para cualquier objeto instanciado de la clase Documento. Siempre que un objeto instanciado de Documento cambie la variable version, ésta cambiará para todos los objetos.
De la misma forma se puede declarar un método como estático, lo que evita que el método pueda acceder a las variables de instancia no estáticas:

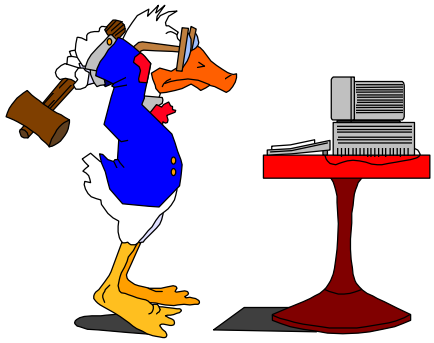
```
class Documento extends Pagina {
   static int version = 10;
   int numero_de_capitulos;
   static void annade_un_capitulo() {
      numero_de_capitulos++;          // esto no funciona
      }
   static void modifica_version( int i ) {
      version++;                      // esto si funciona
  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



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

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

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



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

3.9 this Y super

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

```
public class MiClase {
   int i;
   public MiClase() {
      i = 10;
      }
```

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

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

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

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

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

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

3.10 Clases Abstractas

Una de las características más útiles de cualquier lenguaje orientado a objetos es la posibilidad de declarar clases que definen como se utiliza solamente, sin tener que implementar métodos. Esto es muy útil cuando la implementación es específica para cada usuario, pero todos los usuarios tienen que utilizar los mismos métodos. Un ejemplo de clase abstracta en Java es la clase Graphics:

```
public abstract class Graphics {
public abstract void drawLine( int x1,int y1,int x2,
   int y2 );
public abstract void drawOval( int x,int y,int width,
   int height );
```

```
public abstract void drawArc( int x,int y,int width,
    int height,int startAngle,int arcAngle );
    . . .
    }
```

Los métodos se declaran en la clase Graphics, pero el código que ejecutará el método está en algún otro sitio:

```
public class MiClase extends Graphics {
  public void drawLine( int x1,int y1,int x2,int y2 ) {
    <código para pintar líneas -específico de
    la arquitectura->
    }
  }
```

Cuando una clase contiene un método abstracto tiene que declararse abstracta. No obstante, no todos los métodos de una clase abstracta tienen que ser abstractos. Las clases abstractas no pueden tener métodos privados (no se podrían implementar) ni tampoco estáticos. Una clase abstracta tiene que derivarse obligatoriamente, no se puede hacer un new de una clase abstracta.

Una clase abstracta en Java es lo mismo que en C++ virtual func() = 0; lo que obliga a que al derivar de la clase haya que implementar forzosamente los métodos de esa clase abstracta.

3.11 Interfaces

Los métodos abstractos son útiles cuando se quiere que cada implementación de la clase parezca y funcione igual, pero necesita que se cree una nueva clase para utilizar los métodos abstractos.

Los interfaces proporcionan un mecanismo para abstraer los métodos a un nivel superior.

Un interface contiene una colección de métodos que se implementan en otro lugar. Los métodos de una clase son public, static y final.
La principal diferencia entre interface y abstract es que un interface proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Por ejemplo:

```
public interface VideoClip {
    // comienza la reproduccion del video
    void play();
```

```
// reproduce el clip en un bucle
void bucle();
// detiene la reproduccion
void stop();
}
```

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

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

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

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

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

3.12 Métodos Nativos

Java proporciona un mecanismo para la llamada a funciones C y C++ desde nuestro código fuente Java. Para definir métodos como funciones C o C++ se utiliza la palabra clave native.

```
public class Fecha {
    int ahora;
    public Fecha() {
        ahora = time();
        }
    private native int time();
    static {
        System.loadLibrary( "time" );
        }
    }
```

Una vez escrito el código Java, se necesitan ejecutar los pasos siguientes para poder integrar el código C o C++:

- Utilizar javah para crear un fichero de cabecera (.h)
- Utilizar javah para crear un fichero de stubs, es decir, que contiene la declaración de las funciones
- Escribir el código del método nativo en C o C++, es decir, rellenar el código de la función, completando el trabajo de javah al crear el fichero de stubs
- Compilar el fichero de stubs y el fichero .c en una librería de carga dinámica (DLL en Windows '95 o libXX.so en Unix)
- Ejecutar la aplicación con el appletviewer

Más adelante trataremos en profundidad los métodos nativos, porque añaden una gran potencia a Java, al permitirle integrar a través de librería dinámica cualquier algoritmo desarrollado en C o C++, lo cual, entre otras cosas, se utiliza como método de protección contra la descompilación completa del código Java.

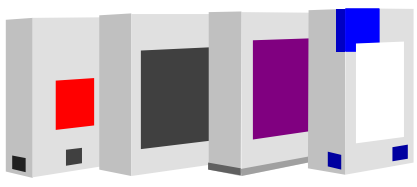
3.13 Paquetes

La palabra clave package permite agrupar clases e interfaces. Los nombres de los paquetes son palabras separadas por puntos y se almacenan en directorios que coinciden con esos nombres. Por ejemplo, los ficheros siguientes, que contienen código fuente Java:

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

contienen en su código la línea:

```
package java.applet;
```



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

java/applet

Import

Los paquetes de clases se cargan con la palabra clave import, especificando el nombre del paquete como ruta y nombre de clase (es lo mismo que #include de C/C++). Se pueden cargar varias clases utilizando un asterisco.

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

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

```
import MiClase;
```

Paquetes de Java

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

java.applet

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

java.awt

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

java.io

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

java.lang

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

java.net

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

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

3.14 Referencias

Java se asemeja mucho a C y C++. Esta similitud, evidentemente intencionada, es la mejor herramienta para los programadores, ya que facilita en gran manera su transición a Java. Desafortunadamente, tantas similitudes hacen que no nos paremos en algunas diferencias que son vitales. La terminología utilizada en estos lenguajes, a veces es la misma, pero hay grandes diferencias subyacentes en su significado.

C tiene tipos de datos básicos y punteros. C++ modifica un poco este panorama y le añade los tipos referencia. Java también especifica sus tipos primitivos, elimina cualquier tipo de punteros y tiene tipos referencia mucho más claros.

Todo este maremágnum de terminología provoca cierta consternación, así que vamos a intentar aclarar lo que realmente significa.
Conocemos ya ampliamente todos los tipos básicos de datos: datos base, integrados, primitivos e internos; que son muy semejantes en C, C++ y Java; aunque Java simplifica un poco su uso a los desarrolladores haciendo que el chequeo de tipos sea bastante más rígido. Además, Java añade los tipos boolean y hace imprescindible el uso de este tipo booleano en sentencias condicionales.

4. PROGRAMAS BÁSICOS EN JAVA

4.1 Una mínima aplicación en java

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

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

HolaMundo

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

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

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

```
class HolaMundoApp {
```

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

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

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

public significa que el método main puede ser llamado por cualquiera, incluyendo el intérprete Java.
static es una palabra clave que le dice al compilador que main se refiere a la propia clase HolaMundoApp y no a ninguna instancia de la clase. De esta forma, si alguien intenta hacer otra instancia de la clase, el método main no se instanciaría.

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

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

%java HolaMundoApp arg1 arg2 ...

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

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

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

```
    }
}
```

Finalmente, se cierran las llaves que limitan el método main() y la clase HolaMundoApp.

Compilacion y Ejecucion de HolaMundo

Vamos a ver a continuación como podemos ver el resultado de nuestra primera aplicación Java en pantalla. Generaremos un fichero con el código fuente de la aplicación, lo compilaremos y utilizaremos el intérprete java para ejecutarlo.

Ficheros Fuente Java

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

Compilación

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

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

```
%javac HolaMundoApp.java
```

Si no se han cometido errores al teclear ni se han tenido problemas con el path al fichero fuente ni al compilador, no debería aparecer mensaje alguno en la

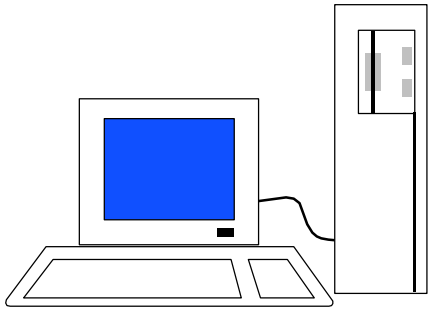
pantalla, y cuando vuelva a aparecer el prompt del sistema, se debería ver un fichero HolaMundoApp.class nuevo en el directorio donde se encuentra el fichero fuente.

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

Ejecución

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

%java HolaMundoApp
y debería aparecer en pantalla la respuesta de Java:
%Hola Mundo!



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

Problemas de compilación

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

%javac: Command not found

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

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

◊ Error tipográfico, el método es println no printl.

%In class HolaMundoApp: main must be public and static

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

%Can´t find class HolaMundoApp

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

```
class HolaMundoapp {
```

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

4.2 El Visor de applets de Sun (appletviewer)

El visualizador de applets (appletviewer) es una aplicación que permite ver en funcionamiento applets, sin necesidad de la utilización de un navegador World-Wide-Web como HotJava, Microsoft Explorer o Nescape. En adelante, recurriremos muchas veces a él, ya que el objetivo del tutorial es el lenguaje Java.

Applet

La definición más extendida de applet, muy bien resumida por Patrick Naughton,

indica que un applet es "una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta in situ como parte de un documento web". Claro que así la definición establece el entorno (Internet, Web, etc.). En realidad, un applet es una aplicación pretendidamente corta (nada impide que ocupe más de un gigabyte, a no ser el pensamiento de que se va a transportar por la red y una mente sensata) basada en un formato gráfico sin representación independiente: es decir, se trata de un elemento a embeber en otras aplicaciones; es un componente en su sentido estricto.

Un ejemplo en otro ámbito de cosas podría ser el siguiente: Imaginemos una empresa, que cansada de empezar siempre a codificar desde cero, diseña un formulario con los datos básicos de una persona (nombre, dirección, etc.). Tal formulario no es un diálogo por sí mismo, pero se podría integrar en diálogos de clientes, proveedores, empleados, etc.

El hecho de que se integre estática (embebido en un ejecutable) o dinámicamente (intérpretes, DLLs, etc.) no afecta en absoluto a la esencia de su comportamiento como componente con que construir diálogos con sentido autónomo.

Pues bien, así es un applet. Lo que ocurre es que, dado que no existe una base adecuada para soportar aplicaciones industriales Java en las que insertar nuestras miniaplicaciones (aunque todo se andará), los applets se han construido mayoritariamente, y con gran acierto comercial (parece), como pequeñas aplicaciones interactivas, con movimiento, luces y sonido... en Internet.

Llamadas a Applets con appletviewer

Un applet es una mínima aplicación Java diseñada para ejecutarse en un navegador Web. Por tanto, no necesita preocuparse por un método main() ni en dónde se realizan las llamadas. El applet asume que el código se está ejecutando desde dentro de un navegador.

El appletviewer se asemeja al mínimo navegador. Espera como argumento el nombre del fichero html que debe cargar, no se le puede pasar directamente un programa Java. Este fichero html debe contener una marca que especifica el código que cargará el appletviewer:

```
<HTML>
<APPLET CODE=HolaMundo.class WIDTH=300 HEIGHT=100>
</APPLET>
</HTML>
```

El appletviewer crear un espacio de navegación, incluyendo un área gráfica, donde se ejecutará el applet, entonces llamará a la clase applet apropiada. En el ejemplo anterior, el appletviewer cargará una clase de nombre HolaMundo y le permitirá trabajar en su espacio gráfico.

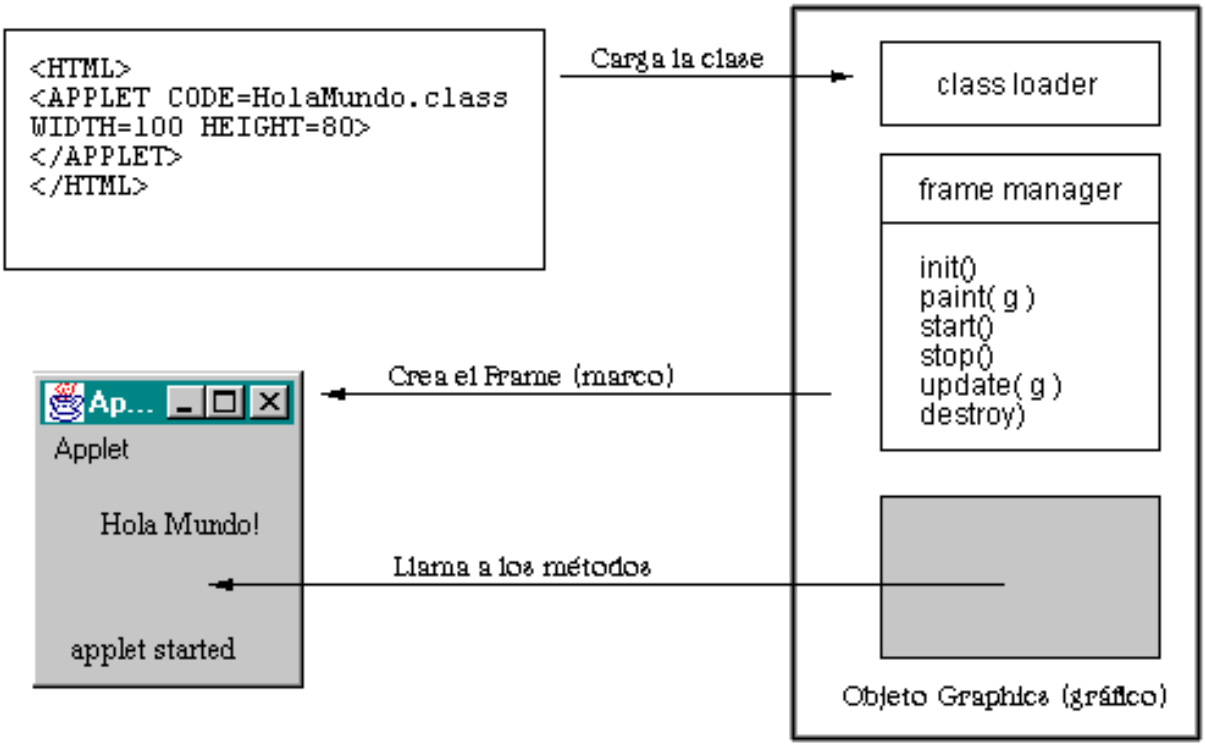
Arquitectura de appletviewer

El appletviewer representa el mínimo interfaz de navegación. En la figura se muestran los pasos que seguiría appletviewer para presentarnos el resultado de la ejecución del código de nuestra clase.

Esta es una visión simplificada del appletviewer. La función principal de esta aplicación es proporcionar al usuario un objeto de tipo Graphics sobre el que dibujar, y varias funciones para facilitar el uso del objeto Graphics.

Ciclo de vida de un Applet

Cuando un applet se carga en el appletviewer, comienza su ciclo de vida, que pasaría por las siguientes fases:



Se crea una instancia de la clase que controla el applet. En el ejemplo de la figura anterior, sería la clase HolaMundo.

El applet se incializa.

El applet comienza a ejecutarse.

El applet empieza a recibir llamadas. Primero recibe una llamada init (inicializar), seguida de un mensaje start (empezar) y paint (pintar). Estas llamadas pueden ser recibidas asíncronamente.

4.3 Escribir Applets Java

Para escribir applets Java, hay que utilizar una serie de métodos, algunos de los cuales ya se hay sumariado al hablar de los métodos del appletviewer, que es el visualizador de applets de Sun. Incluso para el applet más sencillo necesitaremos varios métodos. Son los que se usan para arrancar (start) y detener (stop) la ejecución del applet, para pintar (paint) y actualizar (update) la pantalla y para capturar la información que se pasa al applet desde el fichero HTML a través de la marca APPLET.

init ( )

Esta función miembro es llamada al crearse el applet. Es llamada sólo una vez. La clase Applet no hace nada en init(). Las clases derivadas deben sobrecargar este método para cambiar el tamaño durante su inicialización, y cualquier otra inicialización de los datos que solamente deba realizarse una vez. Deberían realizarse al menos las siguientes acciones:

Carga de imágenes y sonido
El resize del applet para que tenga su tamaño correcto
Asignación de valores a las variables globales

Por ejemplo:

```
public void init() {
   if( width < 200  ||  height < 200 )
      resize( 200,200 );
   valor_global1 = 0;
   valor_global2 = 100;

   // cargaremos imágenes en memoria sin mostrarlas
   // cargaremos música de fondo en memoria sin reproducirla
   }
```

destroy ( )

Esta función miembro es llamada cuando el applet no se va a usar más. La clase Applet no hace nada en este método. Las clases derivadas deberían

sobrecargarlo para hacer una limpieza final. Los applet multithread deberán usar destroy() para "matar" cuanquier thread del applet que quedase activo.

start ( )

Llamada para activar el applet. Esta función miembro es llamada cuando se visita el applet. La clase Applet no hace nada en este método. Las clases derivadas deberían sobrecargarlo para comenzar una animación, sonido, etc.

```
public void start() {
   estaDetenido = false;
   // comenzar la reproducción de la música
   musicClip.play();
   }
```

También se puede utilizar start() para eliminar cualquier thread que se necesite.

stop ( )

Llamada para detener el applet. Se llama cuando el applet desaparece de la pantalla. La clase Applet no hace nada en este método. Las clases derivadas deberían sobrecargarlo para detener la animación, el sonido, etc.

```
public void stop() {
   estaDetenido = true;

   if( /* ¿se está reproduciendo música? */ )
      musicClip.stop();
   }
```

resize (int width,int height)

El método init() debería llamar a esta función miembro para establecer el tamaño del applet. Puede utilizar las variables ancho y alto, pero no es necesario. Cambiar el tamaño en otro sitio que no sea init() produce un reformateo de todo el documento y no se recomienda.

En el navegador Netscape, el tamaño del applet es el que se indica en la marca APPLET del HTML, no hace caso a lo que se indique desde el código Java del applet.

width

Variable entera, su valor es el ancho definido en el parámetro WIDTH de la marca HTML del APPLET. Por defecto es el ancho del icono.

height

Variable entera, su valor es la altura definida en el parámetro HEIGHT de la marca HTML del APPLET. Por defecto es la altura del icono. Tanto width como

height están siempre disponibles para que se puede chequear el tamaño del applet.

Podemos retomar el ejemplo de init():

```
public void init() {
   if( width < 200  ||  height < 200 )
      resize( 200,200 );
   ...
```

paint(Graphics g)

Se llama cada vez que se necesita refrescar el área de dibujo del applet. La clase Applet simplemente dibuja una caja con sombreado de tres dimensiones en el área. Obviamente, la clase derivada debería sobrecargar este método para representar algo inteligente en la pantalla.

Para repintar toda la pantalla cuando llega un evento Paint, se pide el rectángulo sobre el que se va a aplicar paint() y si es más pequeño que el tamaño real del applet se invoca a repaint(), que como va a hacer un update(), se actualizará toda la pantalla.

Podemos utilizar paint() para imprimir nuestro mensaje de bienvenida:

```
void public paint( Graphics g ) {
   g.drawString( "Hola Java!",25,25 );
   // Dibujaremos la imágenes que necesitemos
   }
```

update(Graphics g)

Esta es la función que se llama realmente cuando se necesita actualizar la pantalla. La clase Applet simplemente limpia el área y llama al método paint(). Esta funcionalidad es suficiente en la mayoría de los casos. De cualquier forma, las clases derivadas pueden sustituir esta funcionalidad para sus propósitos.

Podemos, por ejemplo, utilizar update() para modificar selectivamente partes del área gráfica sin tener que pintar el área completa:

```
public void update( Graphics g ) {
   if( estaActualizado )
      {
      g.clear();  // garantiza la pantalla limpia
      repaint();  // podemos usar el método padre: super.update()
      }
   else
      // Información adicional
```

```
        g.drawString( "Otra información",25,50 );
    }
```

repaint()

A esta función se la debería llamar cuando el applet necesite ser repintado. No debería sobrecargarse, sino dejar que Java repinte completamente el contenido del applet.

Al llamar a repaint(), sin parámetros, internamente se llama a update() que borrará el rectángulo sobre el que se redibujará y luego se llama a paint(). Como a repaint() se le pueden pasar parámetros, se puede modificar el rectángulo a repintar.

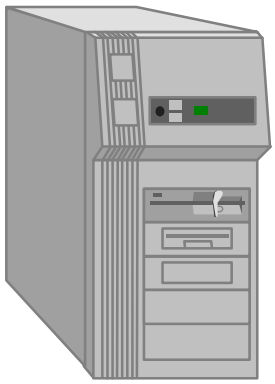
getParameter (String attr)

Este método carga los valores parados al applet vía la marca APPLET de HTML. El argumento String es el nombre del parámetro que se quiere obtener. Devuelve el valor que se le haya asignado al parámetro; en caso de que no se le haya asignado ninguno, devolverá null.

Para usar getParameter(), se define una cadena genérica. Una vez que se ha capturado el parámetro, se utilizan métodos de cadena o de números para convertir el valor obtenido al tipo adecuado.

```
public void init() {
    String pv;

    pv = getParameter( "velocidad" );
    if( pv == null )
        velocidad = 10;
    else
        velocidad = Integer.parseInt( pv );
    }
```



getDocumentBase ( )

Indica la ruta http, o el directorio del disco, de donde se ha recogido la página HTML que contiene el applet, es decir, el lugar donde está la hoja en todo Internet o en el disco.

getCodeBase ( )

Indica la ruta http, o el directorio del disco, de donde se ha cargado el código bytecode que forma el applet, es decir, el lugar donde está el fichero .class en todo Internet o en el disco.

print (Graphics g)

Para imprimir en impresora, al igual que paint() se puede utilizar print(), que pintará en la impresora el mapa de bits del dibujo.

5. EL DEPURADOR DE JAVA - jdb

El depurador de Java, jdb es un depurador de línea de comandos, similar al que Sun proporciona en sus Sistemas, dbx. Es complicado de utilizar y un tanto críptico, por lo que, en principio, tiene escasa practicidad y es necesaria una verdadera emergencia para tener que recurrir a él.

Trataremos por encima los comandos que proporciona el jdb, pero sin entrar en detalles de su funcionamiento, porque no merece la pena. Casi es mejor esperar a disponer de herramientas visuales para poder depurar con cierta comodidad nuestro código Java.

Para poder utilizar el depurador, las aplicaciones Java deben estar compiladas con la opción de depuración activada, -g. Posteriormente se puede lanzar appletviewer con la opción de depuración, debug, y habremos puesto en marcha jdb.

5.1 Depurar HolaMundo

Hemos modificado nuestro applet de ejemplo para utilizarlo en nuestra sesión de ejemplo con el depurador. Se compilaría con el comando:

```
%javac -g hm.java
```

y el contenido de nuestro applet HolaMundo modificado y guardado en el fichero hm.java sería el siguiente:

```
//
// Applet HolaMundo de ejemplo, para depurar
//
import java.awt.Graphics;
import java.applet.Applet;

public class hm extends Applet {
   int i;

   public void paint( Graphics g ) {
      i = 10;
      g.drawString( "Hola Mundo!",25,25 );
      }
  }
```

Una vez compilado, iniciamos la sesión lanzando el visor de applets de Sun con la opción de depuración, utilizando el comando:

%appletviewer -debug hm.html

El fichero hm.html contiene las líneas mínimas para poder activar el applet, estas líneas son las que reproducimos:

```
<html>
<applet code=hm.class width=100 height=100>
</applet>
</html>
```

Se inicia pues la sesión con el depurador y vamos a ir reproduciendo lo que aparece en la pantalla a medida que vamos introduciendo comandos:

```
%appletviewer -debug hm.html
Loading jdb...
0xee301bf0:class(sun.applet.AppletViewer)
>
```

5.2 Comando help

El comando help proporciona una lista de los comandos que están disponibles en la sesión de jdb. Esta lista es la que sigue, en donde hemos aprovechado la presencia de todos los comandos para comentar la acción que cada uno de ellos lleva a cabo.

```
>help
** command list **
threads [threadgroup] -- lista threads
thread <thread id> -- establece el thread por defecto
suspend [thread id(s)] -- suspende threads (por defecto, todos)
resume [thread id(s)] -- continúa threads (por defecto, todos)
where [thread id]|all -- muestra la pila de un thread
threadgroups -- lista los grupos de threads
threadgroup <name> -- establece el grupo de thread actual
print <id> [id(s)] -- imprime un objeto o campo
dump <id> [id(s)] -- imprime toda la información del objeto
locals -- imprime las variables locales de la pila actual
classes -- lista las clases conocidas
methods <class id> -- lista los métodos de una clase
stop in <class id>.<method> -- fija un punto de ruptura en un método
stop at <class id>:<line> -- establece un punto de ruptura en una línea
up [n frames] -- ascender en la pila de threads
```

down [n frames] -- descender en la pila de threads
clear <class id>:<line> -- eliminar un punto de ruptura
step -- ejecutar la línea actual
cont -- continuar la ejecución desde el punto de ruptura
catch <class id> -- parar por la excepción especificada
ignore <class id> -- ignorar la excepción especificada
list [line number] -- imprimir código fuente
use [source file path] -- ver o cambiar la ruta del fichero fuente
memory -- informe del uso de la memoria
load <classname> - carga la clase Java a ser depurada
run <args> - comienza la ejecución de la clase cargada
!! - repite el último comando
help (or ?) - lista los comandos
exit (or quit) - salir del depurador
>

5.3 Comando threadgroups

El comando threadgroups permite ver la lista de threads que se están ejecutando. Los grupos system y main deberían estar siempre corriendo.

```
>threadgroups
1.(java.lang.ThreadGroup)0xee300068 system
2.(java.lang.ThreadGroup)0xee300a98 main
>
```

5.4 Comando threads

El comando threads se utiliza para ver la lista completa de los threads que se están ejecutando actualmente.

```
>threads
Group system:
1.(java.lang.Thread)0xee300098 clock handler cond
2.(java.lang.Thread)0xee300558 Idle thread run
3.(java.lang.Thread)0xee3005d0 sync Garbage Collector cond
4.(java.lang.Thread)0xee300620 Finalizer thread cond
5.(java.lang.Thread)0xee300a20 Debugger agent run
6.(java.tools.debug.BreakpointHandler)0xee300b58) Breakpoint handler cond
Group main:
7.(java.lang.Thread)0xee300048 main suspended
>
```

5.5 Comando run

El comando run es el que se utiliza para arrancar el appletviewer en la sesión de depuración. Lo teclearemos y luego volveremos a listar los threads que hay en ejecución.

```
>run
run sun.applet.AppletViewer hm.html
running...
main[1]threads
threads
Group sun.applet.AppletViewer.main:
1.(java.lang.Thread)0xee3000c0 AWT-Motif running
2.(sun.awt.ScreenUpdater)0xee302ed0 ScreenUpdater cond. Waiting
Group applet-hm.class:
3.(java.lang.Thread)0xee302f38 Thread-6 cond. Waiting
main[1]
```

El visor de applets de Sun aparecerá en la pantalla y mostrará el conocido mensaje de saludo al Mundo. Ahora vamos a rearrancar el appletviewer con un punto de ruptura, para detener la ejecución del applet, y podamos seguir mostrando los comandos disponibles en el jdb.

```
main[1]exit
%appletviewer -debug hm.html
Loading jdb...
0xee3009c8:class(sun.applet.AppletViewer)
>stop in hm.paint
Breakpoint set in hm.paint
>run
run sun.applet.AppletViewer hm.html
running...
Breakpoint hit: hm.paint(hm.java:9)
AWT-Motif[1]
```

5.6 Comando where

El comando where mostrará la pila de ejecución del applet.

```
AWT-Motif[1]where
[1]hm.paint(hm.java:9)
[2]sun.awt.motif.MComponentPeer.paint(MComponenetPeer.java:109)
[3]sun.awt.motif.MComponentPeer.handleExpose(MComponenetPeer.java:170)
AWT-Motif[1]
```

5.7 Comando use

El comando use nos informa del camino donde jdb va a buscar los ficheros fuentes que contienen el código Java de las clases que se están depurando. Por defecto, utilizará el camino que se especifique en la variable de entorno CLASSPATH.

```
AWT-Motif[1]use
/usr/local/java/classes:
AWT-Motif[1]
```

5.8 Comando list

El comando list mostrará el código fuente actual al comienzo del punto de ruptura que hayamos fijado.

```
AWT-Motif[1]list
9       public void paint( Graphics g ) {
10 =>       i = 10;
11          g.drawString( "Hola Mundo!",25,25 ) ;
12          }
13      }
AWT-Motif[1]
```

5.9 Comando dump

El comando dump nos permitirá ahora ver el valor del objeto g pasado desde el appletviewer.

```
AWT-Motif[1]dump g
g = (sun.awt.motif.X11Graphics)0xee303df8 {
  int pData = 1342480
  Color foreground = (java.awt.Color)0xee302378
  Font font = (java.awt.Font)0xee302138
  int originX = 0
  int originY = 0
  float scaleX = 1
  float scaleY = 1
  Image image = null
}
AWT-Motif[1]
```

5.10 Comando step

El comando step nos porporciona el método para ejecutar la línea actual, que estará siendo apuntada por el indicador si hemos utilizado el comando list.

```
AWT-Motif[1]step
```

```
Breakpoint hit: hm.paint(hm.java:11)
AWT-Motif[1]list
9       public void paint( Graphics g ) {
10          i = 10;
11 =>       g.drawString( "Hola Mundo!",25,25 );
12          }
13      }
AWT-Motif[1]
```

6. AWT

6.1 Introducción al AWT

AWT es el acrónimo del X Window Toolkit para Java, donde X puede ser cualquier cosa: Abstract, Alternative, Awkward, Another o Asqueroso; aunque parece que Sun se decanta por Abstracto, seriedad por encima de todo. Se trata de una biblioteca de clases Java para el desarrollo de Interfaces de Usuario Gráficas. La versión del AWT que Sun proporciona con el JDK se desarrolló en sólo dos meses y es la parte más débil de todo lo que representa Java como lenguaje. El entorno que ofrece es demasiado simple, no se han tenido en cuenta las ideas de entornos gráficos novedosos, sino que se ha ahondado en estructuras orientadas a eventos, llenas de callbacks y sin soporte alguno del entorno para la construcción gráfica; veremos que la simple acción de colocar un dibujo sobre un botón se vuelve una tarea harto complicada. Quizá la presión de tener que lanzar algo al mercado haya tenido mucho que ver en la pobreza de AWT.

JavaSoft, asegura que esto sólo era el principio y que AWT será multi-idioma, tendrá herramientas visuales, etc. En fin, al igual que dicen los astrólogos, el futuro nos deparará muchas sorpresas.

La estructura básica del AWT se basa en Componentes y Contenedores. Estos últimos contienen Componentes posicionados a su respecto y son Componentes a su vez, de forma que los eventos pueden tratarse tanto en Contenedores como en Componentes, corriendo por cuenta del programador (todavía no hay herramientas de composición visual) el encaje de todas las piezas, así como la seguridad de tratamiento de los eventos adecuados. Nada trivial.

No obstante y pese a ello, vamos a abordar en este momento la programación con el AWT para tener la base suficiente y poder seguir profundizando en las demás características del lenguaje Java, porque como vamos a ir presentando ejemplos gráficos es imprescindible el conocimiento del AWT. Mientras tanto, esperemos

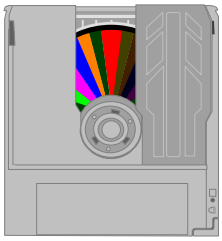
que JavaSoft sea fiel a sus predicciones y lo que ahora veamos nos sirva de base para migrar a un nuevo y maravilloso AWT.

6.2 Interface de Usuario

La interface de usuario es la parte del programa que permite a éste interactuar con el usuario. Las interfaces de usuario pueden adoptar muchas formas, que van desde la simple línea de comandos hasta las interfaces gráficas que proporcionan las aplicaciones más modernas.

La interface de usuario es el aspecto más importante de cualquier aplicación. Una aplicación sin un interfaz fácil, impide que los usuarios saquen el máximo rendimiento del programa. Java proporciona los elementos básicos para construir decentes interfaces de usuario a través del AWT.

Al nivel más bajo, el sistema operativo transmite información desde el ratón y el teclado como dispositivos de entrada al programa. El AWT fue diseñado pensando en que el programador no tuviese que preocuparse de detalles como controlar el movimiento del ratón o leer el teclado, ni tampoco atender a detalles como la escritura en pantalla. El AWT constituye una librería de clases orientada a objeto para cubrir estos recursos y servicios de bajo nivel.



Debido a que el lenguaje de programación Java es independiente de la plataforma en que se ejecuten sus aplicaciones, el AWT también es independiente de la plataforma en que se ejecute. El AWT proporciona un conjunto de herramientas para la construcción de interfaces gráficas que tienen una apariencia y se comportan de forma semejante en todas las plataformas en que se ejecute. Los elementos de interface proporcionados por el AWT están implementados utilizando toolkits nativos de las plataformas, preservando una apariencia semejante a todas las aplicaciones que se creen para esa plataforma. Este es un punto fuerte del AWT, pero también tiene la desventaja de que una interface gráfica diseñada para una plataforma, puede no visualizarse correctamente en otra diferente.

6.3 Estructura del AWT

La estructura de la versión actual del AWT podemos resumirla en los puntos que exponemos a continuación:

- ☛ Los Contenedores contienen Componentes, que son los controles básicos
- ☛ No se usan posiciones fijas de los Componentes, sino que están situados a través de una disposición controlada (layouts)
- ☛ El común denominador de más bajo nivel se acerca al teclado, ratón y manejo de eventos

- Alto nivel de abstracción respecto al entorno de ventanas en que se ejecute la aplicación (no hay áreas cliente, ni llamadas a X, ni hWnds, etc.)
- La arquitectura de la aplicación es dependiente del entorno de ventanas, en vez de tener un tamaño fijo
- Es bastante dependiente de la máquina en que se ejecuta la aplicación (no puede asumir que un diálogo tendrá el mismo tamaño en cada máquina)
- Carece de un formato de recursos. No se puede separar el código de lo que es propiamente interface. No hay ningún diseñador de interfaces (todavía)

6.4 Componentes y Contenedores

Una interface gráfica está construida en base a elementos gráficos básicos, los Componentes. Típicos ejemplos de estos Componentes son los botones, barras de desplazamiento, etiquetas, listas, cajas de selección o campos de texto. Los Componentes permiten al usuario interactuar con la aplicación y proporcionar información desde el programa al usuario sobre el estado del programa. En el AWT, todos los Componentes de la interface de usuario son instancias de la clase Component o uno de sus subtipos.

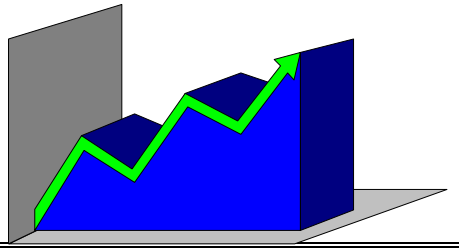
Los Componentes no se encuentran aislados, sino agrupados dentro de Contenedores. Los Contenedores contienen y organizan la situación de los Componentes; además, los Contenedores son en sí mismos Componentes y como tales pueden ser situados dentro de otros Contenedores. También contienen el código necesario para el control de eventos, cambiar la forma del cursor o modificar el icono de la aplicación. En el AWT, todos los Contenedores son instancias de la clase Container o uno de sus subtipos.

Los Componentes deben circunscribirse dentro del Contenedor que los contiene. Esto hace que el anidamiento de Componentes (incluyendo Contenedores) en Contenedores crean árboles de elementos, comenzando con un Contenedor en la raiz del árbol y expandiéndolo en sus ramas.

7. GRAFICOS

7.1 Objetos Gráficos

En páginas anteriores ya se ha mostrado cómo escribir applets, cómo lanzarlos y los fundamentos básicos de la presentación de información sobre ellos. Ahora, pues, querremos hacer cosas más interesantes que mostrar texto; ya que cualquier página HTML puede mostrar texto. Para ello, Java proporciona la clase Graphics, que permite mostrar texto a través del método drawString(), pero también



tiene muchos otros métodos de dibujo. Para cualquier programador, es esencial el entendimiento de la clase Graphics, antes de adentrarse en el dibujo de cualquier cosa en Java.
Esta clase proporciona el entorno de trabajo para cualquier operación gráfica que se realice dentro del AWT. Juega dos importantes papeles: por un lado, es el contexto gráfico, es decir, contiene la información que va a afectar a todas las operaciones gráficas, incluyendo los colores de fondo y texto, la fuente de caracteres, la localización y dimensiones del rectángulo en que se va a pintar, e incluso dispone de información sobre el eventual destino de las operaciones gráficas (pantalla o imagen). Por otro lado, la clase Graphics proporciona métodos que permiten el dibujo de primitivas, figuras y la manipulación de fonts de caracteres y colores. También hay clases para la manipulación de imágenes, doble-buffering, etc.

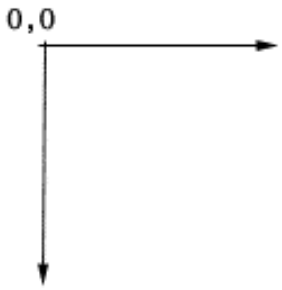
Para poder pintar, un programa necesita un contexto gráfico válido, representado por una instancia de la clase Graphics. Pero, como esta clase es abstracta, no se puede instanciar directamente; así que debemos crear un componente y pasarlo al programa como un argumento a los métodos paint() o update().

Los dos métodos anteriores, paint() y update(), junto con el método repaint() son los que están involucrados en la presentación de gráficos en pantalla. El AWT, para reducir el tiempo que necesitan estos métodos para realizar el repintado en pantalla de gráficos, tiene dos axiomas:

- Primero, el AWT repinta solamente aquellos Componentes que necesitan ser repintados, bien porque estuviesen cubiertos por otra ventana o porque se pida su repintado directamente
- Segundo, si un Componente estaba tapado y se destapa, el AWT repinta solamente la porción del Componente que estaba oculta

En la ejecución del applet que aparece a continuación, EjemploGraf.java, podemos observar como se realiza este proceso. Ignorar la zona de texto de la parte superior del applet de momento, y centrar la mirada en la parte coloreada. Utilizando otra ventana, tapar y destapar parte de la zona que ocupa el applet. Se observará que solamente el trozo de applet que estaba cubierto es el que se repinta. Yendo un poco más allá, solamente aquellos componentes que estén ocultos y se vuelvan a ver serán los que se repinten, sin tener en cuenta su posición dentro de la jerarquía de componentes.

La pantalla en Java se incrementa de izquierda a derecha y de arriba hacia abajo, tal como muestra la figura:



Los pixels de la pantalla son pues: posición 0 + ancho de la pantalla - 1.
En los textos, el punto de inserción se encuentra en la línea base de la primera letra.

7.2 Métodos para Dibujos

Vamos a presentar métodos para dibujar varias figuras geométricas. Como estos métodos funcionan solamente cuando son invocados por una instancia válida de la clase Graphics, su ámbito de aplicación se restringe a los componentes que se utilicen en los métodos paint() y update(). Normalmente los métodos de dibujo de primitivas gráficas funcionan por pares: un método pinta la figura normal y el otro pinta la figura rellena.

drawLine(x1,y1,x2,y2)
drawRect(x,y,ancho,alto)

fillRect(x,y,ancho,alto)
clearRect(x,y,ancho.alto)
drawRoundRect(x,y,ancho,alto,anchoArco,altoArco)
fillRoundRect(x,y,ancho,alto,anchoArco,altoArco)
draw3DRect(x,y,ancho,alto,boolean elevado)
fill3DRect(x,y,ancho,alto,boolean elevado)
drawOval(x,y,ancho,alto)
fillOval(x,y,ancho,alto)
drawArc(x,y,ancho,alto,anguloInicio,anguloArco)
fillArc(x,y,ancho,alto,anguloInicio,anguloArco)
drawPolygon(int[] puntosX,int[] puntosY[],numPuntos)
fillPolygon(int[] puntosX,int[] puntosY[],numPuntos)
drawString(string s,x,y)
drawChars(char data[],offset,longitud,x,y)
drawBytes(byte data[],offset,longitud,x,y)
copyArea(xSrc,ySrc,ancho,alto,xDest,yDest)

7.3 Métodos para Imagenes

Los objetos Graphics pueden mostrar imágenes a través del método:

drawImage(Image img,int x,int y,ImageObserver observador);

Hay que tener en cuenta que el método drawImage() necesita un objeto Image y un objeto ImageObserver. Podemos cargar una imagen desde un fichero de dibujo (actualmente sólo se soportan formatos GIF y JPEG) con el método getImage():
Image img = getImage(getDocumentBase(),"fichero.gif");

La forma de invocar al método getImage() es indicando un URL donde se encuentre el fichero que contiene la imagen que queremos presentar y el nombre de ese fichero:
getImage(URL directorioImagen,String ficheroImagen);

Un URL común para el método getImage() es el directorio donde está el fichero HTML. Se puede acceder a esa localización a través del método getDocumentBase() de la clase Applet, como ya se ha indicado.

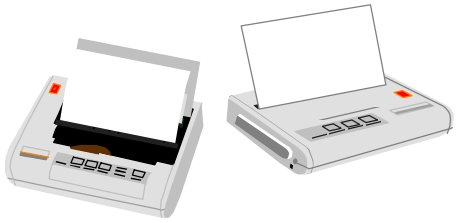
Normalmente, se realiza el getImage() en el método init() del applet y se muestra la imagen cargada en el método paint(), tal como se muestra en el ejemplo siguiente:

```
public void init() {
   img = getImage( getDocumentBase(),"pepe.gif" );
   }

public void paint( Graphics g ) {
```

```
    g.drawImage( img,x,y,this );
    }
```

En el applet Imagen.java, podemos ver el ejemplo completo. Su ponemos en él la existencia del fichero "Imagenes/pepe.gif":



```
import java.awt.*;
import sun.awt.image.URLImageSource;
import java.applet.Applet;

public class Imagen extends Applet {
    Imagen pepe;

    public void init() {
        pepe = getImage( getDocumentBase(),"Imagenes/pepe.gif" );
        }

    public void paint( Graphics g ) {
        g.drawString( pepe,25,25,this );
        }
    }
```

7.4 Sonido en Java

Java también posee métodos predefinidos para reproducir sonido. El ordenador remoto no necesita tener un reproductor de audio; Java realizará la reproducción (evidentemente, el ordenador remoto, en donde se ejecuta el applet, necesitará disponer de hardware de sonido).

Reproducción de sonido

La forma más fácil de reproducir sonido es a través del método play():

```
play( URL directorioSonido,String ficheroSonido );
```

o, simplemente:

```
play( URL unURLdeSonido );
```

Un URL común para el método play() es el directorio donde está el fichero HTML. Se puede acceder a esa localización a través del método getDocumentBase() de la clase Applet:

```
play( getDocumentBase(),"sonido.au" );
```

para que esto funcione, el fichero de la clase y el fichero sonido.au deberían estar en el mismo directorio.

Reproducción Repetitiva

Se puede manejar el sonido como si de imágenes se tratara. Se pueden cargar y reproducir más tarde.
Para cargar un clip de sonido, se utiliza el método getAudioClip():
AudoClip sonido;
sonido = getAudioClip(getDocumentBase(),"risas.au");

Una vez que se carga el clip de sonido, se pueden utilizar tres métodos:
sonido.play();
para reproducir el clip de sonido.
sonido.loop();
para iniciar la reproducción del clip de sonido y que entre en un blucle de reproducción, es decir, en una repetición automática del clip.
sonido.stop();
para detener el clip de sonido que se encuentre en ese instante en reproducción.

7.5 Entrada por Ratón

Una de las características más útiles que ofrece Java es el soporte directo de la interactividad. La aplicación puede reaccionar a los cambios producidos en el ratón, por ejemplo, sin necesidad de escribir ninguna línea de código para su control, solamente indicando qué se quiere hacer cuando el ratón haga algo.
El evento más común en el ratón es el click. Este evento es gobernado por dos métodos: mouseDown() (botón pulsado) y mouseUp() (botón soltado). Ambos métodos son parte de la clase Applet, pero se necesita definir sus acciones asociadas, de la misma forma que se realiza con init() o con paint().

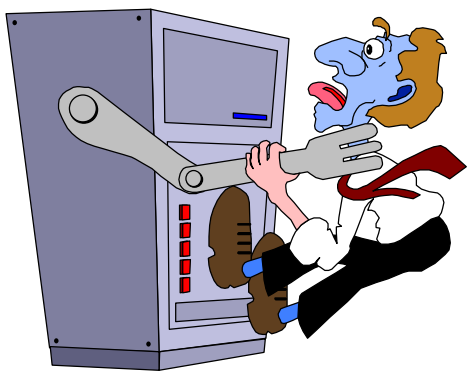
8. EXCEPCIONES EN JAVA

8.1 Manejo de Excepciones

Vamos a mostrar como se utilizan las excepciones, reconvirtiendo nuestro applet de saludo a partir de la versión iterativa de HolaIte.java:

```
import java.awt.*;
import java.applet.Applet;

public class HolaIte extends Applet {
    private int i = 0;
    private String Saludos[] = {
        "Hola Mundo!",
        "HOLA Mundo!",
```



```
    "HOLA MUNDO!!"
    };

  public void paint( Graphics g ) {
    g.drawString( Saludos[i],25,25 );
    i++;
    }
  }
```

Normalmente, un programa termina con un mensaje de error cuando se lanza una excepción. Sin embargo, Java tiene mecanismos para excepciones que permiten ver qué excepción se ha producido e intentar recuperarse de ella.

Vamos a reescribir el método paint() de nuestra versión iterativa del saludo:

```
public void paint( Graphics g ) {
  try {
    g.drawString( Saludos[i],25,25 );
    } catch( ArrayIndexOutOfBoundsException e ) {
      g.drawString( "Saludos desbordado",25,25 );
    } catch( Exception e ) {
      // Cualquier otra excepción
      System.out.println( e.toString() );
    } finally {
      System.out.println( "Esto se imprime siempre!" );
    }
  i++;
  }
```

La palabra clave finally define un bloque de código que se quiere que sea ejecutado siempre, de acuerdo a si se capturó la excepción o no. En el ejemplo anterior, la salida en la consola, con i=4 sería:

```
Saludos desbordado
¡Esto se imprime siempre!
```

8.2 Generar Excepciones en Java

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

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

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

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

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

```
throw ObtejoExcepction;
```

El objeto ObjetoException es un objeto de una clase que extiende la clase Exception. El siguiente código de ejemplo origina una excepción de división por cero:

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

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

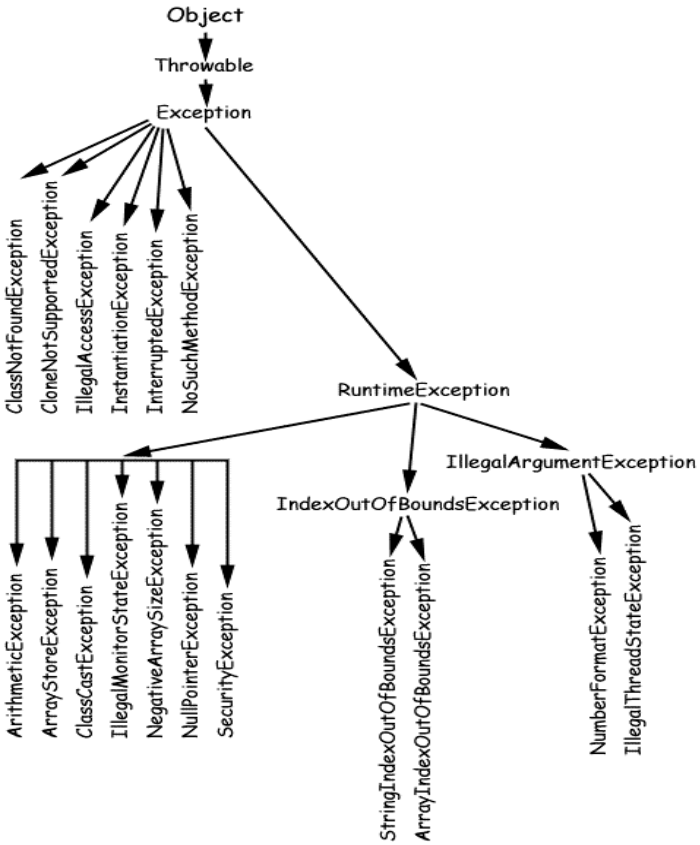
Si compilamos y ejecutamos esta aplicación Java, obtendremos la siguiente salida por pantalla:

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

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

8.3 Excepciones Predefinidas

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



Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:

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

```
int i = 12 / 0;
```

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

```
class Hola extends Applet {
   Image img;
   paint( Graphics g ) {
      g.drawImage( img,25,25,this );
      }
   }
```

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

ClassCastException

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

```
y = (Prueba)x;      // donde
```

x no es de tipo Prueba

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

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

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

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

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

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

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

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

8.4 Crear Excepciones Propias

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

Cualquier método que lance una excepción también debe capturarla, o declararla como parte de la interface del método. Cabe preguntarse entonces, el porqué de

lanzar una excepción si hay que capturarla en el mismo método. La respuesta es que las excepciones no simplifican el trabajo del control de errores. Tienen la ventaja de que se puede tener muy localizado el control de errores y no tenemos que controlar millones de valores de retorno, pero no van más allá.

8.5 Capturar Excepciones

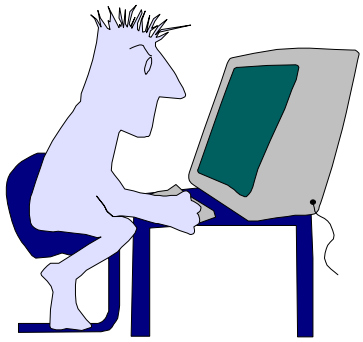
Las excepciones lanzadas por un método que pueda hacerlo deben recoger en bloque try/catch o
try/finally.

try

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

catch

Es el código que se ejecuta cuando se produce la excepción. Es como si dijésemos "controlo cualquier excepción que coincida con mi argumento". En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula. Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.



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

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

finally

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

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

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

8.6 Propagacion de Excepciones

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

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

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

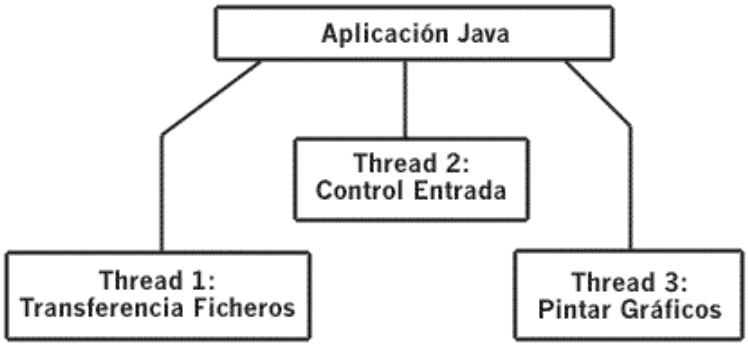
Veamos lo que sucede cuando una excepción no es tratada en la rutina en donde se produce. El sistema Java busca un bloque try..catch más allá de la llamada, pero dentro del método que lo trajo aquí. Si la excepción se propaga de todas formas hasta lo alto de la pila de llamadas sin encontrar un controlador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

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

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

9. Threads y Multithreading

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



9.1 Flujo en Programas

Programas de flujo único

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

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

Si se ha utilizado un navegador con soporte Java, ya se habrá visto el uso de múltiples threads en Java. Habrá observado que dos applet se pueden ejecutar al mismo tiempo, o que puede desplazarla página del navegador mientras el applet

continúa ejecutándose. Esto no significa que el applet utilice múltiples threads, sino que el navegador es multithreaded.

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

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

9.2 Creacion y Control de threads

Creación de un Thread

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

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

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

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

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

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

```
    . . .
    }
```

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

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

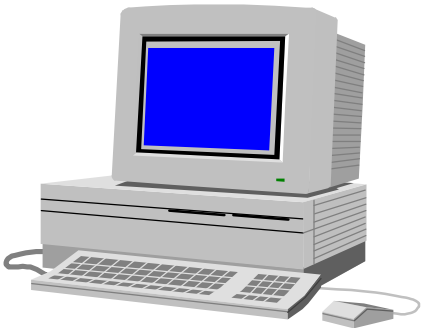
En este caso necesitamos crear una instancia de Thread antes de que el sistema pueda ejecutar el proceso como un thread. Además, el método abstracto run() está definido en la interface Runnable tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible.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

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

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

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

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



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

Arranque de un Thread

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

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

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

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

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

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

Manipulación de un Thread

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

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

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

```
sleep( retardo );
```

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

Suspensión de un Thread

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

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

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

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

Parada de un Thread

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

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

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

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

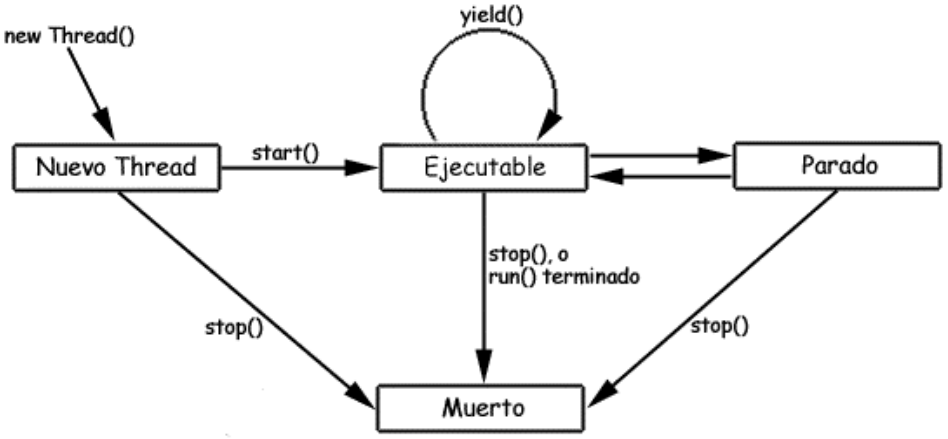
Si se necesita, se puede comprobar si un thread está vivo o no; considerando vivo un thread que ha comenzado y no ha sido detenido.

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

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

9.3 Estados de un thread

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



Nuevo Thread

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

Ejecutable

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

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

Parado

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

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

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

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

Muerto

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

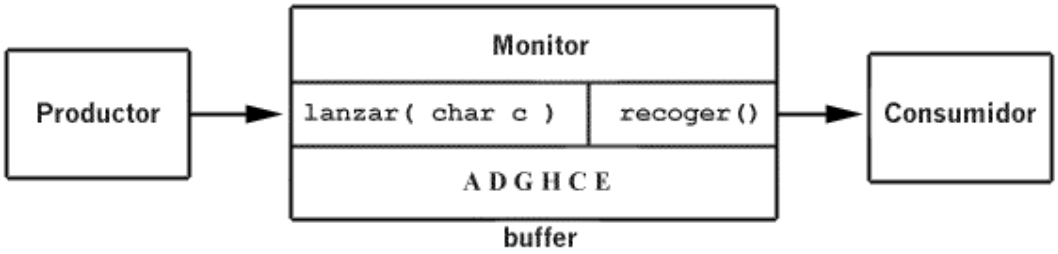
El método isAlive()

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

9.4 Comunicacion entre threads

Otra clave para el éxito y la ventaja de la utilización de múltiples threads en una aplicación, o aplicación multithreaded, es que pueden comunicarse entre sí. Se pueden diseñar threads para utilizar objetos comunes, que cada thread puede manipular independientemente de los otros threads.

El ejemplo clásico de comunicación de threads es un modelo productor/consumidor. Un thread produce una salida, que otro thread usa (consume), sea lo que sea esa salida. Vamos entonces a crear un productor, que será un thread que irá sacando caracteres por su salida; crearemos también un consumidor que ira recogiendo los caracteres que vaya sacando el productor y un monitor que controlará el proceso de sincronización entre los threads.
Funcionará como una tubería, insertando el productor caracteres en un extremos y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.



11.METODOS NATIVOS

Un método nativo es un método Java (una instancia de un objeto o una clase) cuya implementación se ha realizado en otro lenguaje de programación, por ejemplo, C. Vamos a ver cómo se integran métodos nativos en clases Java. Actualmente, el lenguaje Java solamente proporciona mecanismos para integrar código C en programas Java.
Veamos pues los pasos necesarios para mezclar código nativo C y programas Java. Recurriremos (¡Cómo no!) a nuestro saludo; en este caso, el programa HolaMundo tiene dos clases Java: la primera implementa el método main() y la segunda, HolaMundo, tiene un método nativo que presenta el mensaje de saludo. La implementación de este segundo método la realizaremos en C.

10.1 Escribir Código Java

En primer lugar, debemos crear una clase Java, HolaMundo, que declare un método nativo. También debemos crear el programa principal que cree el objeto HolaMundo y llame al método nativo.

Las siguientes líneas de código definen la clase HolaMundo, que consta de un método y un segmento estático de código:

```
class HolaMundo {
   public native void presentaSaludo();
   static {
      System.loadLibrary( "hola" );
      }
   }
```

Podemos decir que la implementación del método presentaSaludo() de la clase HolaMundo está escrito en otro lenguaje, porque la palabra reservada native aparece como parte de la definición del método. Esta definición, proporciona solamente la definición para presentaSaludo() y no porporciona ninguna implementación para él. La implementación la proporcionaremos desde un fichero fuente separado, escrito en lenguaje C.
La definición para presentaSaludo() también indica que el método es un método público, no acepta argumentos y no devuelve ningún valor. Al igual que cualquier otro método, los métodos nativos deben estar definidos dentro de una clase Java.

El código C que implementa el método presentaSaludo() debe ser compilado en una librería dinámica y cargado en la clase Java que lo necesite. Esta carga, mapea la implementación del método nativo sobre su definición.

El siguiente bloque de código carga la librería dinámica, en este caso hola. El sistema Java ejecutará un bloque de código estático de la clase cuando la cargue.

Todo el código anterior forma parte del fichero HolaMundo.java, que contiene la clase HolaMundo. En un fichero separado, Main.java, vamos a crear una aplicación Java que instancie a la clase HolaMundo y llame al método nativo presentaSaludo().

```
class Main {
   public static void main( String args[] ) {
      new HolaMundo().presentaSaludo();
      }
   }
```

Como se puede observar, llamamos al método nativo del mismo modo que a cualquier otro método Java; añadimos el nombre del método al final del nombre del objeto con un punto ("."). El conjunto de paréntesis que sigue al nombre del

método encierra los argumentos que se le pasen. En este caso, el método presentaSaludo() no recibe ningún tipo de argumento.

10.2 Compilar el Código Java

Utilizaremos ahora el compilador javac para compilar el código Java que hemos desarrollado. Compilaremos los dos ficheros fuentes de código Java que hemos creado, tecleando los siguientes comandos:

```
> javac HolaMundo.java
> javac Main.java
```

10.3 Crear el Fichero de Cabecera

Ahora debemos utilizar la aplicación javah para conseguir el fichero de cabecera .h. El fichero de cabecera define una estructura que representa la clase HolaMundo sobre código C y proporciona la definición de una función C para la implementación del método nativo presentaSaludo() definido en ese clase.
Ejecutamos javah sobre la clase HolaMundo, con el siguiente comando:

```
> javah HolaMundo
```

Por defecto, javah creará el nuevo fichero .h en el mismo directorio en que se encuentra el fichero .class, obtenido al compilar con javac el código fuente Java correspondiente a la clase. El fichero que creará, será un fichero de cabecera del mismo nombre que la clase y con extensión .h. Por ejemplo, el comando anterior habrá creado el fichero HolaMundo.h, cuyo contenido será el siguiente:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class HolaMundo */

#ifndef _Included_HolaMundo
#define _Included_HolaMundo

typedef struct ClassHolaMundo {
    char PAD; /* ANSI C requires structures to have a least one member */
} ClassHolaMundo;
HandleTo(HolaMundo);
#ifdef __cplusplus
extern "C" {
#endif
__declspec(dllexport) void HolaMundo_presentaSaludo(struct HHolaMundo *);
#ifdef __cplusplus
}
#endif
```

#endif

Este fichero de cabecera contiene la definición de una estructura llamada ClassHolaMundo. Los miembros de esta estructura son paralelos a los miembros de la clase Java correspondiente; es decir, los campos en la estructura corresponden a las variables de la clase. Pero como HolaMundo no tiene ninguna variable, la estructura se encuentra vacía. Se pueden utilizar los miembros de la estructura para referenciar a variables instanciadas de la clase desde las funciones C. Además de la estructura C similar a la clase Java, vemos que la llamada de la función C está declarada como:
extern void HolaMundo_presentaSaludo(struct HHolaMundo *);

Esta es la definición de la función C que deberemos escribir para implementar el método nativo presentaSaludo() de la clase HolaMundo. Debemos utilizar esa definición cuando lo implementemos. Si HolaMundo llamase a otros métodos nativos, las definiciones de las funciones también aparecerían aquí.
El nombre de la función C que implementa el método nativo está derivado del nombre del paquete, el nombre de la clase y el nombre del método nativo. Así, el método nativo presentaSaludo() dentro de la clase HolaMundo es HolaMundo_presentaSaludo(). En este ejemplo, no hay nombre de paquete porque HolaMundo se considera englobado dentro del paquete por defecto.
La función C acepta un parámetro, aunque el método nativo definido en la clase Java no acepte ninguno. Se puede pensar en este parámetro como si fuese la variable this de C++. En nuestro caso, ignoramos el parámetro this.

10.4 Crear el Fichero de stubs

Volvemos a utilizar la aplicación javah para crear el fichero de stubs, que contiene todas las declaraciones de métodos, con sus llamadas y argumentos, listos para que nosotros rellenemos el cuerpo de los métodos con los algoritmos que necesitemos implementar. Proporciona la unión entre la clase Java y su estructura C paralela.

Para generar este fichero, debemos indicar el parámetro .stubs al ejecutar la aplicación javah sobre la clase HolaMundo, de la siguiente forma:

> javah -stubs HolaMundo

Del mismo modo que se generaba el fichero .h; el nombre del fichero de stubs será el nombre de la clase con la extensión .c. En nuestro ejemplo, será HolaMundo.c, y su contenido será el siguiente:

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <StubPreamble.h>

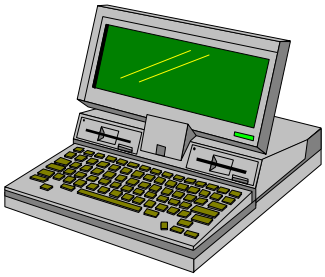
```
/* Stubs for class HolaMundo */
/* SYMBOL: "HolaMundo/presentaSaludo()V",
Java_HolaMundo_presentaSaludo_stub */
__declspec(dllexport) stack_item
*Java_HolaMundo_presentaSaludo_stub(stack_item *_P_,struct execenv *_EE_) {
 extern void HolaMundo_presentaSaludo(void *);
 (void) HolaMundo_presentaSaludo(_P_[0].p);
 return _P_;
}
```

10.5 Escribir la funcion C

Escribiremos la función C para el método nativo en un fichero fuente de código C. La implementación será una función habitual C, que luego integraremos con la clase Java. La definición de la función C debe ser la misma que la que se ha generado con javah en el fichero HolaMundo.h. La implementación que proponemos la guardaremos en el fichero HolaImp.c, y contendrá las siguientes línea de código:

```
#include <StubPreamble.h>
#include "HolaMundo.h>
#include <stdio.h>

void HolaMundo_presentaSaludo( struct HHolaMundo *this
) {
    printf( "Hola Mundo, desde el Tutorial de Java\n" );
    return;
    }
```



Como se puede ver, la implementación no puede ser más sencilla: hace una llamada a la función printf() para presentar el saludo y sale.
En el código se incluyen tres ficheros de cabecera:

StubsPreamble.h

Proporciona la información para que el código C pueda interactuar con el sistema Java. Cuando se escriben métodos nativos, siempre habrá que incluir este fichero en el código fuente C.

HolaMundo.h

Es el fichero de cabecera que hemos generado para nuestra clase. Contiene la estructura C que representa la clase Java para la que estamos escribiendo el método nativo y la definición de la función para ese método nativo.

stdio.h

Es necesario incluirlo porque utilizamos la función printf() de la librería estándar de C, cuya declaración se encuentra en este fichero de cabecera.

10.6 Crear la Libreria Dinámica

Utilizaremos el compilador C para compilar el fichero .h, el fichero de stubs y el fichero fuente .c; para crear una librería dinámica. Para crearla, utilizaremos el compilador C de nuestro sistema, haciendo que los ficheros HolaMundo.c y HolaImp.c generen una librería dinámica de nombre hola, que será la que el sistema Java cargue cuando ejecute la aplicación que estamos construyendo.
Vamos a ver cómo generamos esta librería en Unix y en Windows '95.

Unix

Teclearemos el siguiente comando:

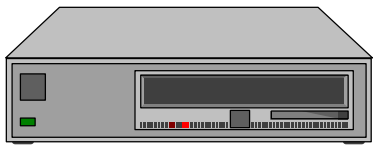
```
% cc -G HolaMundo.c HolaImp.c -o libhola.so
```

En caso de que no encuentre el compilador los ficheros de cabecera, se puede utilizar el flag -I para indicarle el camino de búsqueda, por ejemplo:

```
% cc -G -I$JAVA_HOME/include HolaMundo.c HolaImp.c -o libhola.so
```

donde $JAVA_HOME es el directorio donde se ha instalado la versión actual del Java Development Kit.

Windows '95



El comando a utilizar en este caso es el siguiente:

```
c:\>cl HolaMundo.c HolaImp.c -Fhola.dll -MD -LD javai.lib
```

Este comando funciona con Microsoft Visual C++ 2.x y posteriores. Si queremos indicar al compilador donde se encuentran los ficheros de cabecera y las librerías, tendremos que fijar dos variables de entorno:

```
c:\>SET INCLUDE=%JAVAHOME%\include;%INCLUDE%
c:\>SET LIB=%JAVAHOME%\lib;%LIB%
```

donde %JAVAHOME% es el directorio donde se ha instalado la versión actual del Java Development Kit.

10.7 Ejecutar el Programa

Y, por fin, utilizaremos el intérprete de Java, java, para ejecutar el programa que hemos construido siguiendo todos los pasos anteriormente descritos. Si tecleamos el comando:

> java Main

obtendremos el resultado siguiente:
% Hola Mundo, desde el Tutorial de Java

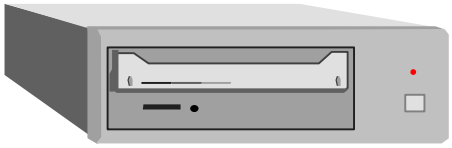
Si no aparece este mensaje de saludo y lo que aparece en pantalla son expresiones como UnsatisfiedLinkError, es porque no tenemos fijado correctamente el camino de la librería dinámica que hemos generado. Este camino es la lista de directorios que el sistema Java utilizará para buscar las librerías que debe cargar. Debemos asegurarnos de que el directorio donde se encuentra nuestra librería hola recién creada, figura entre ellos.

Si fijamos el camino correcto y ejecutamos de nuevo el programa, veremos que ahora sí obtenemos el mensaje de saludo que esperábamos.

Con ello, hemos visto como integrar código C en programas Java. Quedan muchas cuestiones por medio, como la equivalencia de tipos entre Java y C, el paso de parámetros, el manejo de cadenas, etc. Pero eso supondría entrar en mucha más profundidad dentro de Java de la que aquí pretendemos (por ahora).

11. Entrada/Salida Estándar

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



11.1 La clase System

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

Stdin

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

Stdout

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

Stderr

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

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

```
import java.io.*;
class miType {
   public static void main( String args[] ) throws IOException {
      int c;
      int contador = 0;

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

11.2 Clases comunes de Entrada/Salida

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

- FileInputStream
- DataInputStream
- FileOutputStream
- DataOutputStream

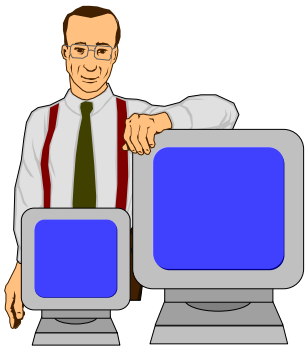
También existen otras clases para aplicaciones más específicas, que no vamos a tratar, por ser de un uso muy concreto:

- PipedInputStream
- BufferedInputStream

PushBackInputStream
StreamTokenizer
PipedOutputStream
BufferedOutputStream

12. FICHEROS EN JAVA

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



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

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

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

12.1 Creación de un objeto File

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

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

o

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

o

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

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

12.2 Ficheros de Acceso Aleatorio

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

Creación de un Fichero de Acceso Aleatorio

Hay dos posibilidades para abrir un fichero de acceso aleatorio:
Con el nombre del fichero:

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

Con un objeto File:

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

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

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

Acceso a la Información

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

También se tienen muchos métodos para moverse dentro de un fichero:
long getFilePointer();
Devuelve la posición actual del puntero del fichero
void seek(long pos);

Coloca el puntero del fichero en una posición determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 marca el comienzo de ese fichero.
long length();
Devuelve la longitud del fichero. La posición length() marca el final de ese fichero.

Actualización de Información

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

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

```
    // añadirá información al fichero
```

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

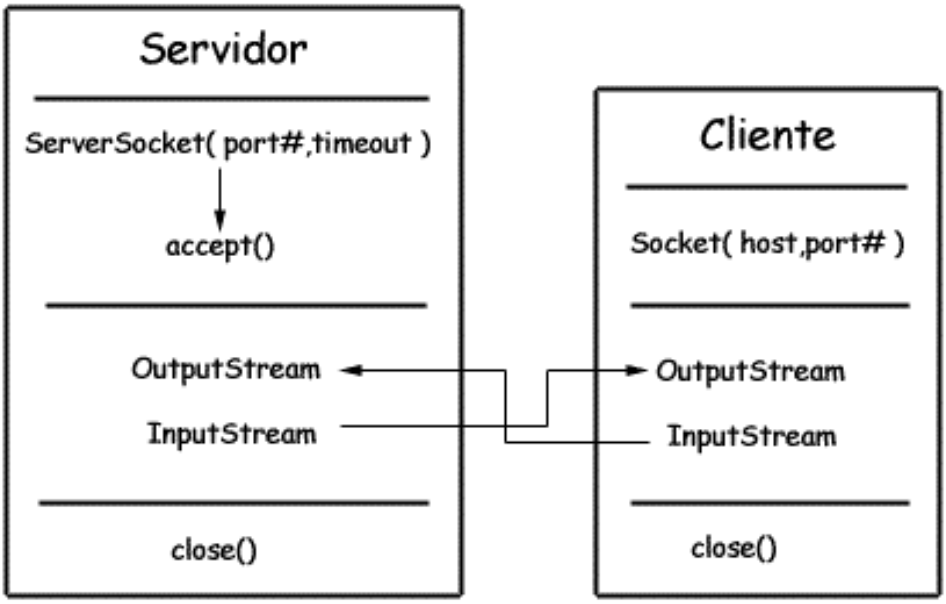
```
import java.io.*;

// Cada vez que ejecutemos este programita, se incorporara una nueva
// linea al fichero de log que se crea la primera vez que se
ejecuta
//
//
class Log {
    public static void main( String args[] ) throws IOException {
        RandomAccessFile miRAFile;
        String s = "Informacion a incorporar\nTutorial de Java\n";
        // Abrimos el fichero de acceso aleatorio
        miRAFile = new RandomAccessFile( "/tmp/java.log","rw"
);
        // Nos vamos al final del fichero
        miRAFile.seek( miRAFile.length() );
        // Incorporamos la cadena al fichero
        miRAFile.writeBytes( s );
        // Cerramos el fichero
        miRAFile.close();
        }
    }
```

13. COMUNICACIONES EN JAVA

13.1 Modelo de Comunicaciones con Java

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



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

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

13.2 Clases Utiles en Comunicaciones

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

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

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

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

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

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

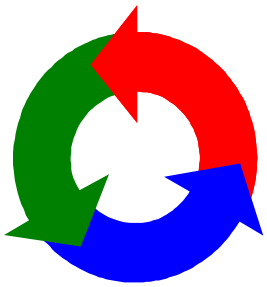
NetworkServer
Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.

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

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

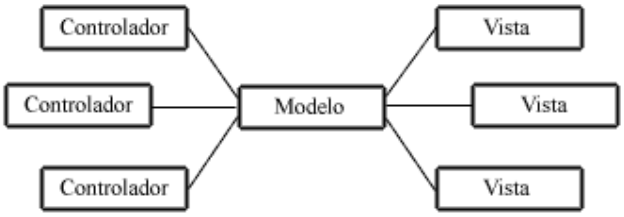
14. ARQUITECTURA Modelo/Vista/Controlador

La arquitectura MVC (Model/View/Controller) fue introducida como parte de la versión Smalltalk-80 del lenguaje de programación Smalltalk. Fue diseñada para reducir el esfuerzo de programación necesario en la implementación de sistemas múltiples y sincronizados de los mismos datos. Sus características principales son que el Modelo, las Vistas y los Controladores se tratan como entidades separadas; esto hace que cualquier cambio producido en el Modelo se refleje automáticamente en cada una de las Vistas.



Además del programa ejemplo que hemos presentado al principio y que posteriormente implementaremos, este modelo de arquitectura se puede emplear en sistemas de representación gráfica de datos, como se ha citado, o en sistemas CAD, en donde se presentan partes del diseño con diferente escala de aumento, en ventanas separadas.

En la figura siguiente, vemos la arquitectura MVC en su forma más general. Hay un Modelo, múltiples Controladores que manipulan ese Modelo, y hay varias Vistas de los datos del Modelo, que cambian cuando cambia el estado de ese Modelo.



Este modelo de arquitectura presenta varias ventajas:

Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado.

Hay un API muy bien definido; cualquiera que use el API, podrá reemplazar el Modelo, la Vista o el Controlador, sin aparente dificultad.

La conexión entre el Modelo y sus Vistas es dinámica; se produce en tiempo de ejecución, no en tiempo de compilación.

Al incorporar el modelo de arquitectura MVC a un diseño, las piezas de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes, posteriormente, se observa que funciona mal, puede reemplazarse sin que las otras piezas se vean afectadas. Este escenario contrasta con la aproximación monolítica típica de muchos programas Java. Todos tienen un Frame que contiene todos los elementos, un controlador de

eventos, un montón de cálculos y la presentación del resultado. Ante esta perspectiva, hacer un cambio aquí no es nada trivial.

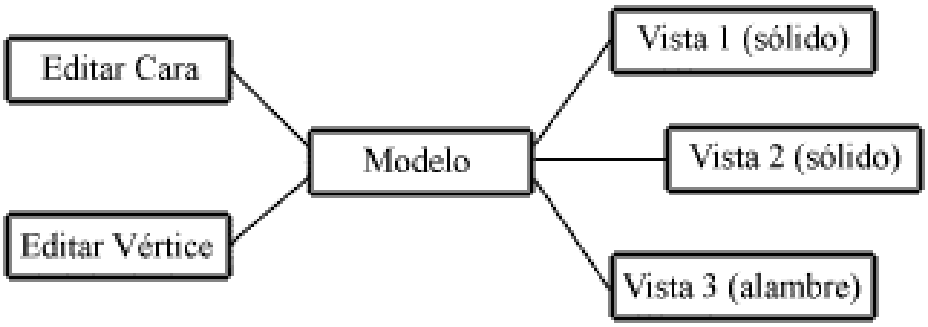
14.1 Definición de las partes

El Modelo es el objeto que representa los datos del programa. Maneja los datos y controla todas sus transformaciones. El Modelo no tiene conocimiento específico de los Controladores o de las Vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el Modelo y sus Vistas, y notificar a las Vistas cuando cambia el Modelo.

La Vista es el objeto que maneja la presentación visual de los datos representados por el Modelo. Genera una representación visual del Modelo y muestra los datos al usuario. Interactúa con el Modelo a través de una referencia al propio Modelo.

El Controlador es el objeto que proporciona significado a las ordenes del usuario, actuando sobre los datos representados por el Modelo. Cuando se realiza algún cambio, entra en acción, bien sea por cambios en la información del Modelo o por alteraciones de la Vista. Interactúa con el Modelo a través de una referencia al propio Modelo.

Vamos a mostrar un ejemplo concreto. Consideremos como tal el sistema descrito en la introducción a este capítulo, una pieza geométrica en tres dimensiones, que representamos en la figura siguiente:



En este caso, la pieza central de la escena en tres dimensiones es el Modelo. El Modelo es una descripción matemática de los vértices y las caras que componen la escena. Los datos que describen cada vértice o cara pueden modificarse (quizás como resultado de una acción del usuario, o una distorsión de la escena, o un algoritmo de sombreado). Sin embargo, no tiene noción del punto de vista,

método de presentación, perspectiva o fuente de luz. El Modelo es una representación pura de los elementos que componen la escena.

La porción del programa que transforma los datos dentro del Modelo en una presentación gráfica es la Vista. La Vista incorpora la visión del Modelo a la escena; es la representación gráfica de la escena desde un punto de vista determinado, bajo condiciones de iluminación determinadas.

El Controlador sabe que puede hacer el Modelo e implementa el interface de usuario que permite iniciar la acción. En este ejemplo, un panel de datos de entrada es lo único que se necesita, para permitir añadir, modificar o borrar vértices o caras de la figura.